Monitoring-aware IDEs

Winter, Jos; Aniche, Maurício; Cito, Jürgen; van Deursen, Arie

Important note
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

Monitoring-Aware IDEs

Jos Winter
jos.winter@adyen.com
Adyen N.V.
Amsterdam, The Netherlands

Jürgen Cito
jcito@mit.edu
Massachusetts Institute of Technology
Cambridge, MA, USA

Maurício Aniche
M.FinavaroAniche@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Arie van Deursen
Arie.VanDeursen@tudelft.nl
Delft University of Technology
Delft, The Netherlands

ABSTRACT

Engineering modern large-scale software requires software developers to not solely focus on writing code, but also to continuously examine monitoring data to reason about the dynamic behavior of their systems. These additional monitoring responsibilities for developers have only emerged recently, in the light of DevOps culture. Interestingly, software development activities happen mainly in the IDE, while reasoning about production monitoring happens in separate monitoring tools. We propose an approach that integrates monitoring signals into the development environment and workflow. We conjecture that an IDE with such capability improves the performance of developers as time spent continuously context switching from development to monitoring would be eliminated. This paper takes a first step towards understanding the benefits of a possible Monitoring-Aware IDE. We implemented a prototype of a Monitoring-Aware IDE, connected to the monitoring systems of Adyen, a large-scale payment company that performs intense monitoring in their software systems. Given our results, we firmly believe that Monitoring-Aware IDEs can play an essential role in improving how developers perform monitoring.

CCS CONCEPTS

• **Software and its engineering → Integrated and visual development environments**;

KEYWORDS

software engineering, devops, systems monitoring, runtime monitoring, Integrated Development Environment, IDE.

1 INTRODUCTION

Monitoring provides information about the runtime behavior of software in the form of logs and has been used to understand large-scale systems in production. The analysis of logs is a widespread practice that has been studied in many different contexts. By leveraging log data, researchers were able to help development teams with process mining [15, 29, 53], anomaly detection [9, 24, 28, 60, 61], passive learning [57], fault localization [58, 65], invariant inference [10], performance diagnosis [33, 40, 49, 50, 64], online trace checking [5], and behavioural analysis [4, 43, 62].

However, understanding runtime behavior of deployed software is an activity that has been classically associated with operations engineers. In recent years, practices and culture of development and operations have evolved to unify their responsibilities (often referred to as DevOps). Teams no longer solely focus on either development or operations; rather, these responsibilities are more and more intertwined and unified [6, 21, 46]. Monitoring is one fundamental activity in this congregation that enables a real unification of both sides. Developers see the analysis of monitoring information as part of their primary responsibilities, and perform it seamlessly with their development tasks.

Interestingly, monitoring mainly happens in monitoring tools (*e.g.*, Kibana), whereas software development mainly happens in an Integrated Development Environment (IDE). The current situation leads to increased context-switching [18] and split attention effects that increase cognitive load [13]. If developers have to leave the IDE to do some other development-related task, then, one might say that the integrated development environment has failed.

Monitoring-Aware IDEs. We propose to integrate operational aspects into the workflow and context of software development tasks by developing the concept of Monitoring-Aware IDEs. If IDEs were to provide seamless support for monitoring activities, we hypothesize that developers would better perform development tasks, such as understanding the reason of a bug, or how a new deployed version behaves in production.

To validate the proposal, we implemented a prototype for a Monitoring-Aware IDE and integrated it into the workflow of 12 developers from 7 different teams, and evaluated it in a one-month field experiment at Adyen, a large-scale payment company which produces around 40 billion lines of log data per month. Adyen follows DevOps practices and performs intense monitoring in their software systems.

Our results indicate that Monitoring-Aware IDEs can provide essential benefits in modern large-scale software development. Developers made repeated use of the monitoring features to perform various development activities they would have not performed without our approach. Moreover, the provided information supports their development tasks in different ways, such as to better understand how their software works, how stable and performant their implementation is, and even to identify and fix bugs. Finally, their overall perception is that, while a Monitoring-Aware IDE does not replace their existing monitoring systems entirely, it helps them in reducing cognitive load and saving time by avoiding constant context switches between monitoring tools and their IDE.

The main contributions of this paper are:

- A proposal outlining how Monitoring-Aware IDEs can support developers in better performing monitoring and DevOps by incorporating monitoring data into the workflow of working with source code (Section 3)
- A 4-week field experiment that brings evidence on the usefulness of Monitoring-Aware IDEs to monitoring and DevOps teams (Sections 5 and 6).

2 BACKGROUND

In this section, we describe existing related work on the field. More specifically, we dive into the DevOps movement, log analysis and monitoring techniques as well as enhancements researchers have been proposing to IDEs. Next, we present Adyen, our industry partner (and our case study), and how they have been applying monitoring and DevOps within their development teams. We also explain why Adyen serves as a perfect case for this study.

2.1 Related Work

The DevOps movement. Different people define DevOps in different yet similar ways. Hüttermann [32] defines DevOps as "practices that streamline the software delivery process, emphasizing the learning by streaming feedback from production to development and improving the cycle time". DeGrandis [20] affirm that "The [DevOps] revolution in the making is a shift from a focus on separate departments working independently to an organization-wide collaboration – a systems thinking approach.". Walls [55] says that DevOps is a "cultural movement combined with a number of software related practices that enable rapid development." Bass et al. [6] 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.". To Loukides [38], DevOps is about integrating the infrastructure and the development teams: "Rather than being isolated, they [infrastructure team] need to cooperate and collaborate with the developers who create the applications.".

Indeed, the movement is becoming more and more popular among practitioners. A 2016 survey with 1,060 IT professionals [45] indicates that its adoption increased from 66% to 74%, especially in the entreprise world (in comparison with 2015). However, its adoption is still not as smooth as expected. Smeds et al. [47], after a literature review and interviews with experts, affirm that an important difficulty for its adoption in industry is related to its unclear definition and the company's expected goals with the adoption.

Ghantous and Gil, also based on a literature review, affirm that the main challenges consist of constructing a tool pipeline that supports the process, and overcoming the mental barrier between development and operations teams. Yet, we also observe successful DevOps adoption in industry, such as the ones in Adyen, our industry partner, but also as reported by other researchers [39].

