Increasing Operational Awareness using Monitoring-Aware IDEs

Master's Thesis

Jos Winter

Increasing Operational Awareness using Monitoring-Aware IDEs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Jos Winter

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
https://www.adyen.com/

Increasing Operational Awareness using Monitoring-Aware IDEs

Author: Jos Winter
Student ID: 4290356
Email: jos.winter@student.tudelft.nl

Abstract

It is important to detect problems fast and to have a clear overview of what is happening within a system after deployment to maximize the uptime and functional quality of the system. Therefore it is necessary to increase the awareness that developers have of errors and logs. Increasing the awareness that developers have of errors and logs has a positive impact on finding problems and solving them. This thesis aims to use monitoring information to bridge the gap that exists between development and operations. We propose to do this by linking the logs to source code to provide this missing link between development and operations. We provide a theory for a monitoring-aware IDE which aims to tackle some of the challenges and enhance some of the practices that exist in the field. We implemented a monitoring-aware IDE and performed a field study to measure its effect. Our results show that a monitoring-aware IDE assists the developer in understanding the system, fixing problems in the code, and improving the monitoring code.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University Supervisor:	Dr. M. F. Aniche, Faculty EEMCS, TU Delft
Company Supervisor:	Ir. C. Borlovan, Adyen B.V.
Committee Member:	Dr. W. P. Brinkman, Faculty EEMCS, TU Delft

Preface

This thesis has been submitted for the degree of Master of Science in Computer Science at the Delft University of Technology.

First of all, I would like to thank the Delft University of Technology, my teachers, and the other students for everything I have been able to learn during these five years of studying. While I was working on this thesis, I had a lot of assistance from my university supervisor, Maurício, with whom I had great discussions and collaboration. I want to thank him for all the support and guidance he provided me during the past nine months. I would like to thank Adyen for providing me the opportunity to write my thesis with them. During my time here I had a lot of fun with all my colleagues. I am grateful to everyone who helped me by participating in the interviews, participating in the field study, and using the monitoring-aware IDE. Especially I would like to thank my company supervisor, Călin, for all the discussions we had and all the advice he provided me. I would like to thank Jürgen Cito for listening to my ideas and the insights he provided on my thesis. I would also like to thank the other members of my thesis committee, Arie van Deursen and Willem-Paul Brinkman, for their time and efforts.

Finally, I would like to thank my family and friends for all their unconditional support.

Jos Winter
September 7, 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In large software systems, logs are generated at a large scale which makes it difficult to have a good overview of what is happening. For most companies that maintain large software systems, it is important to detect problems fast and to have a clear overview of what is happening within their system after deployment to maximize the uptime and functional quality of the system. Therefore it is necessary to increase the awareness that developers have of errors and logs. Previous studies that evaluate observing large software systems show that increasing the awareness that developers have of errors and logs has a positive impact on finding problems and solving them [14, 39]. DevOps is defined as "a set of practices that are trying to bridge the developer-operations gap at the core of things and at the same time covers all aspects which help in speedy, optimized and high-quality software delivery" [29]. Cito et al. [9] coin the term feedback-driven development as "the integration of runtime monitoring data from production deployments of the software into the tools developers utilize in their daily workflow to enable tighter feedback loops.".

This thesis aims to use monitoring information to bridge the gap that exists between development and operations. Shang [34] proposes that the divide between software developers and operators should be bridged using logs. We propose to do this by linking the logs to source code to provide this missing link between development and operations. By showing this link, operation data gets integrated into the development process and using this integration we want developers to get a better overview of what is happening in the system.

We interviewed developers within the company to understand the challenges and practices that exist within the field. Based on the collected information we provide a theory for a monitoring-aware IDE which aims to tackle these challenges and enhance these practices. We implemented this theory and developers used it. Their interactions were measured and evaluated to determine the impact on their behavior. Our results indicate that monitoring-aware IDEs assist the developer in understanding the system, fixing problems in the code, and improving the monitoring code.

The main contributions of this thesis are that we provide a detailed description of challenges and practices for log analysis, a theory that IDEs and monitoring should be combined as a monitoring-aware IDE, and a field study within a large organization using monitoring-aware IDEs to see its effect. In addition to submitting this thesis, we also submitted a paper to ICSE 2019. Therefore parts of this thesis are reflected in the submitted paper.

Chapter 2

Background and Related Work

DevOps and Continuous Integration are popular in the industry and as a response to that, a lot of research has been performed to uncover the best practices, advantages, and disadvantages of it. When adopting DevOps it is critical and a best practice to have measurement and logging present in the system which allows for efficient monitoring, analysis and debugging of the operational system. In response to this, tools have been developed which show operational analysis in the IDE. The generation and analysis of logging is a well-studied research topic by industry and academia as it is the main tool for finding problems and their causes in operational systems. The challenges of logging and existing tools for log analysis are explored. Because of the scale on which software systems operate, the scalability of logging and log analysis is an important topic. Techniques exist for the extraction of log patterns from the source code and pattern matching provides a scalable approach to detect the origin of log messages using these log patterns.

2.1 DevOps

Organizations are interested in using DevOps but it remains difficult to apply DevOps successfully in practice. Smeds et al. [36] investigated and identified these burdens by means of interviews and a literature study. The main difficulty in adopting DevOps in the industry is related to its unclear definition and therefore results in incorrect adoption. Bass et al. [1] define DevOps as “a set of practices intended to reduce the time between committing a change to a system and the change being placed into normal production while ensuring high quality.”. Nagpal and Shadab [29] define DevOps as “a set of practices that are trying to bridge the developer-operations gap at the core of things and at the same time covers all aspects which help in speedy, optimized and high-quality software delivery”. To address the problem of the unclear definition of DevOps its aspects such as the concepts, benefits, practices, and challenges are identified. Ghantous and Gill [16] identify these attributes from literature, the main challenges consist of constructing the tool pipeline and overcoming the mentality between development and operations.

In addition to these aspects of DevOps, tools have been created to address other challenges that have arisen in DevOps. One of such problems is maintaining the runtime performance of a system. Waller et al. [38] aim to detect performance issues early using tooling to prevent additional performance decrease as a result of further development on top of these performance issues. Kunz et al. [22] detect these performance issues by constructing a performance model from operations data. In comparison Cito et al. [12] detect these performance issues by identifying whether common causes are present in the system. In contrast to decreasing the cost of the system by evaluating the runtime performance, the price of resource and capacity allocation is evaluated. Resource and capacity allocation has been investigated with the aim of minimizing the cost of the system. Leitner et al. [23] constructed a tool which can evaluate the cost of the system while being computationally cheap. These tools are introduced into the build system to automate the evaluation in the development process.

2.2 Continuous Integration

Manglaviti et al. [26] define continuous integration (CI) as “a software development practice where changes to the codebase are compiled and automatically checked for software quality issues.”. Motivations of CI consist of enabling rapid and high-quality software development and delivery [4] reports based on empirical analysis. Consequently, DevOps and CI have overlapping motivations as both enable rapid and high-quality software development and delivery. Based on the literature Smeds et al. [36] identify CI as one of the main capabilities of DevOps and therefore CI is one of the best practices and main requirements for successful DevOps [36].

Hilton et al. [20] analyze the usage of CI in open-source projects to understand its usage. Their results support the claim that CI helps projects release more often. In contrast to this Hilton et al. [18] analyze the usage of CI in proprietary projects to understand how it is used in open-source projects by interviewing and surveying developers. There is reported that the observed test quality and productivity level of developers increases as a result of CI. In contrast to open-source projects, a significant part of developers want to adopt CI but are not allowed to do so for proprietary projects. One of the observed main challenges of CI in proprietary projects is the difficulty of constructing and maintaining the build system [18, 19]. Build systems are the key component of CI as they are used to define the build process and the tools used for evaluation. Because of the automated nature of build systems and the rapid release cycles which are often used in combination with CI, there is overhead introduced on the software development process [27]. In addition to the difficulty of configuring a build system, there is observed that the runtime performance of the build system has an adverse effect on the development process. They identified that the effect of this challenge could be mitigated by only rebuilding parts of the system which are changed in the development process.

2.3 Feedback-Driven Development

Cito et al. [9] coin the term feedback-driven development (FDD) as “the integration of runtime monitoring data from production deployments of the software into the tools developers utilize in their daily workflow to enable tighter feedback loops.”. FDD is becoming more and more relevant to use due to the rise of CI. To successfully adopt continuous integration, developers need better means to anticipate runtime problems [33]. Metrics, therefore, need to be provided with improved fault localization to make sure that the number of problems in each deployment can be mitigated. In contradiction to this, Cito [8] observed that a significant amount of developers instead work based on their intuition and do not use metrics when it is unintuitive to access them. Metrics should be made more easily accessible for developers to tackle this problem [8].

Cito et al. [9] divide FDD into two classes: analytic FDD and predictive FDD. Analytic FDD aims to bring runtime feedback from production to the IDE. When local changes are made in the IDE, the runtime feedback from production is not representative anymore for the local code. Predictive FDD aims to predict the feedback when such local changes happen in the IDE. When changes are made to production, then all runtime feedback from before that change is not representative anymore for the production code. Cito et al. [9] use the term feedback freshness to define when unrepresentative feedback should not be considered anymore. Cito et al. [12] use changepoint analysis to identify the feedback freshness. Changepoint analysis can detect fundamental shifts in the data instead of detecting spikes in the data. When local changes in the IDE are made, then there is no feedback available. Using the changes in the dependency graph the new values can be predicted using the already collected feedback for the connected nodes. This prediction can be performed using statistical inference and feedback propagation.

Cito et al. [11] found that runtime information that can be used as feedback is scattered when troubleshooting a specific problem. To tackle this problem Cito et al. [11] aim to construct a graph structure which unifies runtime traces and source code. Using this graph structure runtime information from multiple sources can be organized through a multitude of tools. Cito et al. [11] compare other approaches which aim to solve traceability between textual artifacts and source code with their approach. When developers use this graph structure the number of analysis steps to solve a problem is significantly reduced. Cito et al. [9, 10] found present realizations of FDD which are focused on measuring the performance of a system and detecting the most expensive parts. Alternative realizations are focused on measuring the cost of a system to minimize its deployment cost [23].

2.4 Logging and Log Analysis

The generation of logs and the usage of logs is well studied. Logging in a system is a simple and effective debugging tool and can be applied in any location in the code. Developers generate logs in key locations to understand the state, execution sequence and the presence or absence of certain events. However, challenges exist which make it difficult to use logging in a system effectively. Some difficulties and challenges related to logging have been identified in literature.

The effectiveness of logging with respect to debugging and diagnosing of failures is well studied. Log statements, in general, can be improved to make debugging more easy and reproducible. Most of the time, these error conditions cannot be reproduced, or it is not feasible to run the system again using the exact same scenario because of a lack of information. This challenge can be solved by adding the missing relevant information to a log statement. Cinque et al. [6] found that missing relevant information can be detected proactively to improve the logging statements in the code. Yuan et al. [42] use log enhancement to add contextual variables to log statements so generated log messages can be used to reproduce scenarios. This enhancement can have a negative effect on the logging performance because log statements become more complex and log message size increases. However, Yuan et al. [42] applied this approach to real projects and measured that 95% of the found contextual variables are already included in the log statements. Similarly, difficulties are found in detecting and diagnosing software faults by means of logs because a significant part of software faults do not generate a log message [6]. For example, some systems assume a simplistic error propagation model and therefore critical errors that distort the control flow make it impossible to log [7]. When working with large complex software systems, it is not feasible to achieve full coverage as logging at every possible failure location is inefficient. Cinque et al. [6] and Pecchia and Russo [32] provide a set of guidelines which help in increasing this coverage in critical places. In response to these guidelines, multiple tools have been implemented which aim to solve this challenge. Yuan et al. [41] constructed a tool which indicates positions in the source code which could be improved by adding a log statement. The evaluation of this tool showed that the addition of these log statements barely increases the logging overhead and significantly speeds up the failure diagnosis. Fu et al. [15] studied common logging practices especially focussing on where developers log, they conclude that common logging practices can be used to automate the logging process. Zhu et al. [43] implemented a tool which learns the common logging practice and uses it to indicate positions that can be improved by adding a log statement. In addition, Cinque et al. [7] aim to solve the same problem using a rule-based approach. Their results show a significant increase in the number of software failures that are covered by log messages.

Because of the size and evolution of complex software systems the size of the generated log messages constantly increases which makes it difficult to store all log data and to find relevant information in the logs. As the size of the log message set increases, it becomes increasingly important to limit the number of irrelevant log messages [2]. Chen and Jiang [3] studied anti-patterns which are defined as recurring mistakes in logging code which may hinder the understanding and maintainability of the logs. Anti-patterns which decrease the logging quality of a system were found from common logging practices and in response a tool was constructed which detects these anti-patterns. By removing log statements that are identified as anti-patterns, the quality of the logging code can be improved, and the size of the generated log messages can be decreased.

Logs are typically represented as multiple log messages which happen within a certain temporal ordering. There are challenges in preserving this chain structure as interleaving occurs between the log messages of multiple sources, multiple threads, and multiple modules [2, 30]. Depending on the context of the log messages the temporal ordering can be recovered. However, on a large set of logs, it is not scalable to detect the temporal ordering of related logs. Evers [14] uses a cache to store all log messages in a certain interval and only compares log messages which are generated in close temporal proximity, using contextual identifiers there can be detected whether two log messages are part of the same chain. In addition to these logging challenges and their improvements, multiple applications of logging and logging analysis have been investigated. The main goals of logging analysis are assisting the developer in finding bugs, increase the developer confidence due to the absence of certain bugs and problems, and increasing the developer awareness and understanding of the system. Purposes of log analysis are optimizing and debugging the system performance, detecting and inspection of security breaches and misbehavior, profiling the system and resources, and prediction of the system and resources [2, 31].

One of the main applications of log analysis is the localization of faults. Cinque et al. [6] provide guidelines to improve fault localization using log analysis. In addition to using logging data for the localization of faults, Cinque et al. [5] evaluate multiple direct monitoring techniques consisting of event logs, assertions, and source code instrumentation. They analyzed the ability of each technique to detect injected software faults. Their results show that each technique has certain specialties in finding faults of a specific type. Therefore a combination of the techniques could be used to improve the reporting of software failures.

Because of the volume of log data in large software systems, not every log can be observed by developers and therefore some filtering or prioritization has to be applied to make sure that all relevant logs are observed. Multiple implementations of log clustering have been studied to solve this problem. Lin et al. [24] applied log clustering at Microsoft by assigning each new log message to an existing or a new cluster. The generation of these clusters in combination with limited developer input can be used to efficiently determine the priority of new log messages. A similar log clustering approach has been applied by Evers [14]. This clustering method proved successful and is still being used and assists in finding problems which previously would not be noticed as fast.

