SN Computer Science (2021) 2:447
https://doi.org/10.1007/s42979-021-00826-y

ORIGINAL RESEARCH

# Closing the Feedback Loop in DevOps Through Autonomous Monitors in Operations

**Adha Hrusto**[1,2] · **Per Runeson**[1] · **Emelie Engström**[1]

Received: 2 April 2021 / Accepted: 18 August 2021 / Published online: 7 September 2021


## Abstract

DevOps represent the tight connection between development and operations. To address challenges that arise on the borderline between development and operations, we conducted a study in collaboration with a Swedish company responsible for ticket management and sales in public transportation. The aim of our study was to explore and describe the existing DevOps environment, as well as to identify how the feedback from operations can be improved, specifically with respect to the alerts sent from system operations. Our study complies with the basic principles of the design science paradigm, such as understanding and improving design solutions in the specific areas of practice. Our diagnosis, based on qualitative data collected through interviews and observations, shows that alert flooding is a challenge in the feedback loop, i.e. too much signals from operations create noise in the feedback loop. Therefore, we design a solution to improve the alert management by optimizing when to raise alerts and accordingly introducing a new element in the feedback loop, a smart filter. Moreover, we implemented a prototype of the proposed solution design and showed that a tighter relation between operations and development can be achieved, using a hybrid method which combines rule-based and unsupervised machine learning for operations data analysis.

**Keywords** DevOps · Development · Operations · Design science

## Introduction

The software industry has gone through several revolutionary changes over the last decades. A major change is that software is no longer delivered as a box product. Technological advancements and availability of cloud computing platforms have enabled continuous delivery of *software systems* leveraging the flexibility and reliability of various *cloud* delivery solutions [1]. Moreover, cloud providers offer an infrastructure for developing and operating large-scale software systems empowered by continuous practices and DevOps, the latest industry concept based on principles of collaboration, automation, measurements, and monitoring [2]. However, it also comes with an abundance of data to be managed as it is considered to be the *fuel* of the DevOps process [3].

The software life cycle includes continuous integration, continuous testing, and continuous deployment practices [4]. During deployment, software systems are transitioned from development to operations, to be continuously used by end-users. The connection between development (Dev) and operations (Ops), known as DevOps, ensures faster development cycles and frequent releases. However, keeping the same level of software quality becomes challenging due to shorter testing cycles. Run-time monitoring of services in operations [5], which is the focus of this study, is of high importance for gaining confidence in a software system and providing feedback to the development.

This article is part of the topical collection "New Paradigms of Software Production and Deployment" guest edited by Alfredo Capozucca, Jean-Michel Bruel, Manuel Mazzara and Bertrand Meyer.

✉ Adha Hrusto
adha.hrusto@cs.lth.se; adha.hrusto@systemverification.com

Per Runeson
per.runeson@cs.lth.se

Emelie Engström
emelie.engstrom@cs.lth.se

[1] Department of Computer Science, Lund University, Lund, Sweden

[2] System Verification AB, Malmö, Sweden

Through the run-time monitoring system, a vast amount of data is continuously collected and saved for manual or automatic analysis. The data analysis serves as feedback to development teams and provides deep and quick insight into the status of the software system during operational execution [3]. Consequently, developers and project managers can act as soon as they are notified about anomalies. The notification is typically implemented as *alerts* sent through a messaging platform, like Slack, triggered by alert rules, which are defined as functions of the operational data. However, the abundance of data and particularly alerts from minor or major malfunctions in system components, tend to flood over the developers and create noise that drowns the important alerts.

In the literature, there are examples of various methods for the analysis of operations data but only a few are addressing real industrial needs and challenges companies are facing in relation to the feedback from operations to development [6]. Consequently, there is a limited choice of potential solutions available in the literature for designing more context-specific solution designs based on the identified industrial needs. *Thus, with our research, we aim to fill this gap by addressing challenges related to the flow—and overflow—of data from operations to development.* We intend to explore and improve existing solution designs in the context of the case company's feedback loop from operations to development. Thus our study complies with the principles of a design science paradigm [7].

We conducted a study in collaboration with a Swedish company responsible for ticket management and sales in public transportation. Their main product is the back-end system for ticketing and payments, developed and operated in a DevOps environment using Microsoft services and tools. Following design science principles, we *explore and describe* the existing DevOps environment and identify main challenges on the borderline between operations and development, using qualitative data collected through interviews and observations. To address the identified challenges, we *design* a solution for more effective processing of data available through the monitoring system in operations by introducing a smart filter in the feedback loop. Thus our research adds to the new research and innovation discipline called AIOps, artificial intelligence for IT operations [8]. Moreover, we present a *prototype implementation and validation* of the proposed design. It includes a description of the labeling process of unlabeled operations data, using unsupervised anomaly detection and considering the service vulnerabilities, as well as learning new advanced alert rules using a supervised, decision tree-based Python module.

The contributions of our paper are threefold:

C1. **Problem conceptualization.** We identified alert *targeting*, signal to noise *optimization*, and system *interoperability* as being three important problem instances of the general alert flooding problem in the feedback from operations to development.

C2. **Solution design.** We present a unique technical solution that combines various systems' and applications' metrics for learning advanced alert rules within the new element in the feedback loop, a smart filter.

C3. **Prototype implementation.** We performed a pilot implementation of the proposed solution in the case environment as a proof of concept for further work.

The rest of the paper is structured as follows. We first present the "Background and Related Work" in this field. Next we elaborate the "Research Approach" followed by a "Case Description" of the case company. Identified problem instances are introduced in section "Problem Conceptualization". The solution proposal is presented in section "Solution Design", followed by "Prototype Implementation and Empirical Validation". Finally, we conclude with section "Discussion and Conclusion".

## Background and Related Work

Ståhl et al. [2] conclude in their systematic mapping study on continuous practices and DevOps, that the concepts of continuous software engineering practices and DevOps are ambiguous in the literature. We adhere to their proposed definition that "**Continuous deployment** is an *operations practice* where release candidates evaluated in continuous delivery are frequently and rapidly placed in a production environment". In contrast, "**Continuous release** is a *business practice* where release candidates evaluated in continuous delivery are frequently and rapidly made generally available to users/customers." Depending on the environment, a release may be achieved through deployment, for example in most SaaS (Software as a Service) environments. On the contrary, for user installed software, continuous deployment is not an applicable concept as the user must take actions to install a new version. However, continuous releases may still be offered to the users.

Ståhl et al. [2] find DevOps be a broader term, including culture and mindset. It also comprises tools, processes, and practices. We adhere to this broad definition of DevOps, as we want to investigate "the interplay between specific continuous practices and DevOps principles, processes and methods" [2], which aligns well with Fitzgerald and Stol's scoping of continuous software engineering [4].