Monitoring tools in industry. There are 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. The critique to the common dashboard solutions in current practice is that the amount of different, seemingly unrelated, graphs is overwhelming and it is hard to come to actionable insights [16].

Logging analysis and visualization. Log data is vastly rich, and thus, several analysis techniques have been proposed. Aiming at failure detection, Reidemeister et al. [44], based on previous logs, train a decision tree to detect recurrent failures. Similarly, Fronza et al. [23] uses SVM, Lin et al. [36] use clustering algorithms, and Bose and van der Aalst [12] exploit associative rule mining to discover failure patterns in event logs. While the above techniques are good in detecting previously known failures, others focus on detecting anomalies (i.e., failures not seen before). Clustering algorithms are commonly used for such [34, 36].

Logs are also used to build models of the software system. Tools such as Synoptic [10] and DFASAT [30, 57] devise finite state machines that represent a software, based on its logs. And given that logs are often ordered in a timely manner, related work also has explored temporal invariant inference [10, 41].

Finally, given that logs are often not easy to be understood as they are, visualizations that aim to support reasoning of runtime behavior have also been proposed. Examples of such work are visual depictions to better understand performance issues [3, 11], to understand how the different components of a distributed system behave and/or relate to each [1, 42, 63], and to visualize the different nodes of a cluster by means of a city landscape metaphor [22].

Augmenting existing IDEs. Work that is conceptually closest to our approach are development environments that augment source code with runtime information. Lieber *et al.* [35] augment JavaScript code in the debug view in the browser with runtime information on call count of functions asynchronous call trees to display how functions interact. Other work focuses on augmenting method definitions in the IDE with in-situ visualizations of performance

---

[1]https://www.elastic.co/webinars/introduction-elk-stack
[2]http://grafana.org/
[3]https://www.splunk.com/
[4]https://www.loggly.com/
[5]https://www.datadoghq.com

profiling information [7, 16, 17]. Hoffswell *et al.* [31] introduce different kinds of visualizations related to runtime information in the source code to improve program understanding. Lopez and van der Hoek [37] augmented IDEs to warn developers, on a line-by-lines basis, about the volatility of the code they are working on.

Our approach is the first to integrate information and traceability links from production logs into the source code view. This enables a more general-purpose approach to reasoning about production behavior that is guided by signals put in place by developers themselves (log statements).

2.2 Monitoring and DevOps

All observations in this research are based on the teams that follow the DevOps model at Adyen, a large-scale payment company that provides services for more than 4,500 companies all around the world. Adyen had a transaction volume of $120 billion dollars in 2017.

The distributed software systems that run their entire business produced around 40 billion log lines solely in July 2018. Due to their scale and sensitive business market, monitoring is a vital activity at Adyen. Adyen follows DevOps practices as part of their culture, and the barriers between development and production have been getting smaller and smaller over the years. Developers of all teams are responsible for the monitoring of their systems and are supported by a dedicated monitoring application, whose focus is to build any customization a team might need to conduct better monitoring. Thus, at Adyen, monitoring is a vital task for all developers.

Due to their efforts on monitoring over the last years, we firmly believe that Adyen offers an exemplary place for software engineering researchers to study (and evolve) monitoring and DevOps practices. And for this research, more specifically, to study the benefits of Monitoring-Aware IDEs.

Adyen's monitoring and DevOps practices. In Figure 1, we summarize Adyen's monitoring and DevOps practices. The model contains ten practices (P1..P10) grouped in six broad themes. Throughout the following text, we use circles to connect the model in the Figure to the explaining text, *e.g.*, (P1) refers to practice number 1.

At Adyen, 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, 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
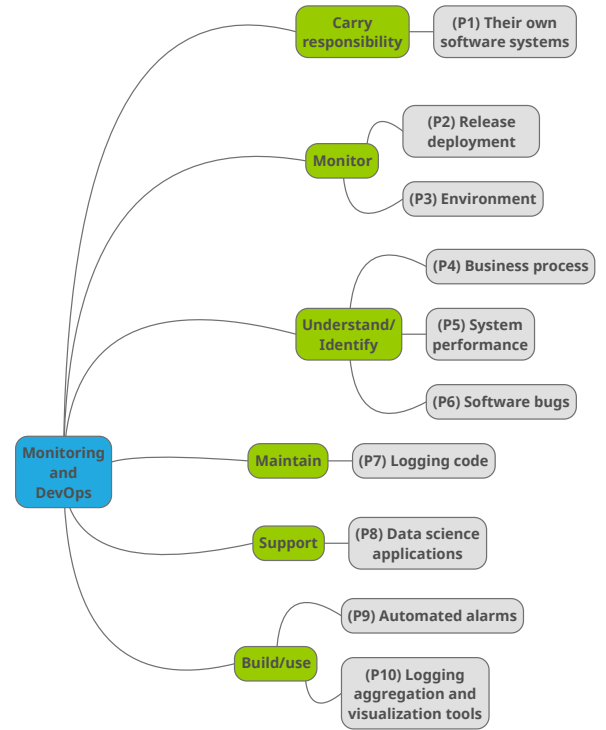

Figure 1: Monitoring and DevOps practices at Adyen.

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, have 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 (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 (see Section 2.1), 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.

3 MONITORING-AWARE IDES

In modern teams following a DevOps model, **developers go back and forth between monitoring data and the source code to reason about their software systems**. Even with the current state-of-the-art monitoring and IDE/development tools, developers still struggle with connecting the two worlds. The current situation leads to increased context-switching [18] and split attention effects [13] that increase cognitive load.

We theorize that, for developers to be better equipped to deal with monitoring and DevOps practices, IDEs and monitoring systems should be connected (giving rise to what we will call, *Monitoring-Aware IDEs*). A Monitoring-Aware IDE provides developers with an integrated view of both the implementation of their software systems and monitoring information.