Logging data can contain valuable information related to security breaches that occur within the system. Cotroneo et al. [13] look into tracking of security-related events, logging of events related to security issues, and the identification of security breaches. It is important to be able to check if a security breach has occurred. In addition, it is also relevant to be able to detect who the attacker is and how the system was breached so the security can be improved at critical positions. Cotroneo et al. [13] propose a framework consisting of a method which identifies security breaches using large volumes of security alerts by filtering them and applying a decision tree to the data.

2.5 Log Pattern Detection

For fault localization and other log analysis, it can be helpful to track the origin of log messages to a certain position in the source code.

Xu et al. [40] created a method to efficiently extract log patterns from the source code using the AST [28]. Log patterns are extracted by generating the AST and by performing a partial evaluation to get the most accurate and unique log patterns. If no partial application is used, a log statement which logs a string variable will always have the same single wildcard pattern, and therefore message patterns will likely not be unique. Some challenges exist in the generation of these log patterns. When the codebase of a system evolves, it becomes increasingly difficult to maintain the logging code. As an effect duplicate log messages or duplicate log patterns can be introduced to the system. When duplicate log patterns exist in the system, there are multiple possible origins for a certain log message and therefore the accuracy of matching log messages to log statements decreases. When a system depends on unavailable or inaccessible source code, log messages can exist which cannot be linked to their origin because no log patterns are generated for their source [17].

Log patterns can be detected from the source code but to find the corresponding source of a log message the respective log message needs to be matched to all log patterns. Using contextual filtering, most log patterns can be ignored, for example by using the class where a log message has been generated. The matching of log patterns or regular expressions to log messages or strings is an essential part of the method that is applied in this thesis. Regular expressions and the matching of these expressions to strings is well researched, and efficient methods have been applied for doing this. Depending on the requirements of a system there are multiple options for pattern matching with different space and time complexities. In general there holds that DFAs are time-efficient but space-inefficient, and NFAs are space-efficient but time-inefficient [37]. Sidhu and Prasanna [35] use FPAs for pattern matching. For this method, the time complexity of matching a regular expression with length n to a string with length m is $O(n+m)$ and the space complexity of matching the same instance is $O(n^2)$. As the length of log messages and log patterns are relatively short, the computation time of determining whether there is a match is computationally inexpensive.

2.6 Monitoring Tools

There are also a vast amount of monitoring tools that have originated in industry. Most tools display metrics (often extracted from information in logs) in dashboards that are customizable in different dimensions (*e.g.*, visualization, groupings, alerts) and are searchable. Probably the most prominent open-source toolchain in the context of monitoring is the ELK stack[1] (ElasticSearch, Logstash, Kibana) where logs from distributed services are collected by Logstash, stored on ElasticSearch, and visualized in Kibana. Another well-known open-source dashboard is Grafana[2], that is mostly used to display time series for infrastructure and application metrics with an extensible plugin system. Commercial counterparts to these services include, for instance, Splunk[3], Loggly[4], DataDog[5], and many more.

2.7 Adyen

Adyen is a payment service provider (PSP) which means that they offer worldwide services to their merchants which are businesses. These services consist of handling all customer payments for a merchant, providing over 250 different payment methods which include

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

[2] https://www.grafana.com/

[3] https://www.splunk.com/

[4] https://www.loggly.com/

[5] https://www.datadoghq.com

local payment methods and more services related to payments[6]. Because Adyen has merchants all over the globe and payments are being processes continuously the uptime, and the successful functionality of the system is of the highest importance for Adyen. Logging is used to monitor problems related to the uptime and processes that can affect it. As the Adyen platform is relatively complex and large and because it is distributed across many systems it is difficult to overview all the logs generated by the whole system. In addition, it is hard to debug problems that happen in a production setting as local ways of debugging are inapplicable. Therefore the standard way to track important runtime information has become logging.

The main development environment used at Adyen is IntelliJ[7] which they use to implement their system in Java. The standard logging library which is used for their Java systems is Log4j. The configuration files used for Log4j specify that the current logs include additional information about the time when the log message has been generated, which class in the code generated the log message, the severity of the log message, and identifiers for the thread and the payment service provider. When any system generates logs they are sent to a central point, the ELK-stack which stores the logs. The ELK-stack consists of ElasticSearch[8], LogStash[9], and Kibana[10] which are responsible for storing, organizing and searching the log messages. These systems are responsible for handling approximately a billion log messages each day. These log messages can be divided into four categories based on the severity level that has been provided by the developer which implemented the log statement:

- Info messages are used and generated most often as they are used to indicate important information and events which are used to monitor and track the status of processes. Approximately a billion log messages are generated with the info severity per day.

- Warning messages are generated considerable fewer times as a warning indicates a serious issue in the code base which should not be able to occur. Approximately two million log messages are generated with the warn severity per day.

- Error messages are generated the fewest as the error log severity is used for issues that definitely should not be able to happen and which effects can be harmful. Approximately 25 thousand log messages are generated with the error severity per day.

- Debug messages are only used and generated in a local environment. Therefore, there can be no log messages generated by the system in operation with the debug severity.

As the deployment and updating of the system are complex, there are a lot of risks for the production environment. To mitigate this risk, a feature framework is used where features

[6] https://www.adyen.com/

[7] https://www.jetbrains.com/idea/

[8] https://www.elastic.co/products/elasticsearch

[9] https://www.elastic.co/products/logstash

[10] https://www.elastic.com/products/kibana

can still be turned off and on after deployment to guarantee the uptime of the system. The feature framework enables the functionality of the platform and allows for fast iterations. The possibilities for monitoring the system are limited because of the limited data that can be retrieved from the ELK-stack as it should not be overloaded. It would be expensive and insecure to query all logs from the ELK-stack. As there are around a billion log messages generated a day there are approximately ten thousand log messages generated each second on average. It is not possible to query such large amounts of data from the current log pipeline. The log pipeline could be improved to allow for a different way of handling traffic and querying all log data. These changes would be fundamental to the infrastructure of the logging system which could introduce many difficulties and risks. As a result of these limitations, the scope of the project should be limited. There are multiple solutions to limit the scope of the project and the data:

- Instead of considering all log messages from all classes in the code base, only a specific subset of the code base and classes within it could be examined. Wieman et al. [39] only considered a subset of the code base to limit the amount of data that is requested from the ELK-stack.

- Alternatively, only log messages could be considered which have a warn or error severity level as the amount of warning and error messages is much smaller than the number of info messages, in addition to this, the importance and priority of warning and error log messages is much higher. Evers applied the same technique within Adyen to limit the amount of data that is requested from the ELK-stack [14]. Research has been done concluding that logs with the warn and error severity level are more likely to be useful for assessing and diagnosing problems [41]. Approximately 25 log messages per second would be generated after filtering on the severity level.

Most monitoring is performed directly on the ELK-stack. However, some existing monitoring tools have been deployed as otherwise not all problems can be detected. Logness, a log analysis tool, generates clusters and uses labels by developers to decide whether newfound problems are relevant or not relevant [14]. A passive learning tool has been applied to the point of sale system to generate graph models from the logs which can be used to compare the paths that are available in the system. Bugs can be found, contexts can be compared, and timings can be analyzed using this graph [39].

Chapter 3

Developers' Experience with Log Analysis

The goal of this chapter is to shed a light on the current practices in logging analysis as well as challenges and opportunities for improvements. In addition, we aim to understand how these practices can be improved and how these challenges can be solved using logging data.

3.1 Research Questions

The purpose of these interviews is to identify which challenges exist in the usage of log analysis tools and which data is relevant for overcoming these challenges and how this data can be used to achieve this goal. To fulfill this purpose the following research questions have been constructed which have been investigated during the interviews:

RQ$_1$ Why is logging data used to perform monitoring and debugging?

RQ$_2$ What logging practices do developers apply to monitor and debug?

RQ$_3$ What information do developers perceive as fundamental for successful monitoring and debugging?

RQ$_4$ What challenges do developers face when using logging data for monitoring and debugging?

RQ$_5$ What improvements do developers identify which can improve these practices and solve these challenges?

The research questions are the basis for the questions which are part of the interviews. This includes the questions stated in the fixed set of interview questions in Appendix A and the questions that are asked during the interview based on the answers received during the interviews.

3.2 Methodology

During the interviews, the focus is to explore what different motivations, practices, challenges, and possible improvements there are. It is not the main focus to collect quantitative data on these topics. Therefore a semi-structured approach is used during the interviews where additional questions may be asked if there are topics that can be uncovered. Semi-structured interviews encourage participants to freely share their thoughts and enable researchers to follow up and explore interesting topics that might emerge [21]. To collect these motivations, practices, challenges, and improvements, 7 participants were interviewed. This section will describe the design of the interview, the processes which are followed during the interview, and the methodology used to generate results from the interviews.

3.2.1 Interview Design

To generate results which can be grouped and compared from this interview we came up with a set of interview questions. The set of used interview questions can be found in Appendix A. The set of interview questions focuses on the experiences of the participants which makes it easier for them to answer the questions in contrast to theoretical questions. If the participant does not provide the motivations for a certain example they will be asked to provide it.

The interview questions in Appendix A are grouped into six categories, based on their purpose and the research question they are answering:

- The four questions in the participant information section aim to collect information about the participant.
- Question 2 from the logging usage section supports answering RQ_1.
- Question 1, 3, 4, and 6 from the logging usage section support answering RQ_2.
- Question 1 from the feedback extension section supports answering RQ_3.
- Question 3, 4, 5, and 8 from the logging usage section support answering RQ_4.
- Question 2 from the feedback extension section supports answering RQ_5.

3.2.2 Interview Procedure

The developers were invited to participate in the interviews through the internal chat including a short description of why the interview is being performed and what it is about. In addition, developers were asked to agree that their interviews will be recorded and if parts of their answers or summaries of their answers could be used in the thesis, which they all agreed to. The interviewing sessions were conducted in person and lasted from 20 to 60 minutes, with an average duration of 35 minutes. Firstly, some questions about the participants were asked. Secondly, the main part of the interview was conducted which is about the logging experiences of the participant indicating the motivations, practices, and challenges. To avoid confusion this part was started with a small introduction on logging analysis tools to make it clear to the participant what is meant with log analysis tools by mentioning some tools which are used within the company. Lastly, the explorative part of possible improvements was conducted. To make the questions more clear a small introduction was provided on the topic of my thesis and its goals.

The data was analyzed by listening to the recordings from the interview and labeling the relevant parts of the recording. The relevant parts were transcribed and annotated with the time of the fragment within the recording. Some irrelevant information in combination with repetitions were removed from the results. These labeled parts for all participants were grouped together wherever possible. This data analysis method results in an overview of statements where you can see for each participant whether they agree and why they agree.

3.2.3 Participants

The 7 participants of this interview were selected from the group of developers at Adyen. The development experience, log analysis experience, and experience within the company for the participants can be seen in Table 31 and Table 32. Developers who are involved in the supervision of this thesis are excluded from being interviewed to avoid bias in the results. For the participant selection, we informed a single developer if they would be willing to participate. If the developer was willing to participate, the interview would be performed and if no new data was found no additional interviews would be performed. Using this method of participant selection we made sure that all the motivations have been explored for the constructed research questions. No additional interviews are conducted when the data becomes repetitive and saturation occurs.

Code	Development Experience (Years)	Log Analysis Experience (Years)
P1	15	1 (7%)
P2	2	2 (100%)
P3	10	5 (50%)
P4	2.5	2.5 (100%)
P5	7	2.5 (36%)
P6	10	4.5 (45%)
P7	12	6 (50%)

Table 31: The codename of each participant and the experience each participant has in terms of development and log analysis

	Low	High	Average
Development Experience (Years)	2	15	8.4
Adyen Experience (Years)	1	4.5	2.3
Log Analysis Experience (Years)	1	6	3.4

Table 32: Experience of the participants, showing the lowest, highest, and average answer provided by all participants

3.3 Results

This section presents the results generated by the interviews. The reasons for using log analysis, the perceived challenges, and the way this process can be improved are described.

3.3.1 Why is logging data used to perform monitoring and debugging? (RQ_1)

The participants pose multiple reasons for using log analysis tools and there exists an overlap between their reasons.

Log analysis tools can be used to trace business processes. All participants agree that one of the reasons for using log analysis tools is the traceability of business processes (P1, P2, P3, P4, P5, P6, P7). When logging a reference together with the log, all the logs from beginning to end for a certain business process can be easily found and grouped together.

All data from multiple sources can be stored, filtered and aggregated in one place. Adyen works with distributed systems where logs from a single business process can be scattered across multiple machines (P2, P3, P5, P6). P3 says "*Because our systems are multi-threaded, distributed and multiple applications are involved, log analysis tools allow for observing all log data for distributed processes. Aggregating and filtering in one single place makes it much faster than searching through the physical log files*".

Log analysis tools provide feedback on your code. By adding debug information to logs there can be investigated whether something is going wrong and what exactly is going wrong (P1, P3, P4). P1 says "*It is an iterative process of getting feedback on my code. Making some little changes and see if it works and if it does not work, see where things went wrong*".

Log analysis tools can detect changes between versions and potential problems. Problems that happen after deployment that did not occur before are important as the cause of the problem is likely caused by changes between the versions (P2, P5, P7). P5 states that "*Log analysis tools can be used to check if the behavior of the system changed between two different versions and that way can identify problems that were introduced by changes and filter out problems that already existed within the system*".

Clustering the logs can be used to filter out the unimportant data. There is stated that there are too many log messages in the system and because of that, you do not know which logs are important. The most important logs are not always the logs that occurred the most, there exists a risk that problems which occur less frequently might remain undiscovered. P2 enforces this: "*If you only look at the trends you can not immediately see which logs are important*". Using clustering there can be detected which logs occur the most but are not important (P2, P5, P6). By filtering out the unimportant logs which occur frequently it is easier to find important logs.

Log analysis tools can provide automated continuous analysis. When using automated log analysis tools there is no need for the focus of the participants when it is not necessary (P2, P3, P6, P7). P2 states: "*We occasionally build some monitoring on top of the logs which notifies us when some behavior is happening which is uncommon*". P3 enforces this by saying "*It is very simplistic but it allows for some automation so I don't need to physically look into the logs*".