Despite the observed ambiguity, there are additional research summaries. Laukkanen et al. [9] presented a literature review of problems, causes and solutions, when adopting continuous delivery. They build on a previous literature review by Rodriguez et al. [10], and summarize topics

related to build design, system design, integration, testing, release, human and organizations, and resources. However, the operational aspects are not included. Similarly, Shahin et al. [11] do not cover practices beyond continuous deployment in their review and Mishra and Otaiwi [12] only briefly mention operational feedback as contributing to software quality in DevOps, in their systematic mapping study.

There is, however, research related to post-deployment activities. Suonsyrjä et al. [13] studied how automatically collected data from operations could be used as feedback to the development. They reviewed the literature and surveyed practitioners' interest in such activities. They conclude that topics related to post-deployment monitoring appeared in the scientific literature during the 20$^{th}$ century but, not during the last two decades [13]. As an exception, Orso et al. [14] presented the GAMMA system 2002, as an approach to support monitoring software's behavior during its lifetime.

Monitoring is not only focused on the software. According to Pietrantuono et al. [15], monitoring of the software product in operation can be used for collecting usage data. The data is afterward analyzed and reused for selecting the most representative test cases, based on usage profiles, which are used in their approach to "continuous software reliability testing".

Moreover, monitoring has also been part of alarm systems used for triggering warning signals in case of unusual rises in systems' metrics. Xu et al. [16] proposed a Process-Oriented Dependability (POD)-Monitor for reducing a number of false alarms focusing on sporadic and infrequent operations. Their approach utilizes process-context information and the Support Vector Machines (SVM) algorithm for learning when to suppress alarms and reduce the overload on operators.

*Alerts* is another term used for denoting the same or similar events as *alarms* and according to Zhao et al. [6], they represent a key source of anomalous events in operations. Zhao et al. [6] reported an approach for handling alert storms consisting of alert storm detection using Extreme Value Theory (EVT), alert filtering using ML Isolation Forest method, alert clustering using Similarity Matrix Construction, and representative alert selection. Furthermore, Zhao et al. [17] published another study on enhancing the quality of services by utilizing the monitoring data. Similarly, they analyzed alerts but with aim of identifying the severity level. They proposed a framework AlertRank for extracting severe alerts based on textual and temporal alert features as well as features extracted from monitoring metrics. Since there are two different terms in the literature, in the rest of the paper we use *alerts* to denote signals of unexpected systems' behaviors in operations.

Monitoring in operations can be utilized even without alert rules, thus considering raw operations data. Cito et al. [18] identified three main categories of operations data: system metrics, application metrics, and application system metrics [18]. Recently, researchers and practitioners have devoted significant effort to the analysis of aforementioned operations data considering, among others, machine learning techniques and to the development of various applications. Anomaly detection is one of the available applications for early detection of a system's abnormal behavior. It has been used for detecting deviations in software releases based on the data generated by a DevOps toolchain [19]. Further, Du et al. [20] presented DeepLog, a model based on deep learning for natural language processing, which is used for learning patterns in logs and detecting anomalies in log data. More thorough research on anomaly detection has been undertaken by He et al. [21] where they provide an overview of supervised and unsupervised machine learning techniques used for log analysis. In addition, logs have been studied for several other applications. Clustering log sequences into groups, identifying causal dependencies, and creating failure rules are the main steps in the root cause analysis and failure prediction approach proposed by Fu et al. [22].

More attempts at problem identification by log analysis can be found in papers by He et al. [23] and Lin et al. [24] where KPI (Key Performance Indicators) are used in a combination with logs. In both papers, the authors deal with clustering-based techniques, but their solutions differ in the second phase of the proposed approaches. In the solution by He et al. [23], the second phase consists of correlation analysis of identified clusters with system KPIs, while the second phase by Lin et al. [24] includes extracting most representative logs from clusters and comparison of clusters created in test and production environment for simpler problem identification. Furthermore, feedback from operations has been used for decision making and improving feature planning [18] as well as for feedback-driven development where monitoring data has been used for improving developer's tools [25].

In summary, operations data has been studied and analyzed for different purposes but still, there is more to be explored in DevOps contexts, to improve the feedback from operations to development. State of the art solutions [6, 19, 26] address relevant challenges in managing operations data. However, situations of *alert* flooding in DevOps environments are not extensively explored. Thus, we aim to contribute to the design of solutions that better manage alerts in DevOps.

## Research Approach

Our study, as shown in Fig. 1, is a problem-driven design science approach [7]. Thus our starting point was to gain deeper insights into the specific challenges of our case company. As a first step, we explored how the general problem, of incorporating feedback from operations in the development, manifests as a *problem instance* in the industrial context

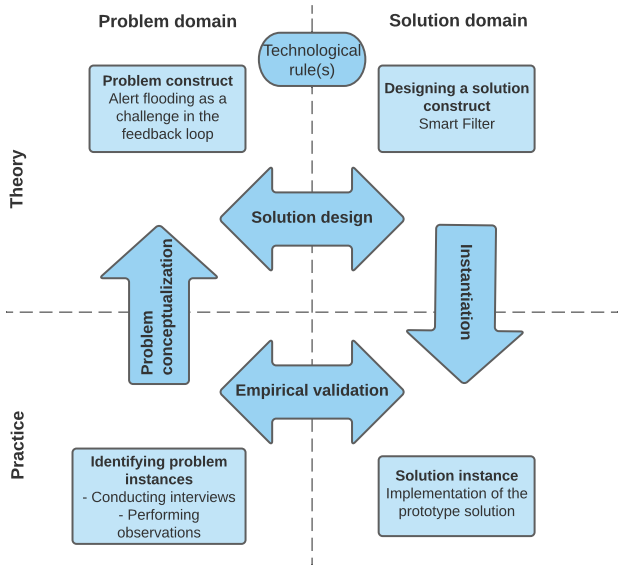



**Fig. 1** Overview of the Design Science Approach

under study. For that purpose, we conducted six interviews and performed observations in the case company to identify and articulate the main problem instances on which to focus further improvements.

To obtain a comprehensive overview of the issues, we selected interviewees in senior positions with different responsibilities within the team including a product owner, a test manager, a test developer, a system architect, and two developers. During the interviews, we asked general as well as more specific questions related to the DevOps cycle. The interviews were semi-structured since we wanted to flexibly explore the interviewee's opinions and let them speak about their main issues. Focus areas and examples of questions used in the interviews are shown in Table 1. All collected qualitative data, notes and video records, was analyzed using the NVivo tool. Furthermore, we observed their processes in operations and the way they were handling operations data. This enabled uncovering insights and defining problem instances.

In the *problem conceptualization* step, we described three identified problem instances (Section 5) through the lens of envisioned matching solutions, i.e. we formulated three high level technological rules. However, in this paper, we refined only one of them in the conceptual *solution design*. Hence, we improve the feedback loop from operations to development by introducing a new element, a smart filter, for optimization of alert to noise ratio. In the design process, we considered the insights gained through interviews, results of the intensive discussions with the development team, and state of the art solutions for alert management [6, 17].