Developers need not to go out of their IDEs to know whether an exception that they just decided to throw happened ten times in the last week, or that the time between two log statements has been increasing continually. Based on what we observe at Adyen, we conjecture that such an IDE would:

(1) Assist developers in monitoring their new features and release deployments and, as a consequence, provide them with enough information to identify bugs and performance issues,
(2) Assist developers in using log data to understand the business process of software systems, and
(3) Assist developers in maintaining logging code, such as extending or removing log statements from the source code.

To achieve this goal, we propose that a Monitoring-Aware IDE must have the following characteristics:

(1) **Timely Integrated Feedback:** Monitoring data, *e.g.*, how often a log statement or an exception happens in production, should be timely available at the Monitoring-Aware IDE, so that developers can make data-driven decisions based on the most recent data (and without the need of opening the monitoring system for that),
(2) **Traceability:** There should be a direct connection/link between the monitoring information and the source code, in case one tool does not contain the required information at that moment. The source of monitoring information (*e.g.*, a log statement or an exception) can be found based on monitoring information, and monitoring information can be found based on its source.
(3) **Search Capability:** Monitoring information should be searchable in the IDE, *e.g.*, the classes with the highest number of exceptions.
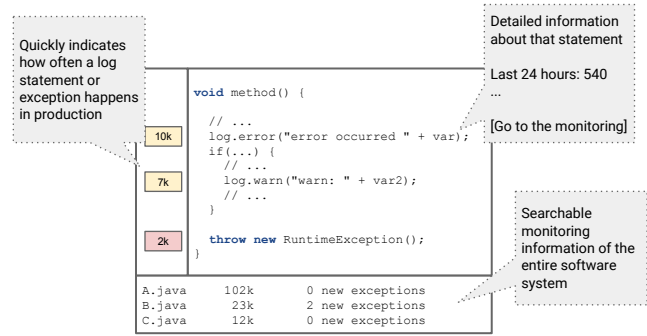

Figure 2: Interaction design of a Monitoring-Aware IDE. Numbers on the left bar indicate how often a log statement or exceptions happen in production. Developers can ask for more detailed monitoring information (box on the right) or, as last resource, go to the real monitoring system and observe the full data there. Finally, search options at the bottom of the IDE (e.g., filter by class name, order by exception frequency).

4 MONITORING-AWARE IDE PROTOTYPE

To empirically study our proposal, we built a prototype of a Monitoring-Aware IDE. We set the following goals for the prototype:

(1) it should deliver enough value to Adyen developers, so that they would benefit from this study,
(2) to be as non-obtrusive as possible to Adyen developers, so that they would not feel the burden of using an "unknown" tool,
(3) to deliver enough features so that we, as researchers, could empirically validate our Monitoring-Aware IDEs proposal.

We highlight the fact that this tool was also developed inside in partnership with Adyen, incorporating iterative feedback (from February to June 2018). Throughout its five months of development, our prototype received feedback from several Adyen developers after beta versions. The first three authors of the paper discussed all their suggestions and whether they were useful or essential for the prototype. In this paper, we report the final version of the prototype.

Our tool collects monitoring data that is currently available in Adyen's monitoring systems (ELK stack). Adyen allowed us to collect new data from their monitoring systems every 15 seconds, which gives near real-time information. We trace back the origin of a log message to its original log statement in the source code using heuristics (Adyen does not log class name and line number that originates the message due to performance reasons). More specifically, we generate regular expressions based on every log statement of the source code and matched them against the log message that comes from the monitoring system, following Xu's et al. work [61]. We observed that the link, as also reported by Xu et al., indeed happens with high accuracy (97% after an evaluation in 100k log messages), which implies that our tool is accurately able to show monitoring information in the source code. Finally, we show the monitoring information inside IntelliJ, the Java IDE that is used

at Adyen, by means of a plugin that we developed. We discuss the details of the prototype's architecture in Section 7.1.

In Figure 2, we present an interaction design of how the tool presents information to developers[6]. The tool supports all the requirements we set out in Section 3. Whenever developers open any class in their source code, our tool shows monitoring information near all the log statements and thrown exceptions. The information is continuously extracted from Elasticsearch, the underlying document database Adyen uses to store the monitoring data of their production systems. The numbers near every log statement in the left box show how often they have been triggered in the last month. When developers hover with their mouse, our tool shows a summary of monitoring information about that statement (currently, how often that statement was executed in the last hour, 24 hours, and month). To facilitate the switching to the monitoring tooling with more detailed information, we also provide a direct traceability link to the dashboard of that specific class and log statement. Finally, the tool also provides developers with search options, such as filter by class name, and order by exception frequency.

5 FIELD EXPERIMENT

In the remainder of this paper, we take the first step towards **empirically understanding the value of Monitoring-Aware IDEs** we posed in the previous section. To that aim, we propose three research questions:

RQ$_1$. How do developers interact with a Monitoring-Aware IDE?

RQ$_2$. What impact does a Monitoring-Aware IDE bring to software development teams?

RQ$_3$. What are the developers' perceptions about the usefulness of a Monitoring-Aware IDE to support their monitoring practices?

Given the complexity of simulating an environment that requires constant monitoring, such as the likes of Adyen, we opted for a *field experiment*. According to Stol and Fitzgerald [48], a field experiment refers to an experimental study conducted in a natural setting with a high degree of realism. In this strategy, the researcher manipulates some properties in the research setting with the goal of observing an effect of some kind. Also, according to Stol and Fitzgerald, the natural study setting is realistic, but subject to confounding factors that can limit the precision of measurement.

To that aim, we make use of quantitative and qualitative data that we collected after providing 12 developers from Adyen with a Monitoring-Aware IDE prototype for four weeks. In summary, our field experiment happened as follows:

(1) We recruited 12 developers from Adyen (the selection criteria is explained in Section 5.2), installed the prototype in their IDEs, and gave them a short tutorial on what the prototype does and how it works,

(2) The 12 participants used our Monitoring-Aware IDE prototype for four weeks to perform their daily tasks,

(3) We collected information about the usage of the prototype, automatically via telemetry,

(4) We collected information about the impact of the tool through a weekly survey,

(5) At the end of the four weeks, we performed a final survey with the 12 participants to understand their overall perception of the benefits of a Monitoring-Aware IDE.