3.3.2 What logging practices do developers apply to monitor and debug? (RQ$_2$)

The participants use multiple log analysis tools with multiple different practices for monitoring. They changed their logging practices so that it becomes easier to use the log analysis tools for monitoring.

The most simple form of log analysis that is being used is manual log analysis. This form of log analysis is not automated and heavily manual (P1, P3, P4, P5, P6). Using manual log analysis, the logs can be searched on a local machine with simple commands (P1). The same principle can be applied to a machine that is live (P5, P6). In addition, you can search in the database both locally and live which can be used to deduce what happened within the system (P3, P4, P5).

There can be effectively searched and navigated through the logs using a custom search engine. All participants make use of logsearch, a search engine that can be used to search through the total set of logs from all machines in the distributed system and in addition provides contextual information and links between these logs (P1, P2, P3, P4, P5, P6, P7).

The logs are monitored during the deployment to detect unexpected behavior. Patch monitoring is being used to detect changes between versions and potential problems (P2, P5, P6, P7). P2 says "*Patch monitoring shows which log messages are occurring for the first time after patching. So it compares a short period before a patch with the activities that occur after the patch and detects what behavior stands out*".

The logs are clustered to detect new problems as fast as possible. Data science is being used to detect bugs that did not happen before by clustering the log messages where the creation of a new cluster represent a new bug happening in the system, its goal is to detect unknown issues as fast as possible (P2, P5, P6). P5 says "*Clustering the logs is useful for seeing whether new errors are happening within the system*".

The logs are also being used for company-specific monitoring. Important information with business value can be extracted using log analysis tools (P4, P7). P7 says "*Log data is being used as one of the sources for our data science platform on which custom logging analysis tools are deployed because they are fitting every purpose*".

The interview participants detect software faults in their code by different means and they share challenges that they observe in the process of detecting their software faults. The following practices for monitoring and identifying software faults have been reported. In addition, the participants indicated multiple improvements they incorporated into their logging behavior to benefit monitoring the system.

Participants monitor their own changes after deployment or are notified that their changes are not functioning correctly. The participants monitor the behavior of their own changes after it is deployed (P1, P2, P3, P4, P5, P6). P2 says "*When my changes are deployed on the testing environment I test whether my changes are functioning correctly by monitoring for specific logs and logs from specific classes*". P4 enforces this by saying "*If the change has a high chance of not working I monitor it by specifically adding a log there. I basically just monitor the class and see if it behaves abnormally*". Alternatively, it happens that the participants are notified about a software fault that happened in their changes by other people within the company (P1, P2, P3, P4, P5, P6). This especially happens when software faults are happening during the monitored deployment. P5 says "*It happens that I'm not actively monitoring during a deployment and someone else notifies me about a software fault I introduced. Other developers are not always aware when their code is going live and do not proactively check if it is working*".

Multiple stages of testing and detection are used before going live with changes. The use of these testing stages can happen in combination with the practice indicated above (P2, P3, P4, P5, P6, P7). The goal of using these multiple testing stages is to mitigate the risk of deploying faulty changes. P5 says that "*There are multiple staging environments test and beta. We expect all developers to test their own changes. The behavior of the system is monitored when changes are deployed to beta and live.*" P7 adds to this that "*When something faulty has been implemented and errors are being thrown during deployment, a tool that I wrote, automatically detects the problem if it has not happened before*".

The severity of a log statement is changed when they are not used as intended. The log level of a log statement is being changed whenever there is detected by the participant that it has the wrong severity level (P3, P6). P3 explains "*The severity of log messages is important, error level messages, for example, result in direct messages to the people who are monitoring the system which should only be used for reasonable problems*". P6 enforces "*It is nice to have info, warn, and error log messages but if you are not using them as intended it has no use. Errors should be something that you can actively fix. Warnings should be something that is kind of odd but could be ok. Info is used to look up some information that we really need*".

Log statements are added where incorrect functionality is not detected. Some code in the system is not being monitored and logging is added to this part of the system to make sure incorrect functionality can be detected (P2, P6). P6 says "*It also happens that a problem cannot be reproduced because there is not a lot of logging or there are no stack* *traces being logged. Then you have to add some log lines and patch the system to see what is going on in the system*". It can also happen that something is logged in the wrong place (P4). When this happens the participant moves the log statement to a different position, beforehand it is not always clear where to log something.

Log statements are formatted in a specific way so the data can be aggregated for business specific goals and monitoring. Some participants report that some logs should be constructed in such a way that is it easier to aggregate them (P2, P6, P7). This especially applies to use cases where structured logging can be used from other frameworks where the data is being used for business specific goals. P6 explains "*Some log messages related to payments are changed to a structured format to make the messages more searchable*".

In contrast to detecting software faults using log analysis tools, these tools can also be used for debugging purposes. Multiple practices for using log analysis tools for finding the origin of a software fault are reported and challenges that are observed are provided. The following practices for finding the origin of a software fault have been presented by the participants. Multiple improvements have been incorporated into the logging behavior of the participants to benefit debugging the system.

Contextual information can indicate which conditions triggered the problem. Some of the participants report that it is important to log enough contextual information (P2, P4, P6). Contextual information can be added to log messages in the form of variables. P6 states the reason for this practice: "*Using a log message you can find where it came from and what contextual information was involved, if you can reproduce the problem locally using this data then there are no issues when debugging*". When a log message is generic it becomes harder to find the origin of a problem and there is no contextual information provided which might indicate the cause of a problem (P1, P2, P3, P4, P5, P6, P7). It is important that specific contextual information is added to the log message. P1 says "*Try not to implement too generic log messages, always put some variables inside it which are related to the event and what possibly went wrong*". However, it is also important that not too much contextual information is added as noise to the log message to make monitoring and debugging easier (P2, P4).

The traceability between business processes can be used for debugging. All participants indicated that it is important to log a reference which links the log to other related messages (P1, P2, P3, P4, P5, P6, P7). Using this reference there exists traceability between the logs in a single business process. This traceability can then be used for finding the origin of a software fault. All participants try to add a reference that can be linked to other log messages whenever possible. P5 describes "*Often there is a reference or identifier logged which can be searched in logsearch to see which events happened before and after this event and that way you can see where it went wrong*". P3 reinforces this "*Having other log messages which preceded this event I can fill in some missing puzzle pieces to deduce what exactly caused a software fault*".

The origin of the log can be found in the source code using the message and the class of the log. In addition, using the log message the participants try to find the corresponding log statement in the source code (P2, P3, P4, P6). P6 describes "*If you have an issue you should hope there is some logging for it, see if you can look up the log in the code base, and tackle what has happened. It is useful to be able to check where a log message originates from*". P4 says "*I copy-paste the text in the log message and I try to see which class outputs it and where it got logged in the class*".

The origin of the problem can be found using the stack trace. The most simple practice for finding the origin of a software fault is looking at the stack trace (P2, P3, P6). P2 states "*With a huge stack trace it is straightforward to find where some error occurred*".

Missing stack traces are added to a log statement whenever an exception occurs. The stack trace should be logged whenever a critical exception happens (P5, P6). P6 enforces "*It can happen that the stack trace is omitted from the logs, then it is harder to find out what is happening. Instead you then lookup other logs to see what else happened to get a complete view. If there is an exception going on then try not to just throw it but first log it because then you know the place where it happened*".

3.3.3 What information do developers perceive as fundamental for successful monitoring and debugging? (RQ$_3$)

To tackle the experienced challenges and to improve the current practices, the participant indicated which logging data is valuable when monitoring and debugging the system.

The message within a log is the most basic piece of valuable knowledge. The message of a log contains information about the reason for why the log has been logged and often contextual information (P1, P3). When searching for the origin of a software fault the message is used when there is no stack trace. P3 says "*Using the message you can see with the context which logic happened and in addition, you can possibly jump to the corresponding source in the IDE for a certain log message*".

The severity of the log indicates the urgency of the message and the intention of the developer. The severity of the log message is reported as valuable (P4, P5). P4 says "*We have info, error, and warning severity messages, the severity is an important measure of the importance of a log message*". P5 enforces "*I think that all the warnings and errors are especially useful in the IDE and therefore the severity of the log is important data*".

The distribution of logs is a measure of importance. The distribution of log messages is seen as valuable logging data because it tells something about the importance of a log just like the severity (P1, P3, P4, P5). The same applies for stack traces and the distribution of those, stack traces which did not happen that often before can be important (P2, P6). P1 describes "*A log message is important when it used to happen less frequent but then occurs* *more often*". Similarly, P2 says "*Stack traces which are new or which occur a lot based on the module they are contained in are valuable as feedback*". P3 adds "*How often does the log message happen, that may give some information about bottlenecks in the system*".

Whether the code where a log originates from was recently modified is seen as useful. If the code where the log originated from was recently modified it can be useful for monitoring and debugging as the problem is likely caused by the changes (P1, P2, P6). In addition, it is useful to know who modified the code for the last time. P1 describes "*Based on version control data you can check what change broke certain functionality and who broke the functionality*". P6 enforces "*I need to find out where in the code the problem happens and who did it, that is something that you do anyway*".

The machine a log originates from is fundamental logging data. The machine or resource a log message originates from is seen as valuable logging data as this might be related to the problem (P1, P5, P6). P1 describes "*Based on the used resources you might need to search differently. Some errors are thrown not from the code but from another resource which can be possible be found in the log message*". P5 adds "*Another interesting question to answer is on what server or machine certain logs are happening. Does it just happen on one server, multiple servers or all of them?*"

The log information from the class you are working on is seen as valuable. When the system is being monitored, the system is observed on a class level (P4). Therefore logs that are occurring in other parts of the class might be relevant for you. P4 says "*When I'm starting to work on a certain class and I see a lot of errors happening, I might then be more interested in first fixing these problems before using the functionality in other parts of the code. For example, it would be useful to know which logs were logged recently for a class*".

The timestamps of logs can be used and combined for performance profiling. The time of the message is considered valuable and it can be combined with other messages for performance profiling (P3). P3 reports "*By comparing the time between log messages you can measure the amount of time between them which is a performance measure, this way it can be a profiling tool*". In addition, the first and the last timestamp a certain log occurred is valuable context (P5). The age of a log occurring can be related to its importance.

3.3.4 What challenges do developers face when using logging data for monitoring and debugging? (RQ$_4$)

When monitoring and debugging the system using the identified practices certain challenges arise according to the participants. Multiple challenges that have been experienced by the participants when using the development environment and log analysis tools simultaneously have been reported. In addition to the reasons why participants use log analysis tools, there was also reported what the challenges and the reasons are for not using some of the log analysis tools.

If a software fault is not detected immediately then it gets detected by the end user. Some participants also indicate that it can happen that software faults are not detected in the pre-deployment stage and if a software fault is not detected immediately you have a big problem (P2, P5). P5 says "*If a fault is not immediately detected it will take considerably longer before it is detected because there is no logging for that specific problem*". If there is no logging code present to indicate a specific problem, the problem will not be detected and no logging code will be added until the problem has been found. When such a scenario emerges P2 states that "*Emergency patches are applied if a fault has been deployed on the live system*". P6 indicates that log analysis tools can be behind with the data because it must be collected from multiple sources and aggregated.

Developers are only interested in monitoring the part of the system which they are responsible for. All participants that indicated that they are not using some log analysis tools justify this by not being directly responsible for the health of the system but for the health of a specific subset of the system (P2, P3, P4, P5, P6). P3 says "*Some log analysis tools are only useful for monitoring the system and therefore they are only relevant for the people who are on duty. They need to be aware of all the irregularities in the logs*". P5 enforces "*A big part of the developers are not responsible for the overall health of the system*". P2, in addition, commented that it is just not fun to use some of the monitoring tools because "*No one likes cleaning up other people's mess*" (P2).

When context switching between developing and monitoring you lose focus. Some participants report that when they are monitoring their changes and the changes of other people they lose focus (P2, P3, P5). This focus is lost because there are a lot of context switches, for example, there can be switched from developing to monitoring and back to developing again. P2 says "*If you do a lot of monitoring you lose a lot of focus which is needed to develop intensively on a project, a lot of distraction occurs when monitoring*". P3 describes "*Monitoring always costs focus, some of the problems that I find are not caused by me but are related to the interaction with the changes from other people*". P5 enforces "*I think you lose focus when monitoring very often. For example when I debug some functionality and check the logs to see what happened this results in a lot of context switching which results in loss of focus*".

Log analysis tools need time to aggregate logs which results in a delay. Lag is a problem for log analysis tools especially when deploying and patching because of the importance of monitoring faulty functionality. P6 states that "*Some log analysis tools can be behind with the data so, therefore, it is not a dependable monitoring tool. To tackle this problem a separate monitoring tool has been deployed which tracks the logs for unexpected behavior that way we can rely on it instead of looking it up ourselves. Other tools need some processing time before all the data is available.*"

If the functionality of the system depends too much on configurations it is hard to understand what should be happening. P5 mentions that some changes are not automatically enabled but must be enabled in the configurations. These configurations are introduced when a developer implements something which might break when deployed. When faulty behavior is detected these changes can be easily disabled to mitigate the risk of these change. When too much of these configurations are introduced it is hard to overview what should be the behavior of the system and testing becomes more complex due to all the possible different combinations of configurations. P5 says "*It is hard to observe in what state the system is if some changes are hidden behind some configurations*".

Monitoring the system is seen as a full-time task. Because monitoring can be seen as a full-time task simultaneously using the development environment and log analysis tools does not pose any challenge (P2). P2 says "*If a release is being deployed it is a full-time job to monitor it and you will spend all day checking if everything goes right and if something goes wrong you need to make sure it gets fixed as soon as possible, this costs a lot of focus*".

Monitoring and searching for logs can be hard because of too much noise. Some participants experience that it can be hard to find the specific log message they are looking for because there can be a lot of noise when searching using log data (P1, P4). P1 describes "*You need a smart part of the message to find the log message that you need and to not get too many noise in your search results*".

Not everyone knows where to find the log analysis tools and how to use them. In addition, there are a lot of developers that are not aware of some tools or do not know how to access and use the tools according to the participants (P1, P5, P6).

You cannot search effectively through encrypted or partially encrypted logs. Some participants mentioned that it can be challenging to search through encrypted logs (P1, P6). P6 mentions "*We log a lot of data but a lot of this data is encrypted or parts of them are encrypted which you can decrypt for pinpointing some issues or getting some statistics, but because the data is encrypted in the log analysis tools it is difficult to search through them*".

When using the logs for debugging practices and for finding the origin of a software fault certain challenges arise according to the participants.