Moreover, alongside the proposed solution design, we implemented a prototype *instance* to get a better understanding of the opportunities of the available operations data, its type and characteristics as well as the constraints of the context. In the implementation of the prototype solution, we used unsupervised anomaly detection throughout the labeling process of unlabeled operations data while also considering the service vulnerability and observed metrics frequency. Further, for generating new advanced alert rules, a supervised tree-based machine learning technique was used. Regarding the *empirical validation*, there were time and environment constraints that hindered a full evaluation of the implemented solution. However, we were able to perform a partial evaluation using limited data set for implementation of the multivariate anomaly detection in a prototype environment. In this way, we were able to compare the results obtained by using the smart filter in the feedback loop with the results of using the pure unsupervised ML technique for predicting alerts based on multivariate unlabeled data set.

## Case Description

The system under study is a backend system of an application for ticketing and payments used in public transportation. It is a cloud-based system developed and operated in a DevOps environment, using Microsoft tools and services. The system architecture is leaning towards a microservice

**Table 1** Topic areas and examples of questions used in the semi-structured interviews

| Focus area | Examples of questions |
|---|---|
| CI/CD pipeline | - Could you describe the CI/CD pipeline?<br>- What are the shortcomings and how can they be addressed? |
| Continuous monitoring | - Which parts of the system are monitored?<br>- Which signals are the most critical and good candidates for monitoring? |
| Alerts | - How does the current alert system look like?<br>- In which periods you experience the highest number of alerts? |
| Accessibility of operations data | - Which types of operations data are available for analysis?<br>- Which types of operations data are used for setting the alert rules? |
| Potential improvements | - How/what would you improve in your current monitoring system? |

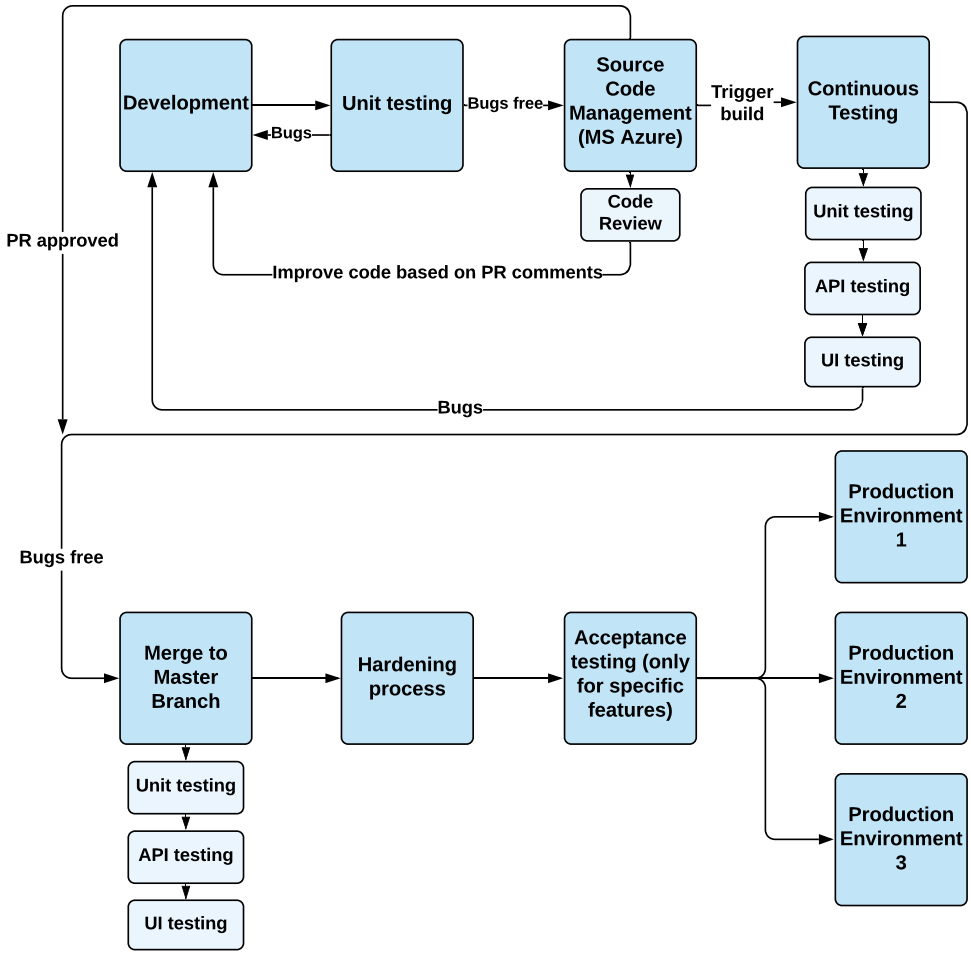



**Fig. 2** CI/CD pipeline

architecture which consists of 20 services that are highly maintainable, testable, and independently deployable.

Throughout the entire CI/CD cycle, shown in Fig. 2, new features or updates of each service are tested on: (1) unit level, every time the build process of the system under test with its dependencies is triggered; (2) API and UI level, every time the master branch is updated as well as every night on the latest build version from the master branch. Moreover, the candidate version for the release is used as a reference version by other teams in the company for a week, which is called the "hardening process". If necessary, the latest version is tested in the acceptance-test environment which serves as a production-like environment. The release cycle is weekly and ends by deploying to three production environments. Hence, the existence of several independent environments enables smooth development, testing, and deployment activities but also multiplies the complexity of the entire system.

The health status of each service is monitored using the Microsoft data platform, Azure Monitor. Azure Monitor collects the data from several sources such as applications or Azure resources into a common platform to be used for analysis, alerting, and visualization. Within this data platform, two types of data are available, metrics and logs. Metrics are numerical values denoting specific system's observations captured within a defined timestamp. Logs are represented by both, numerical and textual values and they describe specific events that happened at a particular moment in time. Both metrics and logs can be used for setting alert rules that signalize that something unexpected is detected in the observations of the targeted resources. The case company has implemented simple rules for detecting failed requests with error 500 and unexpected raises of dependency calls and failed Http requests, as shown in Table 2. When these rules are satisfied, then alerts are triggered and alert notifications are sent either to a dedicated Slack channel or via email.

Operations data shown in Table 2, represent only a small portion of all available data in Azure Monitor but in this paper, we focus on the selected logs and metrics. Among all accessible observations of different system components, we chose metrics and logs related to the data types used for setting current alert rules and the ones used in debugging in case of detected anomalies. Alert rules, shown in Table 2, are configured for all 20 services, and notifications about raised alerts are sent on two different platforms. Alerts that

**Table 2** Types of operations data mapped with configured alerts

| Operations data | | Configured alerts |
|---|---|---|
| ***Logs*** | Exceptions | / |
| | Traces | / |
| | Requests | / |
| ***Application metrics*** | Dependency failures | An unusual rise in the rate of dependency failures |
| | Exceptions | / |
| | Failed Requests | / |
| | Server Exceptions | / |
| ***System Metrics*** | CPU Time | / |
| | Errors Http 4xx | An unusual rise in the rate of failed Http requests |
| | Server Errors 5xx | Whenever there is a server error 500 |
| | Response Time | / |
| | Requests | / |

detect internal server error 500 are sent to the Slack channel, while unusual rises in the rate of dependency failures and failed requests are sent via email.