5.1 Methodology

Data collection and analysis. We added instrumentation to our prototype that collects the following interactions between the developer and the tool: (1) when the developer opens a file containing source code for which monitoring data exists, (2) when the developer asks for detailed monitoring information in a specific line of code as well as how much time they spend on it, and (3) when the developers opt to navigate to the real monitoring system.

To understand whether and how the IDE impacted developers in their development tasks (RQ$_2$), we surveyed the participants weekly, asking about their specific interactions with the tool and what actions they took.

We created surveys tailored for each developer. Based on all the usage data collected from our prototype during that week, we showed a list of all classes in which participants observed any monitoring information during that week. For each of these classes, participants had to answer questions about in what way the tool impacted (or did not impact) their work.

We provided participants with a list of possible follow up actions that one could have taken after having analyzed the monitoring information. We devised this list of consequences in collaboration with Adyen developers (using their monitoring and DevOps practices as a basis, see Section 2.2 and Figure 1). We also give developers a free box where they can provide any other action. We iteratively monitored their open answers to improve our list. We also provided a "did not perform any action" option, so that participants would not feel obliged to choose any consequence.

The final list can be divided into three categories: observations, code changes, and logging code improvements.

- *Observations:* Insights into the behavior of their systems, based on monitoring data: (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, and (O6) Understood the stability of the implementation.

- *Code changes:* Production-code improvements, based on monitoring data: (I1) Fixed a bug, (I2) Improved code quality (refactoring), (I3) Improved code performance, (I4) Improved code security, and (I5) Implemented new functionality.

- *Logging code improvements:* Improvements to the log code based on monitoring data: (L1) Improved log message, (L2) Changed log severity, (L3) Removed log line, and (L4) Added log line.

Although participants may have worked in the same class and asked for its detailed monitoring information (maybe for different purposes) multiple times during the week, that class appeared only once in that week's survey. We made this decision for two reasons: 1) we do not believe participants would have an accurate perception and memory for such a fine-grained survey, 2) the survey would be too extensive as we conjectured that participants would interact with a large number of classes a week. Nevertheless, we allowed participants to choose multiple actions for the same class, which

---

[6]We can not show an actual screenshot of the tool being used as it would reveal proprietary information.

would enable them to express multiple actions they might have taken in that class during that entire week.

In addition, some of the possible interactions with our tool cannot be automatically collected by our prototype (*e.g.*, we have no data to infer whether participants looked at the number we show in front of any log statement). Thus, at the end of the survey, we ask them whether the tool helped (or not helped) in any way that we did not ask before.

Post-questionnaire. Finally, with the goal of augmenting and explaining the data we obtained employing the weekly surveys and the prototype, we asked participants to answer an open questionnaire at the end of the four weeks (P1, P6, and P7 were unavailable for the questionnaire).[7] Questions were based on the results we had obtained until that moment.

The questionnaire contained open questions about both their usage of the tool as well as the impact the tool had on their daily jobs. More specifically, about the tool usage, we asked:

(1) Did you look at the monitoring data we provide at the left bar of your IDE? In your opinion, how important and/or useful are they?
(2) We noticed that you went to the external monitoring while using our tool. Why did you go there?

Concerning the impact of the tool, we asked the following two questions for each of the five most perceived benefits (represented by *<X>* in the following questions):

(1) How does the tool help you in doing *<X>*?
(2) How did you perform *<X>* before having a Monitoring-Aware IDE? What are the differences?

Note that we use this post-questionnaire also as a way to collect perceptions on the comparison between using and not using a Monitoring-Aware IDE, given that establishing a control group is not possible in the context of our study. We use the questionnaire as a way to mitigate the possible threat, which we discuss in detail in Section 7.2.

Data analysis. We applied descriptive statistics to all quantitative data we collected (*i.e.*, usage data coming from the prototype and survey answers). We analyzed the post-questionnaire data using the following procedure:

(1) To each of the questions in the questionnaire, we grouped similar answers in high-level themes.
(2) Whenever a new theme was created, we revisited all the previous answers to that question, and evaluated whether it would better fit the new theme,
(3) We stopped the process when there were no more themes to create.

The first two authors were involved in the coding of the data and in deriving higher-level themes. We use the high-level themes as main topics of discussion in our Results section.

Ethical concerns. We do not collect sensitive or private information from the developers or from Adyen in any of the steps of our field experiment. All the participants were aware of all the data being collected before joining the study. Besides, this field experiment

---

[7]While P1, P6, and P7 did not participate in the post-questionnaire, they provided data for RQs 1 and 2, which we used in the analysis.

Table 1: Profile of the participants in our study. Participants are ordered according to the number of interactions with the tool (P1 interacted the most, P12 interacted the least).

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

was also approved by the Ethics Committee of Delft University of Technology.

5.2 Participants

We invited 12 developers (from 7 different teams) to use our prototype for four weeks. We applied convenience sampling to find the 12 participants of our study. We made a general announcement at Adyen's internal chat application explaining our study and prototype and asked for participants. All participants had to pass the following criteria: (1) more than one year of experience as a software developer, (2) more than six months of experience at Adyen, and (3) a frequent user of Adyen's monitoring systems. We show participants' profiles in Table 1.

We asked participants to perform their regular development tasks using our prototype. Before the field experiment, we gave participants some time to try out the tool and learn how to use it. We highlight that, during these four weeks, we did not force or require developers to use our tool in any situation, as we wanted to observe their real-world behavior.