Generic logs make it more difficult to detect which conditions caused the problem. Log messages that are generic are not useful especially when these errors happen all the time (P1, P3, P5, P6). When a generic log message is logged, it can be challenging to find the origin of the log message and it can be difficult to reproduce the behavior due to the lack of contextual information. P1 reports that "*Some exceptions are very generic, happen everywhere in the code base, and it's not easy to find what triggered it*".

Duplicate log statements make it more difficult to detect where a log originated from. Duplicate log statements in the same class make it significantly more difficult to find the origin of a software fault (P3, P5). P5 describes "*If there are two identical generic log messages you should make them distinct and especially do this at the beginning of the message so it becomes easier to recognize that there are two different log messages generated from different sources*". P3 enforces "*Sometimes it is not that easy to find the log statement where a log message originates from because the message pattern is used multiple times and you do not know which method triggered it*".

Software faults can originate from dependencies of the system. When searching for the origin of a software fault it not necessarily originates from your own code but it can also occur outside the code base such as an external library which you depend on (P6). P6 described "*Some error messages are not originating from the code base but originate from external libraries. When this happens you cannot find the origin of a software fault easily and it can be challenging to fix the software fault*".

Logs are not perfectly temporally ordered when working with distributed and multi-threaded systems. Log messages are not perfectly temporally ordered in the applications and therefore it can be challenging to analyze what the order of the events is (P3). P3 says "*Web applications are inherently multi-threaded and therefore debugging using tracing is not feasible because you do now know for sure what the order of the events is, in addition, it is hard to determine which log message to use as a starting point for the debugging*".

First observing the log analysis tools and then debugging the problem does not cause any focus loss. Additionally, participants indicate that they first observe the logs through log analysis tools and then look at the code to check or fix the problem and therefore they do not experience simultaneously using the development environment and log analysis tools as a challenge (P4, P6). P6 describes "*I'm mostly using the log analysis tools for production issues, so the moment I find the issue I start focussing on fixing the issue in the IDE*". P4 enforces "*Of course I can not do these tasks both at the same time but I don't feel like that I'm losing focus. I first look at the logs and try to understand what is happening, then I find where it is logged and fix the problem using the IDE*".

Too much logging of stack traces takes up too much space. Stack traces in large volume have a high storage footprint and therefore they should not be logged when this is not necessary (P3). P3 says "*The stack trace is very useful but storing too much of them has a negative impact on the performance of log analysis tools*".

3.3.5 What improvements do developers identify which can improve these practices and solve these challenges? (RQ$_5$)

The participants indicated that they would use and visualize the reported logging data in multiple formats to increase the overview of the system, to tackle the experienced challenges, and to improve current practices. In advance the participants were notified about the aim of the thesis.

The distributions of logs occurring can be linked back to the source code. According to the participants the distribution of logs for a certain class should be visualized in the source code or in a separate table (P2, P3, P4, P5). P5 says "*For a certain line with a log statement there could be indicated how many times that log message has happened on the production system*". P4 adds "*It would be helpful if I open my IDE and I see there were this many errors in this part of the code and I can easily detect which part of the system is misbehaving*".

The periodic distributions could be added to this link to see how the distributions evolved over time. In addition to the distribution of a log message, it could also be useful to observe the periodic distributions of the logs to see trends and to make more accurate decisions based on the latest data (P1, P4). P1 describes "*It would be useful if within the distributions of log statements there could be played with since-when constructs*".

It would be useful to get a sample log message for a selected log statement. By being able to get a sample log message for a certain log statement in the code you can more easily find out what happened (P4). P4 reports "*It would be nice when I click on a log statement, it shows a few recent samples of log messages because especially in the case of an error you can see more specifically what caused the problem.*"

The IDE should be able to track your changes and show the recent activity of your log statements. The IDE should be able to recognize what log statements the developer creates and edits and show a personalized overview of the activity for those recently modified statements (P1). P1 reports "*If your logging behavior would be tracked to decide what log messages would be useful for you, you get the valuable information without actively monitoring, this would be super useful*".

Additional information could be acquired by navigating to other log analysis tools from the source code. Participants report that it would be useful to navigate to other log analysis tools from the source code (P1, P5). P1 describes "*I would make it reproducible to go to the log analysis tools from the IDE when a log statement is clicked and see the related log messages in the log analysis tools*". P5 enforces "*It would make the process easier if you could navigate to log analysis tools from the IDE so you can see what is behind the log statement and can answer additional questions based on the context*".

Logs should be searchable from within the IDE so there can be navigated from and to the source code. In addition, there is stated that searching the logs from the IDE could be useful (P1, P3). P1 says "*From the terminal in the IDE be able to search the logs in the log analysis tools and be able to select a log message and jump to the corresponding log statement in the source code*".

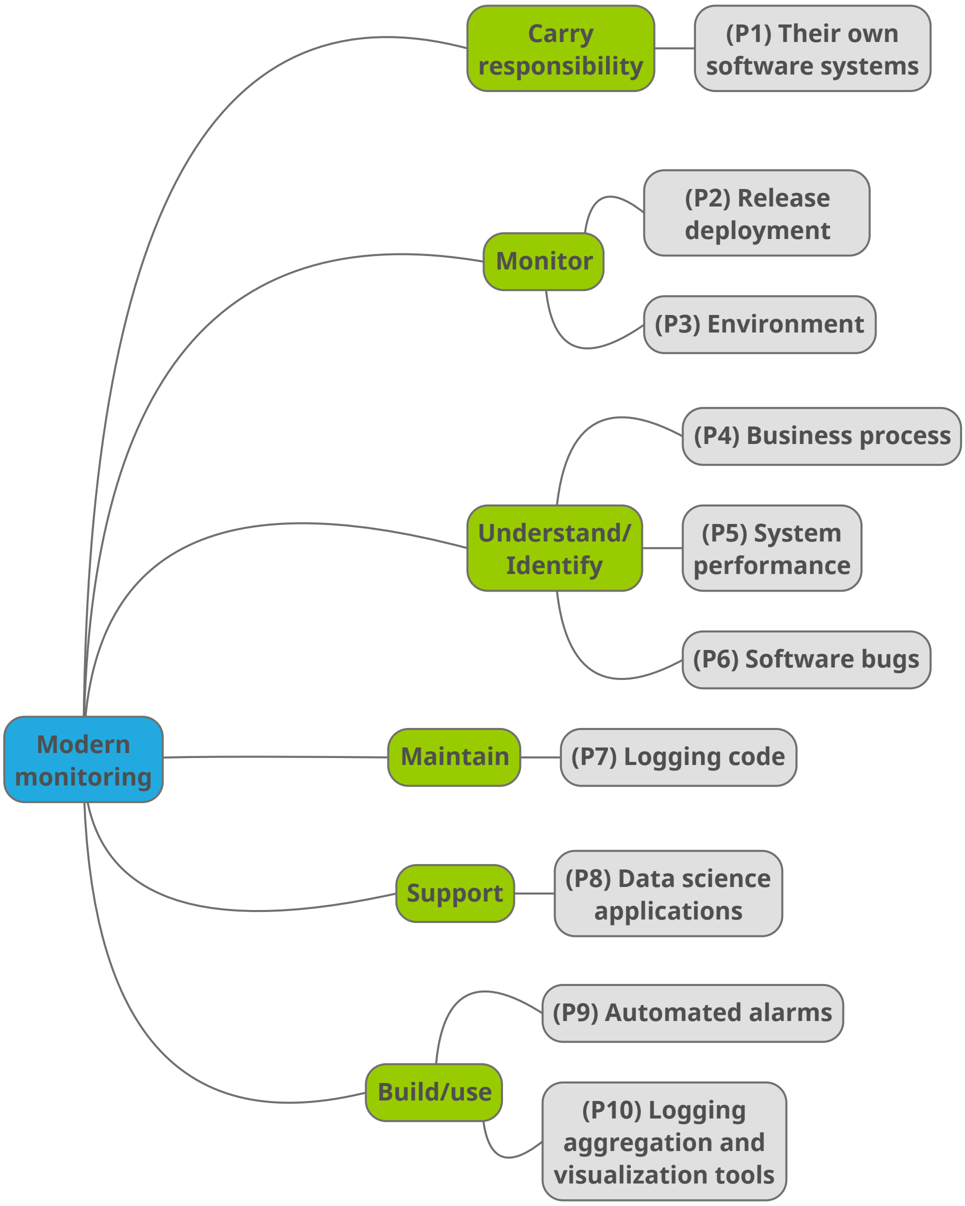


Figure 31: A model of practices in modern monitoring.

3.3.6 A model on modern monitoring practices

In figure 31, we show the resulting model aimed at explaining modern monitoring. The model contains ten practices (P1..P10) grouped in six broad themes. In the following paragraphs, we explain our model in detail. Throughout the text, we use circles to connect the model in the figure to the explaining text, *e.g.*, (P1) refers to practice number 1.

In modern monitoring and as part of DevOps culture, developers are not only responsible for testing their features before release, but to follow up and monitor how their systems behave when released to production (P1). Does it work as expected? Does it meet the performance requirements? Given that predicting how a large-scale software system will behave in production, modern monitoring takes a major role during release deployments. Even with short development cycles, large portions of new source code are released continuously to production. During release, developers intensively focus their monitoring efforts on how their newly implemented features behave in production (P2). Log data from the previous versions are often used as a baseline. Exceptions that never happened before, particularly on new source code, or exceptions that start to happen more often than in previous versions, often trigger alarms to developers who then focus on understanding why that is happening.

Interestingly, developers not only care about exceptions in their software systems, but also about how their systems impact the overall business, *e.g.*, is my system bringing the anticipated return on investment (ROI) to my company? Developers often work closely with data science teams, which also leverage the richness of the log data to extract insightful business knowledge. It is not uncommon for developers to have tasks in their backlog that aim at better supporting data science teams (P8), *e.g.*, by adding more information to existing log statements. In fact, given that developers try as much as possible to log any useful information, the amount of log statement lines in the source code is significant. Adyen, more specifically, has around 30k log statements throughout its source code base. In other words, with log statements playing an essential role in software systems, maintaining logging code (*e.g.*, improving or removing log statements) is a recurrent activity in modern monitoring (P7).

Developers make use of several tools to support their constant monitoring activities. These tools are vital to helping them deal with the large-scale nature of their systems. Besides the fact that these systems produce large amounts of log data, they are also often distributed, which require teams to make use of existing log storage, aggregation and visualization tools (P10), such as the ELK stack, or even build their own tools and automated alarms (P9). Moreover, developers also monitor their entire environments (P3), such as the health of their Linux servers, databases, and servers.

Due to the complexity of their software systems, monitoring data is also fundamental for developers to identify functional (P6), stability and performance (P5) issues. Again, monitoring data provides developers with not only unexpected and new exceptions, but also with information that helps them debug and track the problem. When it comes to performance issues, developers often measure the time it takes between log messages as an indication of a possible problem. Moreover, developers also use monitoring data as a way to trace and comprehend complex business processes (P4). In practice, no developer is able to understand every single detail of the entire business completely. A developer might learn that payment transactions always go first to the Risk Management system, and then later to the Reporting system, by reading log data.

A clear understanding of modern monitoring and its distinct practices enables us to better support them, which is what we aim for in the remainder of this thesis.

Chapter 4

Monitoring-Aware IDE’s

We identified challenges and practices that exist for logging and log analysis from the background and the performed interviews discussed in previous chapters. Using current monitoring and development tools, developers struggle with overviewing what is happening within complex and large software systems promptly. Developers can be assisted in overviewing this operation or monitoring information to overcome these challenges.

Therefore we propose a theory that IDE’s should be aware of monitoring. A monitoring-aware IDE implies that a link is constructed between the monitoring information and the source of this monitoring information. The developer can be informed about monitoring information by navigating through the source files inside of the IDE. Our theory includes that developers will have benefits from such a monitoring-aware IDE. We conjecture that the usage of this IDE will give the developer more awareness of the activity that is happening within the deployed live system. The monitoring-aware IDE provides a link between development and operations through an explicit link between both. The data presented by the monitoring-aware IDE can assist the developer in understanding the system, fixing problems in the code, and improving the code which is responsible for providing the monitoring information.

To enable this theory of monitoring-aware IDE’s to provide this set of advantages there are specific requirements that should be met for an implementation of such a theory. We envision the following set of features and requirements for our theory and these should be adhered to in the implementation of a monitoring-aware IDE:

- Monitoring information should be immediately available to the monitoring-aware IDE to make sure that data-driven decisions can be made based on the most recent data.

- There should be a connection between the monitoring information and its source. The source of monitoring information can be found based on monitoring information, and monitoring information can be detected based on its source using this connection. It should be possible to trace the monitoring information in the IDE back to its source and vice versa.

- The IDE should present monitoring information and the connection to its source. There should be functionality provided that uses the connection between monitoring information and its source to navigate to the initial data source of the monitoring information.

We pose the following set of hypotheses which can be evaluated in the methodology to measure how developers would interact with a monitoring-aware IDE and to measure the impact of a monitoring-aware IDE on their behavior:

H1 Monitoring-aware IDEs would make developers to often interact with monitoring information.

H2 Monitoring-aware IDEs would provide developers with better awareness of how their systems behave in production.

H3 Monitoring-aware IDEs would help developers to understand the business processes of their software systems.

H4 Monitoring-aware IDEs would help developers to fix problems that happen in the production environment of their software systems.

H5 Monitoring-aware IDEs would help developers to improve their monitoring code.

Chapter 5

Methodology

The effectiveness of monitoring-aware IDE's will be measured by letting a group of developers use the tool. The developer's behavior will be measured while using the monitoring-aware IDE for completing their regular daily tasks. A field study will be performed with a subset of 12 developers at Adyen across a period of four weeks.

The goal of this study is to understand how developers behave when having monitoring information inside the IDE. Monitoring information can be provided inside the IDE by means of showing log information. Firstly, we will measure the interaction between the developer and the monitoring-aware IDE. We can provide metrics to indicate how developers interact with the monitoring-aware IDE based on these measurements. Secondly, we will measure developers perspective on their behavior during the field study using a weekly survey. Based on the answers collected by this survey there can be measured how developers perceive the monitoring-aware IDE impacts their behavior. Lastly, we will use these results as a basis for the discussions during a questionnaire with the participants of the field study and see what developers perceptions are on the usefulness of a monitoring-aware IDE.

5.1 Field Study

To test these theories we provide a design for a field study where there is little control over the variables in a complex setting. We chose to perform a field study because it captures a realistic setting and can provide a high degree of detail on how the participants use and benefit from the monitoring-aware IDE. Participants will be tracked and will be surveyed while using the monitoring-aware IDE for their regular daily tasks. At the end of this evaluation, the participants will be asked to join a questionnaire so the effectiveness of the monitoring-aware IDE can be evaluated.