The development team has already reported various challenges in managing and responding to fired alerts with this configuration. Moreover, their every day development tasks are filled with the uncertainty that every alert brings into their development environment due to overload of non relevant alerts. Consequently, this might cause a bottleneck in the information flow from operations to development. The flaws, identified within the monitoring and alert system, are elaborated in the next section.

## Problem Conceptualization

In this section, we present three main problem instances, identified in the problem conceptualization step, with respect to the general goal of better incorporating feedback from operations into development. Based on observations made in the case company, *alert flooding* is identified as the main cause of all three problems. Alert flooding is a phenomenon that appears in a case of a high number of alerts that are not properly managed. In this paper, we focus on the specific aspects of this phenomenon namely, *targeting, optimization*, and *interoperability* problems.

### Alert Flooding as Targeting Problem

The first problem is defined as a targeting problem. This means that the distribution of alerts to target recipients, between the teams and individual assignment of a single or group of alerts within the team, is not fully transparent. Moreover, a lot of time is spent on discussions on how to resolve alerts and who is going to take the responsibility. Currently, there are three teams that can be assigned when an alert is fired. Each team consists of four or five members, mainly developers, and every team is responsible for one of the domains which consist of multiple services. Alert notifications are sent to a dedicated Slack channel, but no one is tagged or directly assigned to the raised alerts. Individual responsibilities within the team are not clear and team members usually discuss specific alerts in the same Slack channel. Sometimes they tag each other and ask if that person has already looked into raised alerts. As acknowledgment, they usually write that they will look at it right away or later. If they agree that an action should be taken, a ticket is created and added to a backlog of the board in Azure DevOps. Hence, two different platforms for communicating alerts are used but the information is not synchronized.

While observing the team and their current practices, we noticed that some team members showed more interest than others in resolving alerts and that some look into alerts that are related only to services they are developing or they are familiar with. Consequently, there is an increasing number of alert notifications because no one takes full responsibility for looking into alerts that frequently appear every day. After talking to some team members, it was clear that they would like to see some structured way of alert management and assignment but they also pointed out that acting on every alert would take too much time since their main focus is development. Because of that, designing a solution for the targeting problem becomes even more challenging.

### Alert Flooding as Optimization Problem

The second problem instance represents an optimization problem, which addresses optimization of a signal to noise ratio. In this case, the signal consists of high priority alerts while the noise represents low priority alerts, which frequently appear every day. Hence, the main question is how to differentiate between alerts that cause failures and alerts that cause temporary glitches that don't affect the system's performance.

While observing the current practices in alert management, we noticed that all alert notifications come to the Slack channel with the same priority. Over time, developers learned which alerts are reoccurring occasionally, and they consider them as "normal alerts". Normal alerts are mostly caused by glitches in an external or internal service or represent a consequence of a failure related to the central service. The central service represents the heart of the system and all alerts related to this service have the highest priority. This priority is not specified as a part of an alert notification, but is something that developers know since they developed the system and they know how vulnerable each of the services is. "Normal alerts" are not normal since they signalize that something might be wrong in the specific service, but they are normal as they occur frequently, and the team got used to them. They also produce noise in the channel used for communicating alerts and because of that some critical things may pass unnoticed. The team raised concerns about this and agreed that addressing and solving this particular problem might help in faster and better response to other more important alerts. One more reason to do so is because they currently do not act upon normal alerts unless there is a high number of occurrences.

The majority of current alert rules aim at discovering internal server errors with error code 500 while a significantly higher number of logs still remain unexplored, Table 2. Hence, there is a need for adding more alert rules. However, the team decided to stick with the existing alert rules since the current ones are not successfully managed. Recently, the team reported that they missed over 20,000 failed Http requests with error code 400. They did not notice this anomaly because they were overwhelmed with other alert notifications but also due to the fact that they do not usually analyze logs or fix issues before they cause severe problems. Hence, designing new or redesigning existing alert rules to optimize the signal to noise ratio, is another challenge that they are facing while at the same time it is important that the number of non-relevant alerts is not increased and that the most critical alerts are prioritized.

### Interoperability Flaws Between Developed System and External Systems

Many large-scale software systems depend on external services developed by third parties. In this way, the original system can offer more features to their end customers. This seems to be a huge benefit but may also increase the vulnerability of the entire system since even the smallest glitches in an external service might cause serious deviations in the original system. Similar issues are experienced in the case company as their backend system also depends on external payment providers, Azure databases, and other software projects developed in their company. There is a special Slack channel where RSS (Really Simple Syndication) feeds and emails from external services are forwarded. However, many problems are still discovered through customer service and user complaints. So, they get notified when something has already failed and is visible to end-users instead of in advance. Moreover, the uncertainty of potential disruptions makes developers even more confused. It is their responsibility to decide if a raised issue is something temporary or it really represents an issue they should look into and report. They usually make a decision based on the alert frequency and side effect appearance. There are no statistics that can prove developers' claims, but a huge number of alerts are caused due to interoperability flaws with external services. The existence of failed Http responses with unknown and unexpected error codes complicates root cause analysis even more. It is important to address this problem, otherwise the system stability will be degraded.

## Solution Design

As stated in Section 3, we provide a conceptual design for the second problem instance, *alert flooding as an optimization problem*. This problem causes the highest information overflow in the feedback loop. By addressing this specific instance, the scope of the first and the third problem instances will be reduced, and individual solutions simplified. The first and the third problem instances will not be individually treated in this paper but will be considered in our future work.

Hence, we propose one solution design and focus on the following challenges related to the second problem instance: (1) reduce the number of noisy alerts without missing the critical ones; (2) increase the number of alert rules without causing an overload of alert notifications; (3) improve developer's responses to the fired alerts while minimizing interference with their development related tasks. Accordingly, we present the overview of the proposed solution for the second problem instance in Fig. 3.

The upper part of Fig. 3, illustrates the previously explained architecture of the software system, consisting of 20 micro services and Azure Monitor, that monitors real-time application performance (Application Insights) and performance of Http-based services for hosting applications (App Services). The lower part of Fig. 3, visualizes the enhanced alert system with a new addition, representing the bridge between MS Azure Monitor and Slack, the platform where alert notifications are sent. The new box, the smart filter, serves as a middle-ware and provides additional features to the existing alert management.