6 RESULTS

6.1 RQ$_1$: How do developers interact with a Monitoring-Aware IDE?

In Figures 3a and 3b, 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, left bar in Figure 2) was perceived as useful by developers: such information enabled them to quickly observe whether
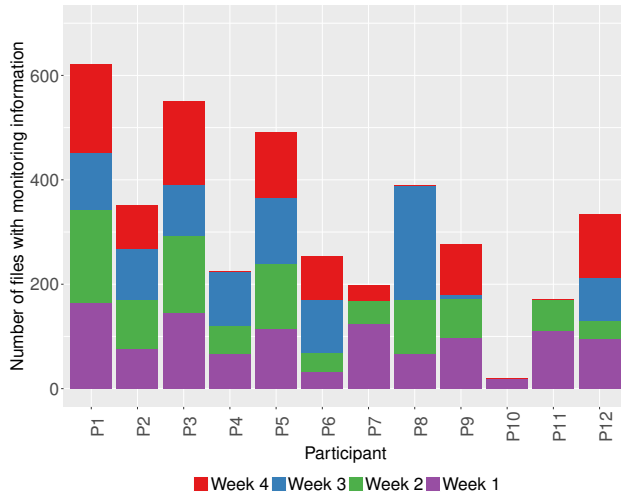
(a) The number of times our IDE displayed a file that contained monitoring information (total=3,879).
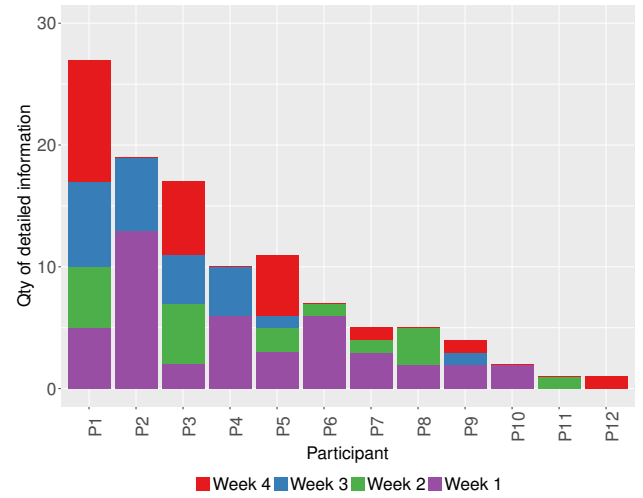
(b) The number of times a participant asked for more detailed information in a log statement or exception (total=109).

Figure 3: How much participants used our Monitoring-Aware IDE (N=12 participants).

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: *"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 also visited the actual monitoring tool to retrieve additional, more detailed 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).

Interestingly, we observed 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.

> P12: *"I myself go back to things I worked on from time to time as well."*

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.

6.2 RQ₂: What impact does a Monitoring-Aware IDE bring to software development teams?

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 4.

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 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 3b). 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 Adyen already 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
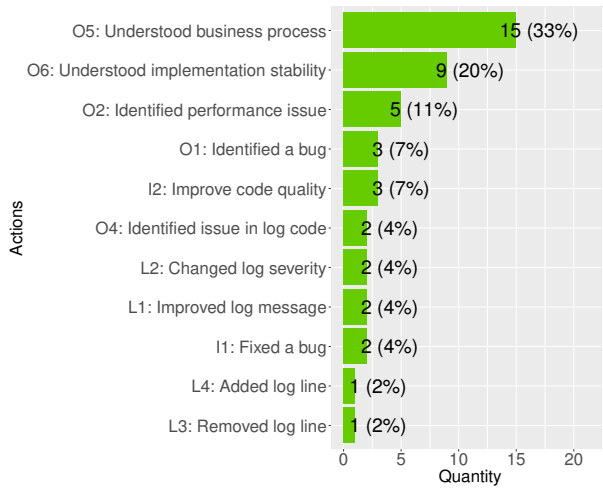
Figure 4: How our Monitoring-Aware IDE impacted our developers (N=45, 12 participants).

their logging code (8 times, 16%). We observed developers adding new log lines (1 times, 2%), improving an existing log message (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 Adyen has already a secure software and security issues do not often happen (and thus the likelihood of such an issue to happen during our field experiment 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.

6.3 RQ₃: What are the developers' perceptions about the usefulness of a Monitoring-Aware IDE to support their monitoring practices?

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 spending less time querying their monitoring systems (P2, P3, P4, P8).

P2: *"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."*

P3: *"[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 establishing traceability by 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: *"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: *"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.

P8: *"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 (corroborating the results of RQ₂). 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 a lot of attention to the frequency of a log statement. Developers seem to implicitly formulate hypotheses on behavior in production. The frequency allows them to immediately make judgments about their hypotheses, *i.e.*, whether this number seems to be "out of place" (*e.g.*, near 0, or very large).

P5: *"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.

P5: *"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: *"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 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).
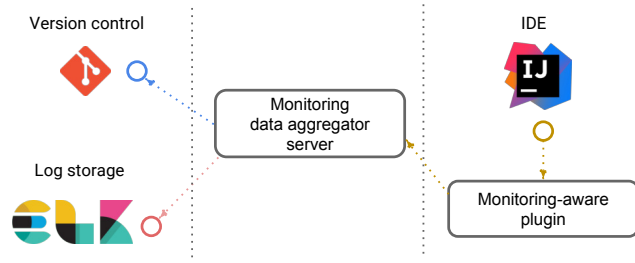
Figure 5: High-level architecture of our Monitoring-Aware IDE. The monitoring data aggregator is responsible for aggregating and linking data from both data sources and for providing monitoring data for the IDE plugin.

> P5: *"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."*

> P8: *"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 entirely replace their existing tooling, but rather complements it. 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: *"Don't 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.

7 DISCUSSION

In the following, we discuss the several challenges of building a Monitoring-Aware IDEs, and how we mitigate possible threats to the validity of this study.

7.1 Building Monitoring-Aware IDEs

The Architecture of a Monitoring-Aware IDE. Designing such an IDE, from an architectural point of view, is worth discussing. Monitoring data can be extensively large (as with our industry partner) and any (local) data analysis might take too much, or even crash the IDE. Thus, Monitoring-Aware IDEs should be designed

with scalability in mind. Our prototype has been shown to be scalable, and thus, we dedicate the next paragraphs to describe our architectural decisions.

As we show in Figure 5, the *monitoring data aggregator* is a large process that runs in a separate server. It is where most of the expensive computational calculations (e.g., parse log data and generate templates, match the log data with its original log statement, update counters, pull up-to-date source code and refresh templates) happen. The Monitoring-Aware IDE is implemented as a plugin on top of an existing IDE, such as IntelliJ. The plugin mostly queries data from the aggregator and shows it to the developer. No heavy calculations happen in the IDE, which means developers do not suffer from possible slowliness.