5.1.1 Field Study Procedure

The developers within the company will be notified before the start of the field study about the existence of the monitoring-aware IDE, its functionalities, that it can be used shortly, and that there will be an accompanying field study in which they can participate to aid the quality of the monitoring-aware IDE and the results of this thesis. We asked participants to install the monitoring-aware IDE and continue with their regular development tasks during a period of four weeks. All developers within the company were asked to participate through a channel which is used for all general development communication within the company. To participate in the field study and to use the monitoring-aware IDE there are some technical requirements for the IDE a developer is using and the project a developer is working on. Prior to the field study, the participants will have some time to try it out, get acquainted with the monitoring-aware IDE, and learn how to use it. The developers will be assisted at the start with installing and using the functionality within the monitoring-aware IDE.

After each week of usage, the participants will be asked to fill in a survey about the effect of the monitoring-aware IDE on their behavior. The participants will be requested to fill in the survey on Friday and not later than Monday to make sure that the developers still remember their actions as detailed as possible. After four weeks the behavior of the participants will not be tracked anymore and the evaluation by means of a survey will not be continued.

5.1.2 Usage Data Collection

The monitoring-aware IDE is instrumented to detect and store some interaction types between the developer and the monitoring-aware IDE. These interactions are used to understand how developers interact with the monitoring-aware IDE. These interactions will be collected during the field study. In addition to each action, we report a user identifier for the participant which performed the action and a timestamp to indicate when the action was performed. The following actions are measured by the monitoring-aware IDE:

- The monitoring-aware IDE tracks it when the participant opens a file containing source code for which monitoring data has been collected.
- When a file is opened inside the monitoring-aware IDE the amount of active patterns for that class is being tracked.
- When a file is opened inside the monitoring-aware IDE the amount of log statements which are generated for that class is being tracked.
- The monitoring-aware IDE tracks it when the participant hovers over a log counter and a tooltip is shown.
- The monitoring-aware IDE tracks it when the participant clicks on a log counter and navigates to an attached link.

Using the interactions that can be measured in the monitoring-aware IDE and the metrics extracted from that data we can indicate how a developer interacts with a monitoring-aware IDE. The measured interactions can be used as a basis for the questions that are asked of the participants during the field study.

5.1.3 Survey Design

To make it possible for the developers to evaluate their behavior during the field study we will provide a weekly survey which asks questions about their actions based on the classes for which they interacted with the plugin.

A the end of each week we generate surveys for all the developers. The survey first displays all the classes in which the participant interacted with the monitoring-aware IDE and the participant is asked to select the classes in that list in which they had an useful interaction with the monitoring-aware IDE. For those classes there is asked in what way the interactions had an effect on their behavior. We cannot detect whether participants looked at the log counters, therefore, we cannot be fully accurate in which classes changed their behavior, therefore we add an option at the end of the survey to indicate any additional scenarios in which the participant changed their behavior. For the classes which the participant interacted with the survey will show a random subset of actions which that participant performed to make it easier for the participant to recall their behavior. For each class, the participants are asked to indicate from a list which behavior changes they observed.

We created a list of behavior changes based on data extracted from the performed interviews in collaboration with developers from within the company. We are aware that this is not a comprehensive list, we give the option to provide another behavior change, where developers can say anything they want. Depending on the answers provided during this survey, additional behavior changes can be appended to this list. For other provided motivations we qualitatively analyze the answers and add them to the correct behavior change groups when developers wrote the same thing using different words. The behavior changes that we identified can be divided into three categories: observations which provide insights, code changes which are motivated by these new insights, and log improvements which can be performed because of these new insights or the lack of these insights.

Observations Based on the monitoring information shown in the monitoring-aware IDE the developer can develop new insights into the behavior of the code. These potential insights can be divided into multiple categories of insights. We identified the following list of observations in collaboration with developers at the company:

O1 Identified a bug

O2 Identified performance issue

O3 Identified security issue

O4 Identified an issue in the log code

O5 Understood the business process

O6 Understood the stability of the implementation

Code changes Based on the insights a developer acquired because of the information in the monitoring-aware IDE, a developer could make a change related to the implementation of the project. These potential implementation changes can be divided into multiple categories of code changes. We identified the following list of implementation changes in collaboration with developers at the company:

I1 Fixed a bug

I2 Improve code quality (refactoring)

I3 Improved code performance

I4 Improved code security

I5 Implemented new functionality

Log improvements Alternatively, the developer could introduce a change and a potential improvement to the log code based on the provided data. These potential log improvements can be divided into multiple categories of log improvements. We identified the following list of log improvements in collaboration with developers at the company:

L1 Improved log message

L2 Changed log severity

L3 Removed log line

L4 Added log line

Using the perspectives of the participants on their changed behavior using the weekly survey and the resulting analysis of this date there can be indicated how developers perceive how the monitoring-aware IDE impacts their behavior. This can, in addition, be used as a starting point for the discussion which is part of the questionnaire to learn how the participant of the field study perceives the usefulness of the monitoring-aware IDE.

5.1.4 Post-Questionnaire

The post-questionnaire will be performed after the field study has been completed and when the results are analyzed. All the participants in the field study will be invited to participate in this questionnaire. The aim of performing the questionnaire is to reflect on the field study and compare their workflow with their workflow prior to using the monitoring-aware IDE. Besides, we want to look at how the participants usually would acquire the same level of understanding without the monitoring-aware IDE. The post-questionnaire, therefore, reinforces the hypotheses provided in the theory chapter. The questions evaluated in the post-questionnaire are provided in Appendix C.

5.2 Participants

We evaluated the behavior of 12 participants during the field study. The participants volunteered from the group of developers at Adyen. The development experience and experience within the company for the participants can be seen in Table 51. For the participant selection we send a message to all the developers in the company through the main communication channel for developers. To make sure that developers are voluntarily participating in the field study they have to enable the tracking of their interaction themselves.

	Group	Development Experience (Years)	Adyen Experience (Years)
P1	Team A	1.5	0.5
P2	Team A	4.5	3
P3	Team B	4	1
P4	Team C	3	2
P5	Team D	5	2
P6	Team E	5	0.5
P7	Team F	8	4
P8	Team F	2	1
P9	Team G	7	1
P10	Team A	6	0.5
P11	Team F	9	0.5
P12	Team E	7	0.5

Table 51: Development experience, experience within the company, and the group the participant is part of.

Chapter 6

Implementation

To test the theory provided in the previous chapter we implemented the following monitoring-aware IDE. In this chapter, the architecture and design are discussed of the monitoring-aware IDE which aims to improve and increases the overview of the live system for developers by providing monitoring information in the IDE. The tool exists out of two systems to make sure that the system is time efficient and to make sure that no redundant computations are performed. The architecture of the tool can be divided into two component types as can be seen in figure 61. The monitoring data aggregator is responsible for generating the patterns from the code, matching the live logs to these generated patterns, and serving the log counters for the monitoring-aware plugin. The monitoring-aware plugin can then access this aggregated data through the API of the monitoring data aggregator. The second system is the monitoring-aware plugin which takes care of pulling the log counters from the monitoring data aggregator, matching these log counters to the source code, and showing these log counters linked to the source code in the IDE. The component architecture seen in figure 61 is necessary to limit the processing time needed for aggregating the data and the amount of data collected from the source code and the production log messages. If this monitoring data aggregator architecture would not be used then each monitoring-aware plugin would need to aggregate the same dataset.

The functionality for the two component types is visualized in Figure 62. The monitoring data aggregator links the source code to the production log messages by first generating patterns from the source code and matching these patterns with the log messages. The log viewers connect to this monitoring data aggregator, request the data that it needs based on what the developer is doing, and visualize the data in the IDE. The monitoring-aware plugin is only responsible for requesting and visualizing the data to limit the amount of processing that needs to be performed within the IDE and to make sure that the developer is not slowed down.

6.1 Monitoring Data Aggregator Architecture

All important logic in the system happens within the monitoring data aggregator. Log patterns are generated from the source code and these are linked to production log messages.

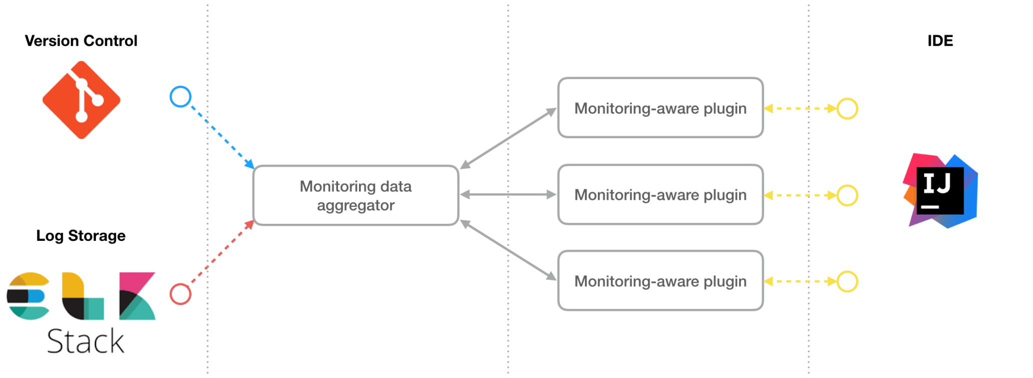


Figure 61: Architecture overview of the system which consists of a monitoring data aggregator and multiple components that request data from the monitoring data aggregator and visualize this within the IDE.

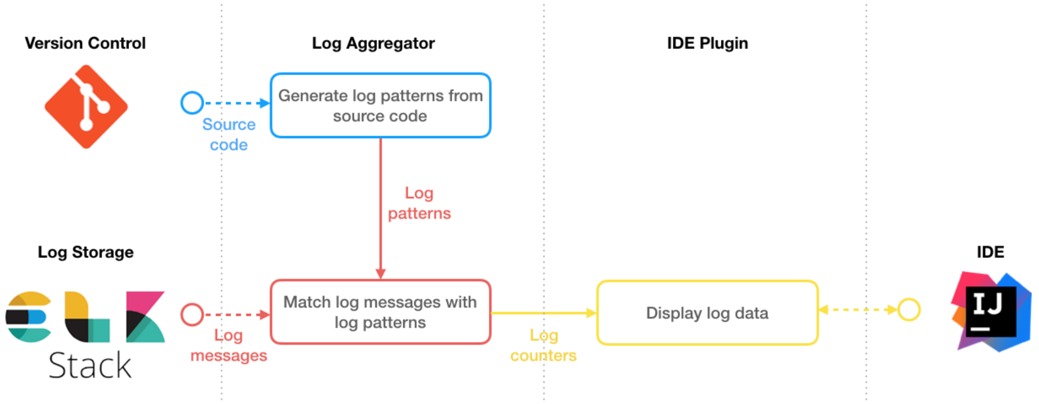


Figure 62: Architecture overview of the system which consists of three separate functionalities.

This aggregated data is stored within the monitoring data aggregator and can be accessed from the monitoring-aware plugin or other systems. Because of the scale of the data, it is important that the algorithms used within the monitoring data aggregator have a low complexity to ensure that the system is scalable.

6.1.1 Pattern Generation

The pattern generation functionality is responsible for generating patterns from the source code of a given project. Pattern generation algorithms look at the abstract syntax tree from the source code, detect log statements in this abstract syntax tree, and investigate what is the pattern of the argument of such a log statement. Multiple algorithms can be used to generate patterns from the source code depending on the accuracy and complexity that is needed [40, 28]. A simple method for pattern generation can be used which constructs a log pattern based on the static and the dynamic parts of the argument of a log statement.

1. The monitoring data aggregator pulls the source code from the version control system. The received file system is used as input for the pattern generation algorithm that is being used.

2. The pattern generation algorithm generates all patterns for all classes in the provided project.

    a) The monitoring data aggregator generates patterns for each log statement in the code based on the argument provided for each log statement. It does so by creating a regular expression based on the dynamic and static parts of the argument of the log statement. The algorithm tries to improve the dynamic parts by inferring the static parts of the string representation of the type of the variable.

    b) Similarly the monitoring data aggregator generates patterns for each throw statement in the code based on the argument provided for the created exception and its type. We generate patterns for exceptions, so that we can link an exception that is thrown in the production system to the line that actually throws it in the source code.

3. After the pattern generation has been completed, the patterns are stored within the monitoring data aggregator so it can be efficiently accessed when it is needed for pattern matching.

4. The monitoring data aggregator periodically checks if new changes are detected for the project, if so the patterns for this project are generated again. This process is repeated each time changes are made to the production system, the intermediate changes that are made to the tracked system between the different versions are not important because only log statements that exist in the production system can be logged.

6.1.2 Pattern Matching

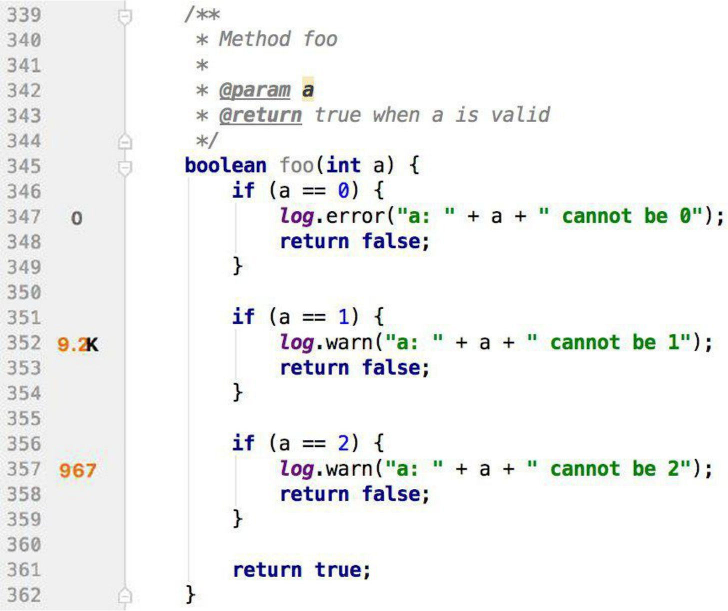
After the patterns are generated and stored, they can be used to match production messages. The complexity of matching logs to patterns depends on the number of patterns that are stored for the class and the complexity of the matching algorithm used which depends on the length of the pattern and the length of the log message. The monitoring data aggregator counts for each pattern the amount of times a log message occurred in the system which matches. This aggregated data which is stored in the monitoring data aggregator can be accessed from the monitoring-aware plugin or other systems. When accessing the monitoring data aggregator, the class for which aggregated data should be returned needs to be provided. The monitoring-aware plugin accesses this data to minimize the complexity of the monitoring-aware plugin and to minimize redundant operations.