The main task of the introduced box is to generate alert rules for sending alert notifications to the messaging

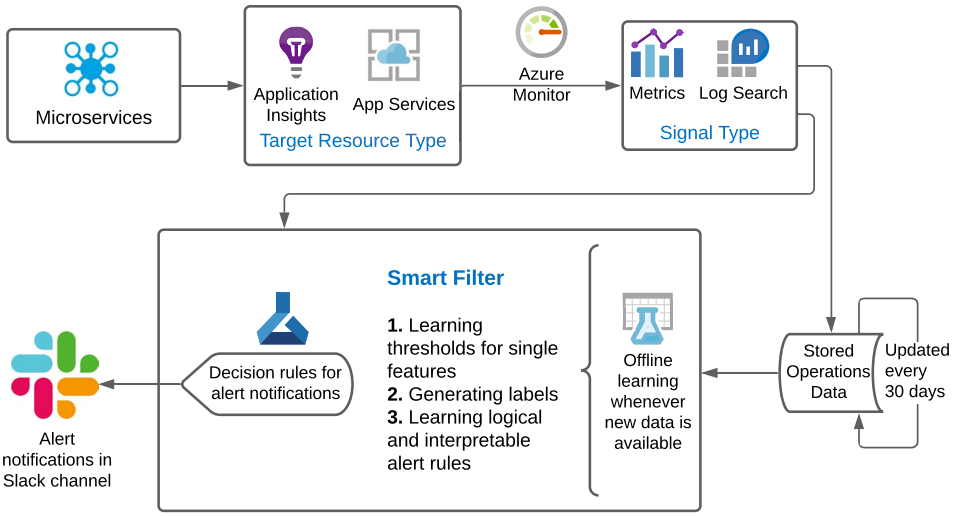



**Fig. 3** Overview of the proposed solution for the second problem instance

**Table 3** Overview of the selected data, service vulnerabilities and desired decision rules

| | |
|---|---|
| **Selected application and system metrics** | - CPU Time<br>- Number of failed requests<br>- Number of exceptions<br>- Number of dependency failures<br>- Http 4xx errors<br>- Internal server errors<br>- Total number of requests<br>- Response time |
| **Services with known vulnerabilities** | - Service B –> buying tickets on vending machines<br>- Service G –> service for validating selected locations<br>- Service M –> main service for ticketing<br>- Service P –> bridge to an external payment service |
| **Example of a decision rule** | IF num_of_failed_requests_SG > threshold_1 AND response_time_SB > threshold_2 AND num_of_Http500_SB > threshold_3 THEN send_notification |

platform. Hence, we temporally disregard current alert notifications and instead focus directly on the most important data, specifically metrics shown in Table 3, holding information about the system's performance. The reason for such an approach is that the current alert rules only catch a limited number of system glitches and failures while at the same time not being able to differentiate noisy alerts from important ones. The smart filter will analyze more data and learn over time to identify new dependencies that may generate new and better decision rules. In this way, we will reduce the risk of omitting important alert notifications while keeping the the Slack channel clean from noisy information. Therefore, in our proposed solution design, new decision rules are learnt based on the features representing the systems' and applications' performance metrics of the mostly affected services. The output of the smart filter is binary, meaning that new decision rules are able to determine when to send and when not to send alert notifications. As shown in Fig. 3, the smart filter involves preprocessing and labeling of the data required for the learning process. The exact procedure is presented in Section 7.

All things considered, the proposed approach of generating new decision rules aims at filtering the incoming performance data and sending only relevant alert notifications to the Slack channel. Newly learnt alert rules should not increase the number of alert notifications in the Slack channel since the learning process also involves learning about the noisy data.

Therefore, the proposed solution design addresses the aforementioned challenge regarding the insufficient alert rules. The purpose of the enhanced alert management is to provide more insights into correlations between alerts and operations data and at the same time enable forwarding more details about potential failures within the alert notifications. In this way, the development team could have all information needed to discover the root causes of potential failures. Moreover, it is expected that developer's awareness of raised alerts will increase and that they will need less time

for resolving critical systems behaviors. Therefore, the proposed solution design intends to resolve the previously listed challenges related to the second problem instance.

## Prototype Implementation and Empirical Validation

In this section, we present technical details of the prototype implementation[1] as well as the effects of the implemented solution prototype in the identified problem context. Prototype implementation includes *data selection*, *tools and methods selection*, *threshold detection for each of the features*, *labeling process*, *training process* and *testing*. While working on the implementation of a solution prototype, we have decided to stick with basic machine learning techniques since we primarily wanted to examine the limitations of the suggested design. Hence, using deep learning or reinforcement learning for identified problem instances is beyond the scope of this paper.

**Data selection.** For the prototype implementation, we have chosen to only work with numerical values representing the various systems' and applications' performance metrics, to keep the simplicity. Logs are not included in the preliminary data selection due to their complex structure and due to the fact that the observed logs including traces, types of exceptions, or failed requests could only help with the explainability of potential failures. The metrics and services selected to be part of the training data (see Table 3) are chosen based on the observations made in the messaging and monitoring platform focusing on metrics frequency and service vulnerability. Therefore, we selected 8 metrics for each of the 11 services, which makes in total 88 features. Every feature vector has 8623 samples collected during a period of one month with a time granularity of 5 min, which was selected based on the current practice within the project.

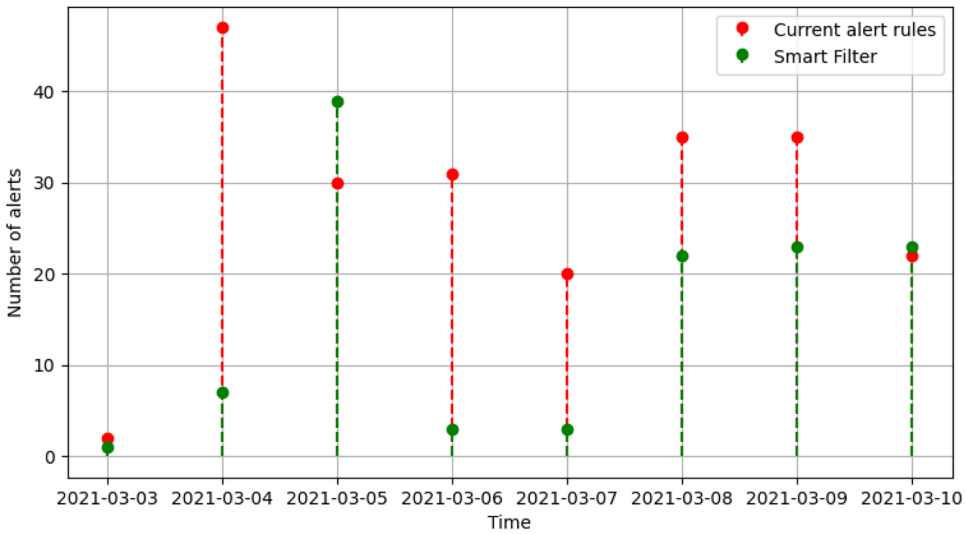
**Tools and method selection.** The presented solution design involves learning new decision rules in the form of logical expressions "IF conditions THEN response" and for such an approach the first choice of ML methods are tree based methods, such as bagging and random forest. Therefore, for implementation, we use Skope-rules [27], a Python machine learning module for extracting rules from the tree ensemble as suggested by Friedman and Popescu [28]. The classification is binary, thus, if an instance representing the combination of multiple features satisfies conditions of the rule, then it is assigned to one of two output classes, "send_notification" or "dont_send_notification". Using this Python module requires labeled data for the learning process, thus making this approach even more challenging since the monitoring data platform collects only raw data and the knowledge about the expected outcomes is unknown.