On the other hand, we still see performance improvements to be done. Our current prototype queries Adyen's monitoring systems every 15 seconds for new log data. Due to Adyen's weekly release cycles, we also re-generate the regular expressions from their source code every week (and not at every new commit, as the generation process currently takes 35 minutes). We refresh the monitoring information in the developers' IDEs whenever they open a class. While this currently gives near real-time up-to-date information to developers, we see the following steps as required to build a state-of-the-art real-time Monitoring-Aware IDE:

(1) A streaming system in place that would stream log data as they come would be needed. Current industry solutions, like the ELK stack, offer such streaming.
(2) The monitoring data aggregator would have to be able to handle the vast amount of regular expression matching that would happen for each log message. Matching regular expressions is neither a cheap or fast operation, particularly in languages like Java, which implements a Nondeterministic Finite Automaton (NFA) backtracking algorithm [19]. We see parallelization as a future requirement.
(3) The monitoring data aggregator would have to generate new regular expressions from the source code every time a new deploys happens. Our current regular expression generator takes around 35 minutes to run in a codebase with a few million lines of code[8], and can take even longer in larger codebases.
(4) The IDE and the monitoring aggregator server would have to periodically communicate with each other, so that the IDE always has up-to-date data. The communication should happen in a way that developers do not notice any delays in their IDEs.

The Importance of Logging Code. It is interesting to notice how important the quality of the log code is, and how much developers monitored and improved their quality. Throughout our study, developers fixed issues, added, and removed log code.

The quality of log lines is indeed important, and researchers have been working on log code best practices. Fu *et al.* [25], for example, studied common logging practices especially focusing on where in the source code developers log. They conclude that common logging practices can be used to automate the logging process partially. Zhu *et al.* [64] implemented a tool which learns common logging practices and uses it to indicate positions that can

---

[8]We are not allowed to disclosure the total LOC of their systems.

be improved by adding a log statement. Chen and Jiang [14] studied anti-patterns, which are defined as recurring mistakes in logging code, which may hinder the understanding and maintainability of log statements. Therefore, given that developers are now quite used to use static analysis tools (or linters) to spot bugs and maintenance issues [8, 51, 52], we suggest tool makers to start incorporating such log code quality measures in their linters.

As an orthogonal aspect, Adyen uses Log4J, the most popular Java logging framework. Given their scale and the number of requests per second their servers receive, Adyen developers can not store the line number of the log statement that originates a log line that one sees in the monitoring tool. This is why we use Xu et al.'s heuristic [61] to link the log line back to its originating log statement. However, although the heuristic has worked well in our settings, our developers had a good amount of implementation work to adapt it to Adyen's code style. From the practical point of view, we see, as future work, logging frameworks being able to log meta-information (*e.g.*, class name, line number) with reduced computational costs.

Custom-made Monitoring-Aware IDEs. Adyen uses Elasticsearch and Kibana dashboards to monitor their systems. We observed that developers pay a great attention to the number and type of exceptions that are going on in production as well as how the (new) code they wrote is behaving. The features of the Monitoring-Aware IDE prototype we study in this paper were based on these observations. However, developers of a different company may use monitoring systems in a different way, *e.g.*, customized metrics or analysis.

Monitoring-aware IDEs should also provide the extensive flexibility that current monitoring tools offer to developers. This means that the perfect IDE for one team might be different than the one for another team. This raises interesting points for IDE makers: how to make a monitoring feature that is generic enough for most developers to use, but customizable enough so that developers can obtain all the benefits that their current monitoring systems offer?

Connected IDEs. We bring to attention the fact that we are used to seeing IDEs as standalone tools. After installation, they tend not to require any connections with the external world and developers can use it even without a network connection. In a world where IDEs are strongly connected with monitoring, both worlds should talk to each other. IDEs should not be standalone tools anymore.

Researchers indeed have been studying cloud-based IDEs [2, 27, 54, 56, 59], and companies have been developing them (*e.g.*, Amazon's Cloud9). Cloud-based IDEs eliminate any need for specific hardware or operational systems, and try to increase collaboration and coding among developers. We argue that the ideas of cloud-based IDEs are in line with Monitoring-Aware IDEs. We conjecture that the fact that cloud-based IDEs naturally exist in a cloud environment would facilitate the development of the monitoring features we suggest in this paper.

Fylaktopoulos *et al.* [26] noticed that runtime monitoring (or auditing, as authors call in their paper) is still an area not yet explored in such IDEs. Authors discuss how developers are currently required to build their own debugging and auditing tools outside of IDEs. We suggest researchers to explore the connection between cloud-based and Monitoring-Aware IDEs.

7.2 Threats to Validity

Internal Validity. (1) We use our prototype as a proxy to understand the impact of a Monitoring-Aware IDE in software development teams. As we present in Section 5, our Monitoring-Aware IDE prototype contains features that we derived from Adyen's monitoring and DevOps practices (Section 2.2). We do not claim that our prototype fully represents and/or contains all possible features of an idealistic Monitoring-Aware IDE. We consider, nevertheless, our prototype sufficient enough to provide initial evidence that such an IDE can provide benefits to developers; (2) Participants P1, P6, and P7 were not available during the post-questionnaire. Nevertheless, we do not believe it affects in any way our conclusions, given that the answers of all other participants clearly converged; (3) We did not have an explicitly controlled baseline in our field experiment, 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 analysis. We deem this setting to be appropriate given our goal to collect qualitative insights into how developers interact with our approach in their natural workflow. As future work, we plan to replicate our study in a more controlled setting, now that we have a better insight into what can/should be used as independent and dependent variables.

External Validity. This entire research was conducted at Adyen, 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. Although we diversified our field experiment with developers from seven different teams that represent various kinds of development contexts, we can not claim any generalization. However, given the size, scale, and importance of the software built by Adyen, we believe this idea is worthy of further investigation.

8 CONCLUSIONS

Software developers reason about the behavior of large-scale software systems in production by examining log data in external monitoring tools. However, most of their software development activity happens in the source code view in the IDE. Leaving their development workflow in the IDE to understand production software behavior leads to increased context-switching and split attention effects that increase cognitive load.