1. The monitoring data aggregator streams the production log messages from a certain source.
2. The input for the pattern matching algorithm contains these production log messages together with the generated patterns. The monitoring data aggregator matches the log messages by first looking up all patterns which match the severity and the origin class for that message.
3. This list of patterns is filtered based on which patterns do not match. For each pattern, there is checked if there is a match between the pattern and the message within the log. A log can possibly have multiple patterns that match because there might be log statements in a class that can generate ambiguous messages.
4. Similarly the list of patterns is filtered based on which exceptions patterns do not match. When a log message is detected which contains a stack trace, the log message can be linked to the location where it was thrown in the source code. From the provided stack trace there can be found in which class the exception was thrown, all exceptions patterns for that class are retrieved. For each exception pattern, there is checked if there is a match between the pattern and the message within the stack trace of the log.
5. The monitoring data aggregator stores the amount of matches that occurred for each pattern. This data is efficiently stored so all counters for a pattern for a given class can be found efficiently.

6.2 Monitoring-Aware Plugin Architecture

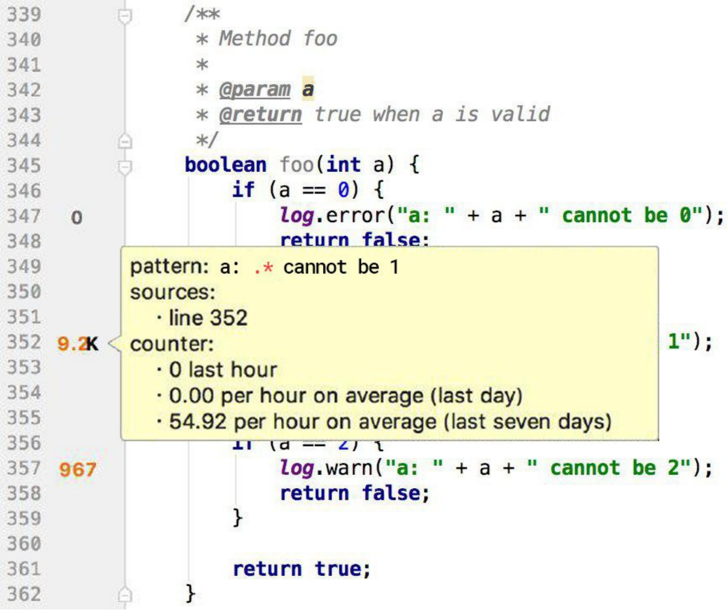
The interaction with the user of the tool happens inside the monitoring-aware plugin. The plugin can be installed in the IDE and every plugin will connect to the monitoring data aggregator. The monitoring-aware plugin introduces a monitoring-aware IDE. The monitoring-aware plugin accesses the number of times a match occurs for a pattern from the monitoring data aggregator and shows this information linked to the source code in the IDE. This amount can be called a log counter. When a developer opens a file, information about the log counters for this file is accessed from the monitoring data aggregator. These log counters are visualized with respect to the source code by adding the frequency of a log statement being called in front of the actual log statement in the code.

The developer can access additional information about these log counters by hovering over the number shown in the log counter. This additional information contains the pattern generated by the monitoring data aggregator, the periodic frequency of the log counter, and other lines in the file where an identical log pattern occurs. When a log counter is selected inside the monitoring-aware IDE, the user is navigated to the same pattern in elastic search so more information can be acquired about that particular log statement.



```
339           /**
340            * Method foo
341            *
342            * @param a
343            * @return true when a is valid
344            */
345           boolean foo(int a) {
346               if (a == 0) {
347   0               log.error("a: " + a + " cannot be 0");
348                   return false;
349               }
350
351               if (a == 1) {
352 9.2K              log.warn("a: " + a + " cannot be 1");
353                   return false;
354               }
355
356               if (a == 2) {
357 967               log.warn("a: " + a + " cannot be 2");
358                   return false;
359               }
360
361               return true;
362           }
```

Figure 63: Screenshot of the Monitoring-aware IDE implementation. The numbers in front of the log statements indicate their frequency in the production system.



```
339           /**
340            * Method foo
341            *
342            * @param a
343            * @return true when a is valid
344            */
345           boolean foo(int a) {
346               if (a == 0) {
347   0               log.error("a: " + a + " cannot be 0");
348                   return false;
349
350
351
352 9.2K                                                    1");
353
354
355
356               if (a == 2) {
357 967               log.warn("a: " + a + " cannot be 2");
358                   return false;
359               }
360
361               return true;
362           }
```

```
pattern: a: .* cannot be 1
sources:
   · line 352
counter:
   · 0 last hour
   · 0.00 per hour on average (last day)
   · 54.92 per hour on average (last seven days)
```

Figure 64: Screenshot of the Monitoring-aware IDE implementation. The tooltip indicates the patterns which was found for that log statement, the locations where this pattern occurs in this class and the periodic distributions.

Chapter 7

Results

To see how developers behave when having monitoring information inside the IDE we implemented such a monitoring-aware IDE and measured the interaction. The developers were asked after each week of usage in what way the monitoring-aware IDE affected their behavior and at the end of the field study we performed a post-questionnaire.

7.1 Interaction with the Monitoring-Aware IDE

We measured the interaction between the developer and the monitoring-aware IDE. We can indicate how developers interact with the monitoring-aware IDE based on the measurements done within the monitoring-aware IDE.

In figure 7.1 and in figure 7.1, we show how much each participant interacted with the monitoring features of our monitoring-aware IDE.

In the four weeks, developers opened 1,249 files that contained monitoring information, which represents 14% of all the 8,958 files opened throughout the four weeks. Inside these files, the IDE displayed data about 4,465 log statements. According to our post-questionnaire, the quick summary we provide near every log statement (*i.e.*, the number of occurrences of that log statement in the last month) was perceived as useful by developers: such information enabled them to quickly observe whether there was any unexpected activity in that part of the system (P2, P12) and whether these problems were urgent (P3, P4, P5, P8, P9, P11). We observed that developers mostly focused on whether the numbers displayed were "out of expected ranges", *e.g.*, near zero or very high numbers. P2 states: "*What matters to me is mostly if the number is zero or not. If it's not zero and very high (*e.g.*, 30K), I can tend to ignore it as it sounds like an 'acceptable' warning. If it's a low number higher than 0 (*e.g.*, 40) I would immediately like to check what's going on. In this case, the actual number was not really important, I was just checking whether the count was higher than 0*".

In 109 occasions, developers asked for more detailed monitoring information (*i.e.*, the periodic distribution of times that log statement appeared in the log data), either directly in the monitoring-aware IDE itself (67 times) or visiting the monitoring tool using the link we provide (42 times). According to the post-questionnaire, developers still had to visit the actual monitoring tool to get extra information about the problem they were investigating, *e.g.*, the stack trace of the problem (P4, P11), the values of certain variables (P3, P5, P12), and to get the log messages that happened before the error under investigation (P9).

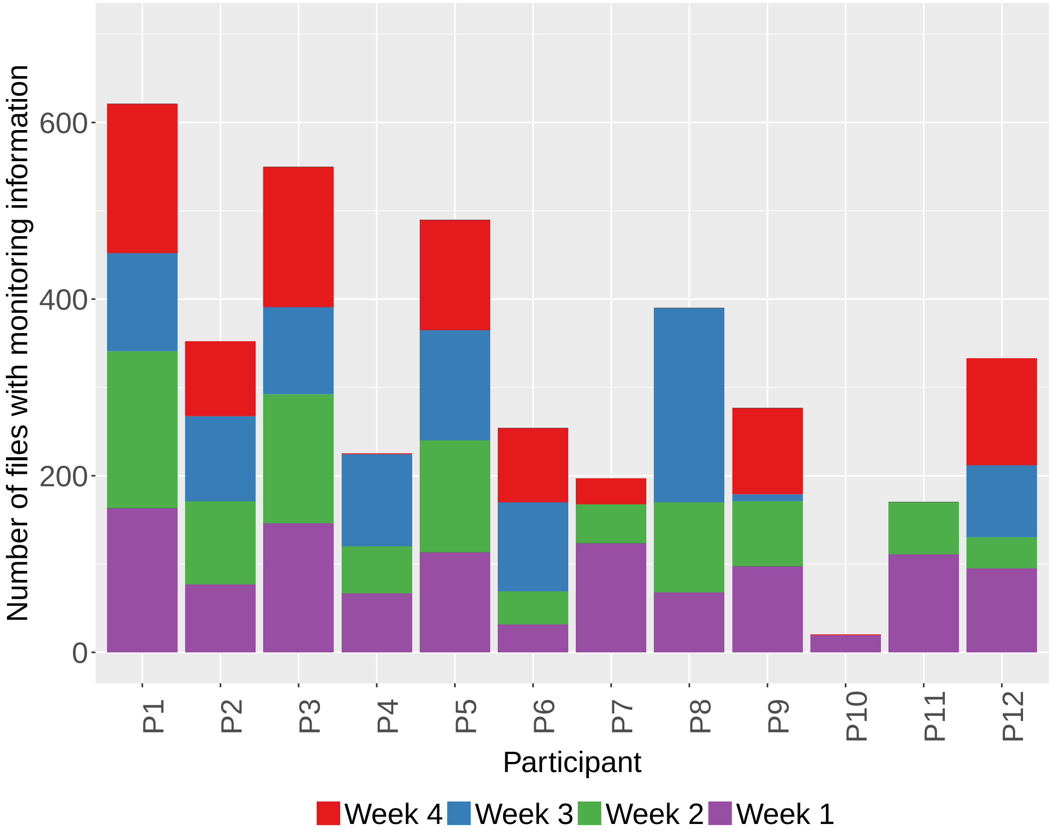


Figure 71: The number of times our IDE displayed a file that contained monitoring information (total=3,879).

Interestingly, we observed during the interviews that, at Adyen, developers have ownership of the features they build. Specific teams are responsible for their features, including their monitoring. This behavior can also be observed in our data. We observed that monitoring the same class over time is a recurrent task. 50.46% of all interactions are part of a series of interactions in the same class in different weeks. In the post-questionnaire, when presented with these numbers, developers affirmed that recurrent monitoring is common due to the size of their systems, and to the size of the features they commonly build (P3, P5, P12), and that due to weekly deployments, they often go back to see whether their features are still working. P12 states: "*I myself go back to things I worked on from time to time as well.*"

Participants significantly interacted with the monitoring-aware IDE and both measurable features. Developers are likely to return to monitor the same class in later weeks to

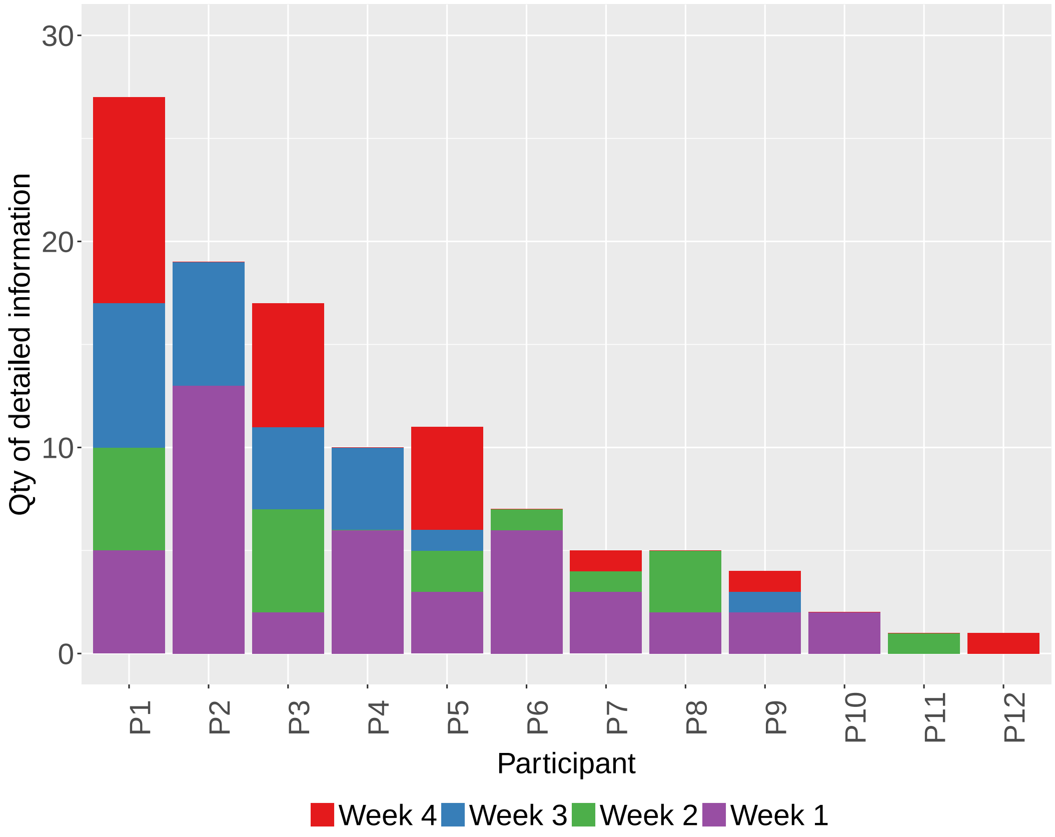


Figure 72: The number of times a participant asked for more detailed information in a log statement or exception (total=109).

observe how the data for these log statements has changed.

7.2 Impact of the Monitoring-Aware IDE

This section aims to answer how the behavior of the developer is impacted by the monitoring-aware IDE. We generated list of classes in which the participant interacted with the monitoring-aware IDE. The developers perspective on their behavior during the field study was measured using a weekly survey. Based on the answers collected by this survey there can be measured how developers perceive the monitoring-aware IDE and how it impacts their behavior. Together, participants completed 29 weekly surveys (out of 48 possible). Developers informed us that, in 45 opportunities, the usage of our monitoring-aware IDE had a positive impact on their software systems, which we show in figure 7.2 and figure 7.2.