[1] https://github.com/adha7/smart-alert-filter, available upon request

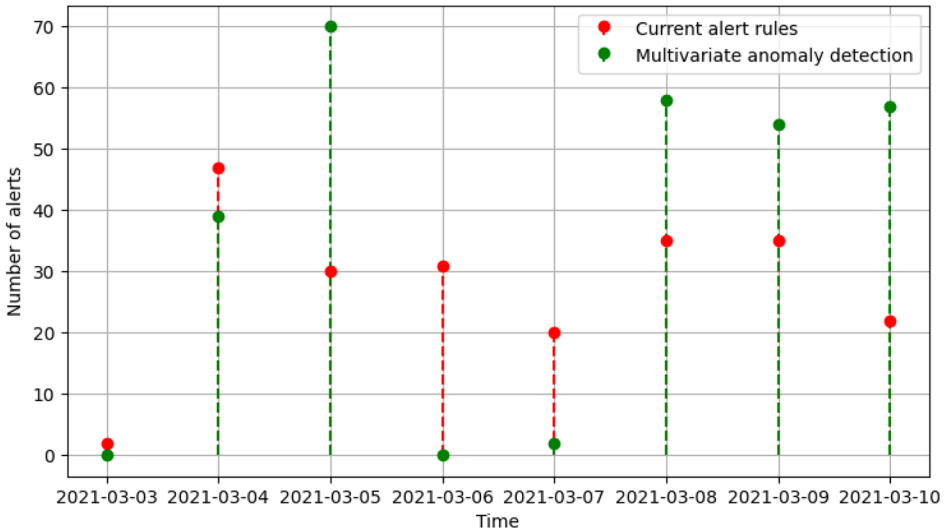
**Identifying thresholds.** Therefore, we decided to generate labels based on the known service vulnerabilities and desired level of contamination. The first step of the labeling process is to identify thresholds for single features using machine learning for anomaly detection (see Fig. 3, step 1). For that purpose, we used a Python toolkit PyOD [29] consisting of 30 different detection algorithms. Hence, the thresholds are predicted for each of the 88 features where the outliers are expected to be extremely high values. By applying one of the algorithms from the PyOD module on a feature vector, we get anomaly scores for each of the values within a feature vector. Larger anomaly scores are assigned to outliers and the threshold is simply determined by picking a value from a sorted feature vector with a large enough score. The score value on the borderline between inliners and outliers is chosen so that the level of contamination of the entire training data equals 0.05. The contamination is determined by the number of outlying objects in the data set, in our case alert notifications that need to be sent to the messaging platform. Selected level of contamination corresponds to the 13 alert notifications per day and represents three times less of the current number of alert notifications. Since there is no optimal number of alert notifications per day we consider this decrease significant and at the same time large enough to not miss the important system failures.

**Labeling process.** After determining the thresholds for each of the features, the warnings are raised in the cases where the features reach values above these border values. Based on these warnings, we generate labels (see Fig. 3, step 2) considering a fixed number of raised warnings in a time slot of 5 minutes as well as capturing for which services warnings are raised, targeting services shown in Table 3. Accordingly, the output class is labeled as 1, if there are more than 8 raised warnings in the same time slot, which means that there are at least two services affected considering that 8 warnings can be related to one service. Further, the output is also denoted as anomalous or 1, if there are warnings raised for the most vulnerable services, as shown in Table 3, no matter the number of raised warnings. When the labeling process is completed, learning logical and interpretable alert rules can be activated (see Fig. 3, step 3).

**Training process.** Through the training process, Skope-rules generated 120 rules for the class "dont_send_notification" and 43 rules for the class "send_notification". The rules are generated by fitting single estimators, decision trees, with predefined precision and recall as input parameters. The precision and recall reached during the training phase are between 0.92 and 0.99 for the output class "dont_send_notification". The precision score for the output class "send_notification" is evenly high as for the opposite class but the recall was significantly lower due to very low contamination, the number of outliers, in the training data set. A low recall score makes the algorithm "picky" when




**(a)** Smart filter




**(b)** Multivariate anomaly detection

**Fig. 4** Number of alerts per day in the test data. RED color: alerts raised with current alert rules; GREEN color: alerts raised with **(a)** the smart filter and **(b)** multivariate anomaly detection

selecting outlying samples which might be good for filtering the noise but on the other hand, it might miss single and isolated outliers.

**Testing.** On this account, we analyze how the implemented prototype scales the number of predicted alert notifications per day to the actual number of raised alerts. We use test data collected within the 7 days (March 3, 20:35 – March 10, 19:40) for predicting outlying objects, alerts, and present the results in Fig. 4.

We conclude that the smart filter produces half the number of alerts in a period of 7 days, 108 compared to 211. Regarding the distribution of alert notifications per day, the number of predicted alerts during the weekend (March 6 and 7) is very low which is expected due to lower stress on the ticketing and payments system. During the workdays, the number of predicted alerts is less than actual except when there are issues in the system that the current alert system is not able to capture. This was the case on March 5, when there was a problem with buying tickets on the vending machines. The smart filter raised an alert 30 minutes earlier than it was reported by customers, which means that this specific failure could have been caught before it was noticed by users.

The implemented prototype reduces the overall overload on the development team but also gives space for further improvement by introducing prioritization of alerts and sending the alerts on different Slack channels based on their priority for even better and clearer differentiation.

**Empirical validation.** In addition to the smart filter implementation, we also implemented multivariate anomaly detection (MAD) to validate our prototype by comparing it with the pure unsupervised ML technique for detecting outliers, representing alerts, in multivariate unlabeled data set. We used the same Python toolkit PyOD [29] for the MAD implementation and selected the COPOD model, copula-based outlier detection introduced by Li at al. [30]. The COPOD model was trained using the same training data but without labels. The predictions, shown in Fig. 4b, using the same test data set, revealed that the MAD trained model does not scale very well the number of predicted alerts. It predicts almost the same number of alerts as the actual alert system, making the same level of noise. Both models, trained using the smart filter and MAD respectively, reach the F1-score, a harmonic mean of precision and recall denoting a model's accuracy, above 0.9. However, the pure unsupervised ML might not be able to capture the imbalance between the target classes and the importance of specific services and their metrics. To clarify this, we look at the alert distribution over the metrics of highly affected services shown in Fig. 5a, b. We noticed that the smart filter produces less noise around the actual failures, such as the one marked with the black arrow from March 5. This means that the actual failure can be more easily identified among the alerts that appear close to the selected alert on the graph. The predicted alerts using multivariate anomaly detection are grouped and based on the graph, they produce several alert floods which is the opposite to what we want to achieve. On the other hand, the smart filter predicts isolated alerts in case of short system's glitches and smaller groups of alerts when there is a larger issue rolling out.