We propose to unify both development and monitoring contexts by developing a new concept of Monitoring-Aware IDEs. We integrate monitoring aspects into the workflow and context of software development tasks by incorporating frequency information on log statements into the source code view of an IDE. We implement this concept as an IntelliJ plugin and conduct a one-month field experiment with 12 developers in a large company, Adyen. Developers using our approach in the field experiment reported that they were able to better understand business processes, identify performance issues and functional bugs, improve code quality, and better maintain their logging code.

We firmly believe that Monitoring-Aware IDEs plays an essential role in improving how developers interact with monitoring to reason about production behavior and take action in development.

REFERENCES

[1] Jenny Abrahamson, Ivan Beschastnikh, Yuriy Brun, and Michael D Ernst. 2014. Shedding light on distributed system executions. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 598–599.

[2] Timo Aho, Adnan Ashraf, Marc Englund, Joni Katajamäki, Johannes Koskinen, Janne Lautamäki, Antti Nieminen, Ivan Porres, and Ilkka Turunen. 2011. Designing IDE as a service. *Communications of Cloud Software* 1, 1 (2011).

[3] Juan Pablo Sandoval Alcocer, Alexandre Bergel, Stéphane Ducasse, and Marcus Denker. 2013. Performance evolution blueprint: Understanding the impact of software evolution on performance. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*. IEEE, 1–9.

[4] Joop Aué, Maurício Aniche, Maikel Lobbezoo, and Arie van Deursen. 2018. *An Exploratory Study on Faults in Web API Integration in a Large-Scale Payment Company*. https://doi.org/10.1145/3183519.3183537

[5] David Basin, Germano Caronni, Sarah Ereth, Matúš Harvan, Felix Klaedtke, and Heiko Mantel. 2014. Scalable offline monitoring. In *International Conference on Runtime Verification*. Springer, 31–47.

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

[7] Fabian Beck, Oliver Moseler, Stephan Diehl, and Gunter Daniel Rey. 2013. In Situ Understanding of Performance Bottlenecks Through Visually Augmented Code. In *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, Los Alamitos, CA, USA, 63–72. https://doi.org/doi.ieeecomputersociety.org/10.1109/ICPC.2013.6613834

[8] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 470–481.

[9] Christophe Bertero, Matthieu Roy, Carla Sauvanaud, and Gilles Trédan. 2017. Experience Report: Log Mining using Natural Language Processing and Application to Anomaly Detection. In *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE, 351–360.

[10] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. 2011. 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, 267–277.

[11] Cor-Paul Bezemer, Johan Pouwelse, and Brendan Gregg. 2015. Understanding software performance regressions using differential flame graphs. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*. IEEE, 535–539.

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

[13] Paul Chandler and John Sweller. 1992. The split-attention effect as a factor in the design of instruction. *British Journal of Educational Psychology* 62, 2 (1992), 233–246.

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

[15] Hsin-Jung Cheng and Akhil Kumar. 2015. Process mining on noisy logs—Can log sanitization help to improve performance? *Decision Support Systems* 79 (2015), 138–149.

[16] Jürgen Cito, Philipp Leitner, Harald C Gall, Aryan Dadashi, Anne Keller, and Andreas Roth. 2015. 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, 14–27.

[17] Jürgen Cito, Philipp Leitner, Martin Rinard, and Harald C. Gall. 2019. Interactive Production Performance Feedback in the IDE. In *Proceedings of the 41st International Conference on Software Engineering (ICSE '19)*. IEEE Press, Piscataway, NJ, USA, 971–981. https://doi.org/10.1109/ICSE.2019.00102

[18] Gregorio Convertino, Jian Chen, Beth Yost, Y-S Ryu, and Chris North. 2003. Exploring context switching and cognition in dual-view coordinated visualizations. In *Coordinated and Multiple Views in Exploratory Visualization, 2003. Proceedings. International Conference on*. IEEE, 55–62.

[19] Russ Cox. 2007. Regular expression matching can be simple and fast (but is slow in java, perl, php, python, ruby,...). *URL: http://swtch. com/˜ rsc/regexp/regexp1. html* (2007).

[20] Dominica DeGrandis. 2011. Devops: So you say you want a revolution? *Cutter IT Journal* 24, 8 (2011), 34.

[21] Andrej Dyck, Ralf Penners, and Horst Lichter. 2015. Towards definitions for release engineering and devops. In *Release Engineering (RELENG), 2015 IEEE/ACM 3rd International Workshop on*. IEEE, 3–3.

[22] Florian Fittkau, Alexander Krause, and Wilhelm Hasselbring. 2017. Software landscape and application visualization for system comprehension with ExplorViz. *Information and software technology* 87 (2017), 259–277.

[23] Ilenia Fronza, Alberto Sillitti, Giancarlo Succi, Mikko Terho, and Jelena Vlasenko. 2013. Failure prediction based on log files using Random Indexing and Support Vector Machines. *Journal of Systems and Software* 86, 1 (2013).

[24] Qiang Fu, Jian-Guang Lou, Yi Wang, and Jiang Li. 2009. Execution anomaly detection in distributed systems through unstructured log analysis. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 149–158.

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

[26] George Fylaktopoulos, Georgios Goumas, Michael Skolarikis, Aris Sotiropoulos, and Ilias Maglogiannis. 2016. An overview of platforms for cloud based development. *SpringerPlus* 5, 1 (2016), 38.

[27] Lakshmi M Gadhikar, Lavanya Mohan, Megha Chaudhari, Pratik Sawant, and Yogesh Bhusara. 2013. Browser based IDE to code in the cloud. In *New Paradigms in Internet Computing*. Springer, 59–69.

[28] Maayan Goldstein, Danny Raz, and Itai Segall. 2017. Experience Report: Log-Based Behavioral Differencing. In *Software Reliability Engineering (ISSRE), 2017 IEEE 28th International Symposium on*. IEEE, 282–293.