We observe that developers took meaningful actions after observing monitoring data. 9 out of the 12 participants (P1-P9) had a positive consequence of using a monitoring-aware

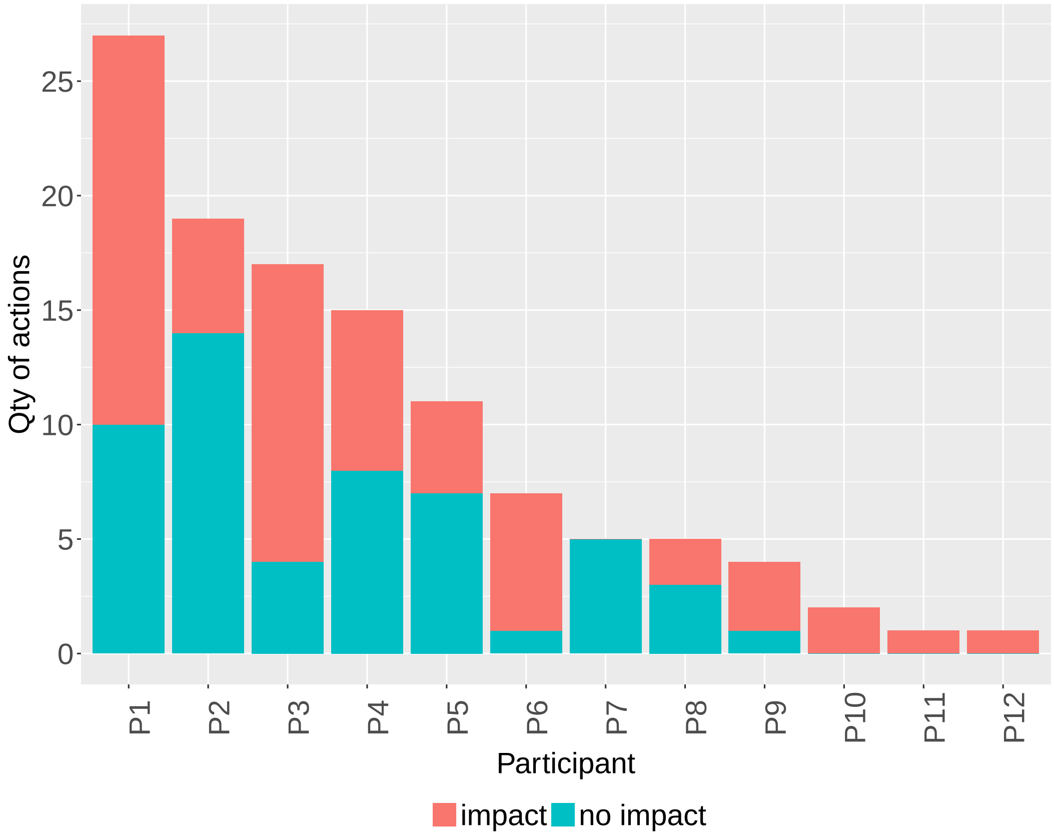


Figure 73: Whether our monitoring-aware IDE impacted our developers (N=45, 12 participants).

IDE. We notice that the three participants who did not observe any positive effects (P10-P12) were the ones with the least number of interactions with our tool (Figure 7.1). There is a strong correlation between asking for detailed information and being positively impacted by our tool (Pearson correlation = 0.85, p-value=0.001).

Understanding the business process through monitoring was the most common consequence of using our monitoring-aware IDE (15 times out of 45, or 33%). Moreover, understanding performance issues (5 times, 11%), as well as the stability of implementation (9 times, 20%) were also common consequences of using our monitoring-aware IDE. Developers accredited a few identification and bug fixing activities in their software systems (3 and 2 times, respectively) to our monitoring-aware IDE. Although identifying and fixing bugs did not happen as often as the understanding, we state that Adyenalready has a mature software and, thus, we would not expect developers to identify and find several bugs that often, and any bug found has significant positive impact on their software. Finally, monitoring information also helps developers in maintaining their logging code (8 times, 16%). We observed developers adding new log lines (1 times, 2%), improving an existing log mes-

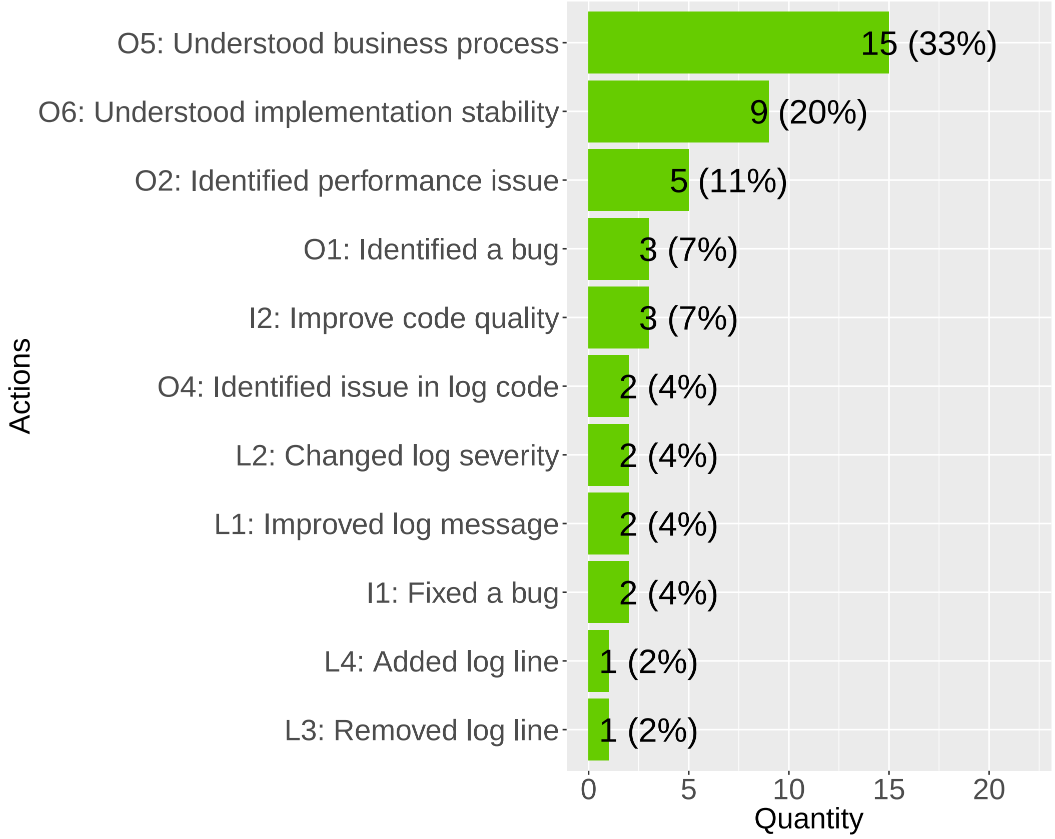


Figure 74: How our monitoring-aware IDE impacted our developers (N=45, 12 participants).

sage (2 times, 4%), changing the severity of a log statement (2 times, 4%), or removing an existing log statement (1 time, 2%).

Interestingly, developers did not identify any security issues using our tool. When asked about it in the post-questionnaire, developers affirmed that they would not expect to find security issues with our tool given that their logs do not focus on it (P2, P4, P5, P11). P11, specifically, said that they would need to write log statements whose sole purpose is to monitor security, which then our tool would help monitor. Finally, P2, P3, P8, and P11 pointed out the fact that Adyenhas already a secure software and security issues do not often happen (and thus the likelihood of such an issue to happen during our field study was too small). We indeed conjecture that providing developers with traditional monitoring data only is not enough for them to observe security issues. A follow-up step for this work would be to study how security-related aspects would fit in a monitoring-aware IDE.

The participants indicated that most interactions impacted their behavior. The monitoring-aware IDE enabled the participants to understand the software system, to fix problems in the software system, and to improving the log code. The impact of these interactions in the monitoring-aware IDE are highly related to understanding the software system in production.

7.3 Usefulness Perception of the Monitoring-Aware IDE

During our first field study, we observed that developers spent a significant amount of time going back and forth between their monitoring tools and their IDEs. Our overall perception was that this context switching was not productive.

These observations were corroborated in our post-questionnaire. Developers affirmed that our monitoring-aware IDE did not replace their monitoring systems, but it helped them in saving time and reducing cognitive load when compared to the way they use to perform the same monitoring tasks before our tool. Several of our participants affirmed to spend less time querying their monitoring systems (P1, P2, P3, P4, P8). P2 says: "*I still use Kibana as much as I used it before. I do like however the easy navigation from a log statement in Intellij to Kibana.*", and P3 says: "*Kibana Requires a lot of manual work (writing query) for the other tools to actually notice errors that happen in a class that you work in.*" Automatically performing the link between the log message and the actual log statement as well as not having to query the monitoring tool also helps developers in following the flow of the source code more productively. P8 states: "*Instead of having to follow the flow of the code by changing parameters on a Kibana search, the faster interaction with the plugin makes navigation smoother.*" P5 says: "*Now I don't have to select a constant string from the log statement and hope to find it in the logs. Also I know earlier whether it is worth investigating further or not.*". Finally, P8 also adds that the tool reduces his amount of context switching and that the tool also saves time when communicating about an error: "*If someone tells me about an error, I can find it in code easily [and] then find all related log instances*".

In the post-questionnaire, developers also perceived other benefits in monitoring-aware IDEs that go beyond saving time. The instant (near) real-time feedback and the timely observations that our IDE offer enables developers to quickly identify possible bugs or bottlenecks (P3, P4, P5, P9, P11, P12). As we stated before, developers pay much attention to the frequency of a log statement, and whether this number seems to be "out of place" (*e.g.*, near 0, or very large). P5 says "*An error or warning on its own doesn't indicate a bug, but the number of time it gets triggered might. That's why the tool is useful, to identify them.*" P5 also provided us with a concrete example of how he was able to track a performance bug: "*It helped me find a situation where data had to be loaded explicitly, while it should have been preloaded.*"

Developers also see a positive impact in having monitoring data and logging code together (P2, P3, P5, P8). P2 affirmed: "*So far it stimulated me to improve logging where the amount of warnings was very high (*e.g.*, 100K).*" Other participants mentioned that thanks to to the real-time feedback, they are better able to decide which log level to use in a log statement (P3), help in identifying situations where better logging is required (P5), and remove less useful log statements (P8). P5 says "*I wouldn't say the tool helps me to improve log messages directly, but it helps me find interesting situations, which may require better logging. In that case it indirectly helps I suppose.*", and P8 says "*You can see which log* *messages are useless, and also given the quicker feedback loop on seeing the detailed logs on Kibana you are more inclined to make improvements.*"

Indeed, developers noticed that a monitoring-aware IDE does not replace any other tool, but rather complements them. P11, for example, says that he still uses the ELK stack to follow the flow of a transaction (as the monitoring tool allows him to see all messages related to a specific transaction ID). P11 also uses another internal tool to help in identifying performance issues. P2 says: "*I do not think the tool covers the need of monitoring via other means and it can't replace them. It gives extra insights only into the code/class that we are working on. Monitoring via automated patch monitoring or Kibana gives better functionalities on aggregating log data from multiple places.*"

Finally, the developers provided us with some insightful suggestions on the next steps of our tool. Most of their suggestions are related to either adding more information (P2, P3, P4, P5, P9) or adding filters (P11). Showing monitoring data at package-level and not only at class-level as is now (P4), personally configuring the date and time periods to show (P9), summarize the status of the log statements developers have written themselves (P11), and adding charts that would show the complete periodic distribution of that log statement (P3) are among the suggestions.

Chapter 8

Discussion

Our field study shows that monitoring-aware IDEs help developers understand complex business processes, identify and fix bugs, and maintain and improve their logging code. In the following section, we will discuss the challenges of building monitoring-aware IDEs, and threats to validity.

8.1 Threats to Validity

While performing our study we observed several threats to the validity. In this section we categorize the threats to the validity in three different categories and discuss how to mitigate these threats to the validity.

Construct Validity

Threats to construct validity concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed. There cannot be perfectly measured during the field study how people interact with the log counter when they do not click or hover it. We cannot measure when a participant looks at and observes monitoring information in the monitoring-aware IDE. The weekly surveys and post-questionnaire captured the impact of a monitoring-aware IDE in a modern monitoring setting through the perceptions of the developers. Because developers might mostly remember the remarkable actions they take, we conjecture that our data, therefore, represents a lower bound of the actual benefits of a monitoring-aware IDE. We did not have an explicitly controlled baseline in our field study, as that would be impractical at Adyen's realistic settings. Instead, we explicitly collected data about the developers' perceptions on using and not using a monitoring-aware IDE in the final questionnaire, which enriched our final analysis. As future work, we plan to replicate our study in a more controlled setting, now that we have better insight into independent and dependent variables. We use our prototype as a proxy to understand the impact of a monitoring-aware IDE in modern monitoring. Due to technical limitations of the logging structure we cannot use info messages and are restricted to use messages with the warning and error severity log level. This significantly reduces the scale at which the monitoring-aware IDE can be used as most log statements have an info severity.

As we present in the field study, our prototype contains features that we derived from the model, together with the Adyendevelopers. Nevertheless, we do not claim that our prototype fully represents or includes all possible features of an idealistic monitoring-aware IDE. We invite HCI researchers to step in and explore what the design space and requirements of such a tool would be.

Internal Validity

Threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated. As we proposed a field study instead of a controlled experiment, we had no control over the tasks developers were performing. This is clear when observing the behavior of P10 (used the tool for a week, rather than four weeks) and P11 (used the tool for two weeks). These developers were mostly working on features not yet deployed, and thus, no monitoring information was available. In addition, some developers might have been spending their four weeks on bug fixing, whereas others focused their time on new features. While this variance may influence, we argue the goal of our field study was not to measure the effects in a precise way, but instead to search for evidence that monitoring-aware IDEs would improve modern monitoring practices.

External Validity

The threats to external validity concern the generalization of results. This entire research was conducted at Adyen. We thus see the generalizability of this research in a similar way that social scientists see transferability in their work [25]. However the developers that participated in the interviews and the field study also have development experience out of Adyen. Further, we try to mitigate this fact by diversifying our field study by conducting it with developers from 7 different teams that represent various kinds of development contexts. Adyenis a large-scale payment company that deals with large amounts of sensitive data, produces large amounts of log data, and sees monitoring as a fundamental activity. Adyen is putting a lot of effort in logging and they deal with a very large amount of log statements and log messages and therefore the results can be specific to Adyen. Although we can not claim any generalization, given the size, scale, and importance of the software built by Adyen, we believe this idea is worthy of further investigation by large and small companies.