There are still some individual events that passed unnoticed but since this is only a prototype version, imperfections and shortcomings are expected. Furthermore, we used a limited data set collected within one month, which could have also affected the training process and learning when to send alert notifications due to a low number of outlying objects. We aim to address this in our future work by considering the larger data set.

## Discussion and Conclusion

The synergy between development and operations in DevOps is important for developing and releasing high-quality software systems, but even more for gaining insights into the system's behavior in the production environment. To ensure the latter, raw operations data, collected through runtime monitoring tools, is analyzed to discover valuable feedback information. Our results have shown that monitoring and utilizing data available in the production may help developer teams to more easily identify, understand and communicate issues in the operations. Further, it helps present the valuable information in an actionable manner and reduces the pressure and overload.

The results obtained, following design science principles, directly relate to three main contributions mentioned in the introduction section, problem conceptualization (C1), solution design (C2), and prototype implementation (C3). We started with the problem conceptualization since the first step in solving a particular problem is understanding its causes and effects. Before our attempt to identify the main challenges on the borderline between development and operations, the everyday routine work at the case company obscured shortcomings in the information flow between operations and development. During the initial stage of interviews and observations, we managed to identify *targeting, optimization* and *interoperability* problem instances related to *alert flooding*. The problem conceptualization (C1) helped both the development team in acknowledging existing issues and the research team, in creating a solution design, which is our second contribution. After presenting our findings, the development team seemed relieved since they finally understood what was hindering them from making full use of operational data and how data overload in operations could be prevented.

The solution design (C2), as previously mentioned, addresses the problem of alert flooding with the emphasis on reducing the number of noisy alerts. The presented conceptual model includes a new element in the feedback loop, responsible for learning new advanced alert rules capable of reducing the total number of alerts and increasing their relevance. The smart filter addresses challenges in the alert management such as insufficient number of alert rules, noisy alert notifications, and slow developer's response on fired alerts. Therefore, this addition in the feedback loop improves the information flow from operations to development by introducing alert rules which combine various systems' and applications' metrics and services with the aim of capturing unexpected and faulty system's behaviors and providing more detailed insights to the development team.

The third contribution (C3) includes implementation of the solution prototype and validation in a specific context, i.e. our case, the ticketing and payment system operated in the DevOps environment. We successfully implemented a prototype version of the smart filter using a hybrid method consisting of unsupervised anomaly detection and supervised decision tree-based Python toolkit while also considering the importance of highly vulnerable services in the labeling process. The prototype was validated using a limited test data set collected through the monitoring system in the production environment. Accordingly, we demonstrated that a severe failure could have been caught if the smart filter was integrated in the feedback loop instead of the current alert system. Furthermore, we compared the implementation of

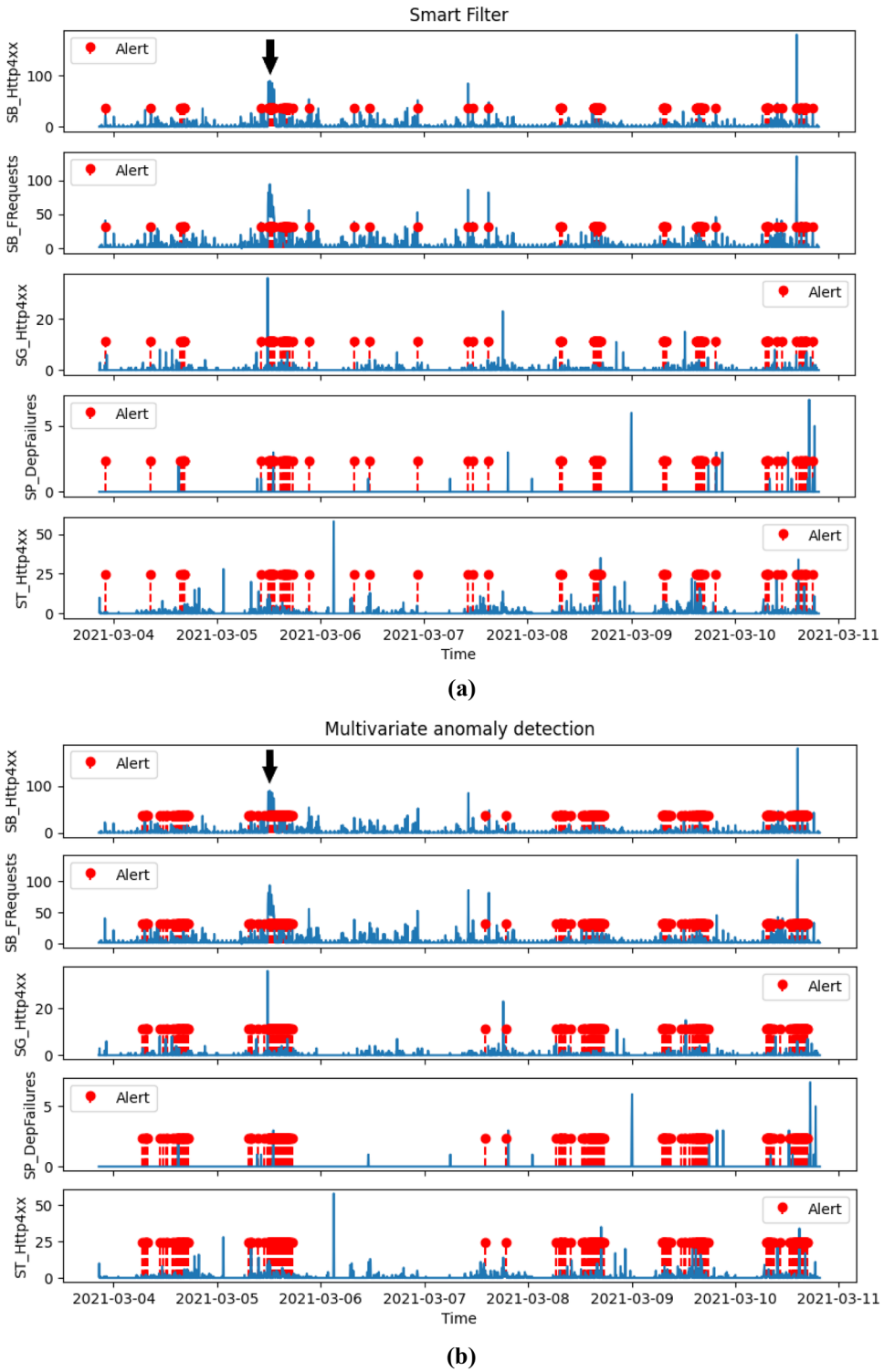



**Fig. 5** Distribution of raised alerts in the test data using **(a)** the smart filter and **(b)** multivariate anomaly detection. BLUE color: selected performance metric; RED color: raised alerts

our prototype with the pure unsupervised ML technique for multivariate anomaly detection. We showed that the customized hybrid method better captures the systems' unbalanced operations data and system-specific characteristics needed for catching both systems' glitches and severe failures. Hence, the feedback information obtained as a final result has tightened the connection between operations and development. There have been several attempts at addressing similar challenges using state of the art solutions based on deep learning [6, 20, 26], while our solution proposal reach promising results while keeping simplicity of the ML approach.