[29] C.W. Günther and W.M.P. Aalst, van der. 2007. Fuzzy mining - adaptive process simplification based on multi-perspective metrics. In *Proceedings of the 5th International Conference on Business Process Management (BPM 2007) 24-28 September 2007, Brisbane, Australia (Lecture Notes in Computer Science)*, G. Alonso, P. Dadam, and M. Rosemann (Eds.). Springer, Germany, 328–343. https://doi.org/10.1007/978-3-540-75183-0_24

[30] Marijn JH Heule and Sicco Verwer. 2010. Exact DFA identification using SAT solvers. In *International Colloquium on Grammatical Inference*. Springer, 66–79.

[31] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting Code with In Situ Visualizations to Aid Program Understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 532.

[32] Michael Hüttermann. 2012. *DevOps for developers*. Apress.

[33] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. 2009. Automated performance analysis of load tests. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*. IEEE, 125–134.

[34] Kamal Kc and Xiaohui Gu. 2011. ELT: Efficient log-based troubleshooting system for cloud computing infrastructures. In *IEEE Symposium on Reliable Distributed Systems*. IEEE, 11–20.

[35] Tom Lieber, Joel R Brandt, and Rob C Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the 32nd annual ACM conference on Human factors in computing systems*. ACM, 2481–2490.

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

[37] Nicolas Lopez and André Van Der Hoek. 2011. The code orb: supporting contextualized coding via at-a-glance views (NIER track). In *2011 33rd International Conference on Software Engineering (ICSE)*. IEEE, 824–827.

[38] Mike Loukides. 2012. *What is DevOps?* " O'Reilly Media, Inc.".

[39] Welder Pinheiro Luz, Gustavo Pinto, and Rodrigo Bonifácio. 2018. Building a collaborative culture: a grounded theory of well succeeded devops adoption in practice. In *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. ACM, 6.

[40] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 26–26.

[41] Adam Oliner, Archana Ganapathi, and Wei Xu. 2012. Advances and challenges in log analysis. *Commun. ACM* 55, 2 (2012), 55–61.

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

[43] Nicolas Poggi, Vinod Muthusamy, David Carrera, and Rania Khalaf. 2013. Business process mining from e-commerce web logs. In *Business process management*. Springer, 65–80.

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

[45] Kim Weins (Rightscale). [n.d.]. New DevOps Trends: 2016 State of the Cloud Survey. "https://www.rightscale.com/blog/cloud-industry-insights/new-devops-trends-2016-state-cloud-survey". Accessed January, 2019).

[46] James Roche. 2013. Adopting DevOps practices in quality assurance. *Commun. ACM* 56, 11 (2013), 38–43.

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

[48] Klaas-Jan Stol and Brian Fitzgerald. 2018. The ABC of Software Engineering. *IEEE Transactions on Software Engineering and Methodology* (2018). https://doi.org/10.1145/3241743 In press.

[49] Yang Sun, Huajing Li, Isaac G Councill, Jian Huang, Wang-Chien Lee, and C Lee Giles. 2008. Personalized ranking for digital libraries based on log analysis. In *Proceedings of the 10th ACM workshop on Web information and data management*. ACM, 133–140.

[50] Mark D Syer, Zhen Ming Jiang, Meiyappan Nagappan, Ahmed E Hassan, Mohamed Nasser, and Parminder Flora. 2013. Leveraging performance counters and execution logs to diagnose memory-related performance issues. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*. IEEE, 110–119.

[51] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie van Deursen. 2017. Why and how JavaScript developers use linters. In *Automated Software Engineering (ASE), 2017 32nd IEEE/ACM International Conference on*. IEEE, 578–589.

[52] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. 2018. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering* (2018).

[53] Jan Martijn EM Van der Werf, Boudewijn F van Dongen, Cor AJ Hurkens, and Alexander Serebrenik. 2008. Process discovery using integer linear programming. In *International conference on applications and theory of petri nets*. Springer, 368–387.

[54] Arie van Deursen, Ali Mesbah, Bas Cornelissen, Andy Zaidman, Martin Pinzger, and Anja Guzzi. 2010. Adinda: A Knowledgeable, Browser-based IDE. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2 (ICSE '10)*. ACM, New York, NY, USA, 203–206. https://doi.org/10.1145/1810295.1810330

[55] Mandi Walls. 2013. *Building a DevOps culture*. " O'Reilly Media, Inc.".

[56] Yi Wang, Patrick Wagstrom, Evelyn Duesterwald, and David Redmiles. 2014. New opportunities for extracting insights from cloud based IDEs. In *Companion Proceedings of the 36th International Conference on Software Engineering*. ACM, 408–411.

[57] Rick Wieman, Maurício Finavaro Aniche, Willem Lobbezoo, Sicco Verwer, and Arie van Deursen. 2017. 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, 564–573.

[58] W Eric Wong, Vidroha Debroy, Richard Golden, Xiaofeng Xu, and Bhavani Thuraisingham. 2012. Effective software fault localization using an RBF neural network. *IEEE Transactions on Reliability* 61, 1 (2012), 149–169.

[59] Ling Wu, Guangtai Liang, Shi Kui, and Qianxiang Wang. 2011. CEclipse: An online IDE for programing in the cloud. In *Services (SERVICES), 2011 IEEE World Congress on*. IEEE, 45–52.

[60] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael Jordan. 2009. Online system problem detection by mining patterns of console logs. In *Data Mining, 2009. ICDM'09. Ninth IEEE International Conference on*. IEEE, 588–597.

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

[62] Xiuming Yu, Meijing Li, Incheon Paik, and Keun Ho Ryu. 2012. Prediction of web user behavior by discovering temporal relational rules from web log data. In *International Conference on Database and Expert Systems Applications*. Springer, 31–38.

[63] Xu Zhao, Yongle Zhang, David Lion, Muhammad Faizan Ullah, Yu Luo, Ding Yuan, and Michael Stumm. 2014. lprof: A non-intrusive request flow profiler for distributed systems. In *OSDI*, Vol. 14. 629–644.

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

[65] De-Qing Zou, Hao Qin, and Hai Jin. 2016. Uilog: Improving log-based fault diagnosis by log analysis. *Journal of Computer Science and Technology* 31, 5 (2016), 1038–1052.