Chapter 9

Conclusion

Monitoring is a new and fundamental role that developers have to play in large-scale software systems. However, modern monitoring does not come without its challenges. In this thesis, we studied the current practices of modern monitoring and propose monitoring-aware IDEs as a way to better support developers during their monitoring activities. We provide this definition of modern monitoring as well as a derived model that explains its current practices, after interviewing developers at the company. Our modern monitoring model shows the tight connection between monitoring activities and software development.

The resulting theory outlines how IDEs that are monitoring-aware which can support developers in better performing modern monitoring by incorporating monitoring data into the workflow of working with source code. Our theory is that for developers to be better equipped to deal with modern monitoring practices, IDEs and monitoring should be connected. A monitoring-aware IDE provides developers with an integrated view of both the implementation of their software systems and monitoring information. We propose that IDEs should be responsible for bridging the gap between the implementation of the system and its operation.

We performed a four week observational field study that brings evidence on the usefulness of monitoring-aware IDEs to modern monitoring. We observe that developers took meaningful actions after observing monitoring data. We firmly believe that monitoring-aware IDEs will play an essential role in improving how developers perform modern monitoring. For better system operations, we should bridge the gap that exists between development and operation. Monitoring-aware IDEs are the first step in that direction.

Bibliography

[1] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.

[2] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. "*Leveraging existing instrumentation to automatically infer invariant-constrained models*". In: *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. ACM. 2011, pp. 267–277.

[3] Boyuan Chen and Zhen Ming Jack Jiang. "*Characterizing and detecting anti-patterns in the logging code*". In: *Proceedings of the 39th International Conference on Software Engineering*. IEEE Press. 2017, pp. 71–81.

[4] Lianping Chen. "*Towards architecting for continuous delivery*". In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE. 2015, pp. 131–134.

[5] Marcello Cinque, Domenico Cotroneo, Raffaele Della Corte, and Antonio Pecchia. "*Assessing direct monitoring techniques to analyze failures of critical industrial systems*". In: *Software Reliability Engineering (ISSRE), 2014 IEEE 25th International Symposium on*. IEEE. 2014, pp. 212–222.

[6] Marcello Cinque, Domenico Cotroneo, Roberto Natella, and Antonio Pecchia. "*Assessing and improving the effectiveness of logs for the analysis of software faults*". In: *Dependable Systems and Networks (DSN), 2010 IEEE/IFIP International Conference on*. IEEE. 2010, pp. 457–466.

[7] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. "*Event logs for the analysis of software failures: A rule-based approach*". In: *IEEE Transactions on Software Engineering* 39.6 (2013), pp. 806–821.

[8] Jürgen Cito. "*Developer targeted analytics: supporting software development decisions with runtime information*". In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ACM. 2016, pp. 892–895.

[9] Jürgen Cito, Philipp Leitner, Harald C Gall, Aryan Dadashi, Anne Keller, and Andreas Roth. "*Runtime metric meets developer: building better cloud applications using feedback*". In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* ACM. 2015, pp. 14–27.

[10] Jürgen Cito, Genc Mazlami, and Philipp Leitner. "*Temperf: Temporal correlation between performance metrics and source code*". In: *Proceedings of the 2Nd International Workshop on Quality-Aware DevOps*. ACM. 2016, pp. 46–47.

[11] Jürgen Cito, Fábio Oliveira, Philipp Leitner, Priya Nagpurkar, and Harald C Gall. "*Context-based analytics: establishing explicit links between runtime traces and source code*". In: *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*. IEEE Press. 2017, pp. 193–202.

[12] Jürgen Cito, Dritan Suljoti, Philipp Leitner, and Schahram Dustdar. "*Identifying root causes of web performance degradation using changepoint analysis*". In: *International Conference on Web Engineering*. Springer. 2014, pp. 181–199.

[13] Domenico Cotroneo, Andrea Paudice, and Antonio Pecchia. "*Automated root cause identification of security alerts: Evaluation in a SaaS Cloud*". In: *Future Generation Computer Systems* 56 (2016), pp. 375–387.

[14] Peter Evers. "*Finding Relevant Errors in Massive Payment Log Data*". Master's thesis. Delft University of Technology, 2017.

[15] Qiang Fu, Jieming Zhu, Wenlu Hu, Jian-Guang Lou, Rui Ding, Qingwei Lin, Dongmei Zhang, and Tao Xie. "*Where do developers log? an empirical study on logging practices in industry*". In: *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM. 2014, pp. 24–33.

[16] Georges Bou Ghantous and Asif Gill. "*DevOps: Concepts, Practices, Tools, Benefits and Challenges*". In: *Proceedings of the 20th Pacific Asia Conference on Information Systems* (2017).

[17] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. "*An evaluation study on log parsing and its use in log mining*". In: *Dependable Systems and Networks (DSN), 2016 46th Annual IEEE/IFIP International Conference on*. IEEE. 2016, pp. 654–661.

[18] Michael Hilton, Nicholas Nelson, Danny Dig, Timothy Tunnell, Darko Marinov, et al. *Continuous Integration (CI) Needs and Wishes for Developers of Proprietary Code*. Tech. rep. Corvallis, OR: Oregon State University, Dept. of Computer Science, 2016.

[19] Michael Hilton, Nicholas Nelson, Timothy Tunnell, Darko Marinov, and Danny Dig. "*Trade-offs in continuous integration: assurance, security, and flexibility*". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM. 2017, pp. 197–207.

[20] Michael Hilton, Timothy Tunnell, Kai Huang, Darko Marinov, and Danny Dig. "*Usage, costs, and benefits of continuous integration in open-source projects*". In: *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE. 2016, pp. 426–437.

[21] Siw Elisabeth Hove and Bente Anda. "*Experiences from conducting semi-structured interviews in empirical software engineering research*". In: *Software metrics, 2005. 11th ieee international symposium*. IEEE. 2005, 10–pp.

[22] Jonas Kunz, Christoph Heger, and Robert Heinrich. "*A generic platform for transforming monitoring data into performance models*". In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion*. ACM. 2017, pp. 151–156.

[23] Philipp Leitner, Jürgen Cito, and Emanuel Stöckli. "*Modelling and managing deployment costs of microservice-based cloud applications*". In: *Proceedings of the 9th International Conference on Utility and Cloud Computing*. ACM. 2016, pp. 165–174.

[24] 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*. ACM. 2016, pp. 102–111.

[25] Yvonna S Lincoln and Egon G Guba. *Naturalistic inquiry*. Vol. 75. Sage, 1985.

[26] Marco Manglaviti, Eduardo Coronado-Montoya, Keheliya Gallaba, and Shane McIntosh. "*An empirical study of the personnel overhead of continuous integration*". In: *Proceedings of the 14th International Conference on Mining Software Repositories*. IEEE Press. 2017, pp. 471–474.

[27] Shane McIntosh. "*Studying the Software Development Overhead of Build Systems*". PhD thesis. 2015.

[28] *Mining Free Text Console Logs*. "`https://code.google.com/archive/p/logm`". (Accessed November 27, 2017).

[29] Shubhangi Nagpal and Anam Shadab. "*Literature Review: Promises and Challenges of DevOps*". In: (2014).

[30] Adam Oliner, Archana Ganapathi, and Wei Xu. "*Advances and challenges in log analysis*". In: *Communications of the ACM* 55.2 (2012), pp. 55–61.

[31] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. "*Industry practices and event logging: Assessment of a critical software development process*". In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*. Vol. 2. IEEE. 2015, pp. 169–178.

[32] Antonio Pecchia and Stefano Russo. "*Detection of software failures through event logs: An experimental study*". In: *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*. IEEE. 2012, pp. 31–40.

[33] Gerald Schermann, Jürgen Cito, Philipp Leitner, Uwe Zdun, and Harald Gall. *An empirical study on principles and practices of continuous delivery and deployment.* Tech. rep. PeerJ Preprints, 2016.

[34] Weiyi Shang. "*Bridging the divide between software developers and operators using logs*". In: *Software Engineering (ICSE), 2012 34th International Conference on.* IEEE. 2012, pp. 1583–1586.

[35] Reetinder Sidhu and Viktor K Prasanna. "*Fast regular expression matching using FPGAs*". In: *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on.* IEEE. 2001, pp. 227–238.

[36] Jens Smeds, Kristian Nybom, and Ivan Porres. "*DevOps: a definition and perceived adoption impediments*". In: *International Conference on Agile Software Development.* Springer. 2015, pp. 166–177.

[37] Randy Smith, Cristian Estan, and Somesh Jha. "*XFA: Faster signature matching with extended automata*". In: *Security and Privacy, 2008. SP 2008. IEEE Symposium on.* IEEE. 2008, pp. 187–201.

[38] Jan Waller, Nils C Ehmke, and Wilhelm Hasselbring. "*Including performance benchmarks into continuous integration to enable DevOps*". In: *ACM SIGSOFT Software Engineering Notes* 40.2 (2015), pp. 1–4.

[39] Rick Wieman, Maurício Finavaro Aniche, Willem Lobbezoo, Sicco Verwer, and Arie van Deursen. "*An Experience Report on Applying Passive Learning in a Large-Scale Payment Company*". In: *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE. 2017, pp. 564–573.

[40] 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.* ACM. 2009, pp. 117–132.

[41] Ding Yuan, Soyeon Park, Peng Huang, Yang Liu, Michael Mihn-Jong Lee, Xiaoming Tang, Yuanyuan Zhou, and Stefan Savage. "*Be Conservative: Enhancing Failure Diagnosis with Proactive Logging.*" In: *OSDI.* Vol. 12. 2012, pp. 293–306.

[42] Ding Yuan, Jing Zheng, Soyeon Park, Yuanyuan Zhou, and Stefan Savage. "*Improving software diagnosability via log enhancement*". In: *ACM Transactions on Computer Systems (TOCS)* 30.1 (2012), p. 4.

[43] Jieming Zhu, Pinjia He, Qiang Fu, Hongyu Zhang, Michael R Lyu, and Dongmei Zhang. "*Learning to log: Helping developers make informed logging decisions*". In: *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on.* Vol. 1. IEEE. 2015, pp. 415–425.

Appendix A

Interview Questions

This appendix lists the questions to be answered by the participants of the interview.

A.1 Participant Information

1. How many years/months of experience do you have as a professional developer?
2. For how many years/months have you been working at Adyen?
3. What role do you have within Adyen?
4. How many years/months of experience do you have with log analysis tools?

A.2 Logging Experiences

Before discussing the logging experiences questions we will give a short introduction about which log analysis tools are used within the company.

1. Can you tell me about the last time you worked with log analysis tools?
2. What are your reasons for using log analysis tools?
3. When you introduce a software fault, how do you detect the software fault, which challenges do you experience?
4. How do you find the origin of software faults and how do you use logs in the process, which challenges do you experience?
5. Do you have any reasons for not using some of the available log analysis tools?
6. How did you change your logging behavior when writing code to better accommodate log analysis, what is the motivation for these changes and what are the results?
7. Do you have a good overview of what is happening within the system after release and what tools do you use for it?

8. Do you experience challenges when using the development environment and log analysis tools simultaneously?

9. What additional value does logging data contain and in what additional scenarios can logging data be used?

A.3 Feedback Extension

Before discussing the feedback extension questions we will give a short introduction about that the aim of this thesis is to construct an IDE extension which increases the overview of the production system based on logging data and log analysis data.

1. Which logging data do you deem valuable as IDE feedback?

2. In what way do you envision that this feedback can be visualized in the IDE to increase the overview of the system?

Appendix B

Ethics Checklist

This appendix contains the ethics checklist for human research for the evaluation done in this thesis [1]. This checklist was submitted before participants were approached to take part in the research study. The ethics checklist indicated that the application has a minimal risk. [2]

B.1 Summary Research

The research focuses on developing a tool (an extension for the programming environment) which increases the log awareness for developers to acquire a better overview of what is happening within a software system after it is deployed. As part of this research the effect of the tool needs to be tested by letting developers use the tool. The tool will show data that is already available to the developer in an aggregated way and allows additional interaction with this data. The effect of the tool will be measured by questioning the participant before, during, and after them using the tool to find and fix problems in the code in a previously deployed dataset. The activity and the time it takes to complete will be measured by observing the participant. The questions will be focused on how they perceive the usability of the tool.The participant will be made aware beforehand how this is being measured. The participants will be professional software developers. The amount of participant for the evaluation will be a maximum of 30 developers.

B.2 Risk Assessment

We do not see any potential risks for the participants, it will only cost them some time and focus to complete the tasks.

[1] This research application was submitted to the ethics committee on 22/02/2018 and approved on 09/03/2018.

[2] https://www.tudelft.nl/en/about-tu-delft/strategy/strategy-documents-tu-delft/integrity-policy/human-research-ethics/

Appendix C

Questionnaire

This appendix lists the questions to be answered by the participants in the post-questionnaire.

C.1 Participant Information

1. How many years of development experience do you have?

2. How many years of Adyen experience do you have?

C.2 Usage of the Tool

1. Did you look to the log counters (the numbers at the left bar, near the line number)?

   a) Why?

   b) How important and/or useful are these numbers, in your opinion?

2. We noticed that you went to Kibana while using our tool (you clicked on the log counter, which redirects you to Kibana). Why did you go to the monitoring tool?

3. Our developers spend an average of 11 seconds in the detailed information by hovering over the log counters. This is quite a lot. Can you explain why?

4. We found that most of the interactions in the plugin occurred in the same class. For example in week 1 you interacted in a certain class and you also did so in week 2. Why do you visit the same class?

C.3 Benefits of the Tool

1. In your opinion, what are the main benefits of the plugin?

2. If you compare to the way you used to do monitoring in the past (using greps, kibana, logsearch etc), what changed?

3. To each one of the benefits we observed, answer the following questions. Why does the tool help you? Feel free to use concrete examples (.e.g, in class A, this happened) to explain yourself.

    a) Understanding the business process:

    b) Understanding performance issues:

    c) Identifying and fixing bugs:

    d) Improve log messages:

    e) Improve code quality (refactoring):

4. How did you do these same activities before having our plugin installed? Why is it different now?

    a) Understanding the business process:

    b) Understanding performance issues:

    c) Identifying and fixing bugs:

    d) Improve log messages:

    e) Improve code quality (refactoring):

5. You did not identify any security issues by means of the plugin. Why do you think the tool is not able to help the developer to find security issues?

6. Did you use the tool to implement any new functionality / implemented new functionality because of your interaction with the tool or did you use it while implementing new functionality?

7. In terms of productivity, do you feel more or less productive using the tool? In the interviews we performed before, we noticed that monitoring is expensive, and can make you lose focus. In what way does the tool affect these points?

C.4 Future Work

1. What didn't you like about our tool?

2. What should be the next steps of our tool?