The smart filter in the feedback loop improves the connection between operations and development but at the same time raises more challenges that need to be addressed in the future. Even though it reduces the total number of alerts, it could still be improved by increasing the level of differentiation between the raised alerts by introducing several levels of priorities and target recipients. We plan for further work to address the raised challenges by considering deep learning and other machine learning techniques as well as implementing the smart filter in the production environment. Consequently, the smart filter will be fully integrated and automated in the feedback loop and will require minimum human assistance. In this way, we would be able to get immediate feedback and insights from developers involved in the alert management, which is needed for obtaining a complete evaluation of the smart filter. Moreover, since our study provides prescriptions for problems in a very specific industrial context, in the future we aim to validate our solution in other similar contexts.

**Acknowledgments** This work was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation. We thank the DevOps teams for their willingness to share insights and respond to our questions.

**Funding** Open access funding provided by Lund University. This study was partially supported by the Wallenberg Artificial Intelligence, Autonomous Systems and Software Program (WASP) funded by Knut and Alice Wallenberg Foundation. No grant number available.

## Compliance with Ethical Standards

**Conflict of interest** Adha Hrusto declares that she has no conflict of interest. Per Runeson declares that he has no conflict of interest. Emelie Engström declares that she has no conflict of interest.

**Ethical approval** All procedures performed in studies involving human participants were in accordance with the ethical standards of the institutional and/or national research committee and with the 1964 Helsinki declaration and its later amendments or comparable ethical standards. This article does not contain any studies with animals performed by any of the authors.

**Consent to participate** Informed consent was obtained from all individual participants included in the study.

**Consent to publish** The case company has consented to the submission of the case report to the journal.

**Code availability** The code is available upon request in a private GitHub repository, https://github.com/adha7/smart-alert-filter.



## References

1. William P, John S, Tony E, Andriy M. On challenges of cloud monitoring. In *Proceedings of the 27th Annual International Conference on Computer Science and Software Engineering*, CASCON '17, page 259–265, USA, 2017. IBM Corp.
2. Ståhl D, Mårtensson T, Bosch J. Continuous practices and devops: Beyond the buzz, what does it all mean? In *2017 43rd Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*, pages 440–448, Vienna, 2017. IEEE.
3. Capizzi A, Distefano S, Mazzara M. From DevOps to DevDataOps: Data management in devops processes. In Jean-Michel B, Manuel M, Bertrand M, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 52–62. Springer, New York, 2020.
4. Fitzgerald B, Stol K-J. Continuous software engineering: a roadmap and agenda. J Syst Softw. 2017;123:176–89.
5. Felderer M, Russo B, Auer F. On testing data-intensive software systems. In Stefan B, Matthias E, Arndt L, Edgar RW, editors, *Security and Quality in Cyber-Physical Systems Engineering, With Forewords by Robert M. Lee and Tom Gilb*, pages 129–148. Springer, 2019.
6. Zhao N, Chen J, Peng X, Wang H, Wu X, Zhang Y, Chen Z, Zheng X, Nie X, Wang G, Wu Y, Zhou F, Zhang W, Sui K, Pei D. Understanding and handling alert storm for online service systems. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 162–171, 2020.
7. Runeson P, Engström E, Storey M-A. The design science paradigm as a frame for empirical software engineering. In Michael F, Guilherme HT, editors, *Contemporary Empirical Methods in Software Engineering*, pages 127–147. Springer, 2020.
8. Dang Y, Lin Q, Huang P. AIOps: Real-World Challenges and Research Innovations. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 4–5, Montreal, QC, Canada, May 2019. IEEE.
9. Laukkanen E, Itkonen J, Lassenius C. Problems, causes and solutions when adopting continuous delivery—A systematic literature review. Inf Softw Technol. 2017;82:55–79.
10. Pilar R, Alireza H, Lucy EL, Susanna T, Tanja S, Juho E, Teemu K, Pasi K, June MV, Markku O. Continuous deployment of software intensive products and services: a systematic mapping study. J Syst Softw. 2017;123:263–91.
11. Mojtaba S, Muhammad AB, Liming Z. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. IEEE Access. 2017;5:3909–43.
12. Mishra A, Otaiwi Z. Devops and software quality: a systematic mapping. Comput Sci Rev. 2020;38:100308.
13. Suonsyrjä S, Hokkanen L, Terho H, Systä K, Mikkonen T. Post-deployment data: A recipe for satisfying knowledge needs in software development? In *IWSM-MENSURA*, pages 139–147. IEEE, 2016.
14. Alessandro O, Donglin L, Mary JH, Richard L. Gamma system: continuous evolution of software after deployment. SIGSOFT Softw Eng Notes. 2002;27(4):65–9.

15. Pietrantuono R, Bertolino A, De Angelis G, Miranda B, Russo S. Towards Continuous Software Reliability Testing in DevOps. In *2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST)*, pages 21–27, Montreal, QC, Canada, May 2019. IEEE.
16. Xu X, Zhu L, Fu M, Sun D, Binh Tran A, Rimba P, Dwarakanathan S, Bass L. Crying wolf and meaning it: Reducing false alarms in monitoring of sporadic operations through pod-monitor. *2015 IEEE/ACM 1st International Workshop on Complex Faults & Failures in Large Software Systems (COUFLESS)*, pages 69 – 75, 2015.
17. Zhao N, Jin P, Wang L, Yang X, Liu R, Zhang W, Sui K, Pei D. Automatically and Adaptively Identifying Severe Alerts for Online Service Systems. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 2420–2429, Toronto, ON, Canada, July 2020. IEEE.
18. Cito J, Wettinger J, Lwakatare LE, Borg M, Li F. Feedback from operations to software development–a DevOps perspective on runtime metrics and logs. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 184–195. Springer International Publishing, 2019.
19. Capizzi A, Distefano S, Araújo LJP, Mazzara M, Ahmad M, Bobrov E. Anomaly detection in DevOps Toolchain. In Jean-Michel Bruel, Manuel Mazzara, and Bertrand Meyer, editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*, pages 37–51. Springer International Publishing, 2020.
20. Du M, Li F, Zheng G, Srikumar V. DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1285–1298, Dallas Texas USA, October 2017. ACM.
21. He S, Zhu J, He P, Lyu MR. Experience Report: System Log Analysis for Anomaly Detection. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 207–218, Ottawa, ON, Canada, October 2016. IEEE.
22. Fu X, Ren R, McKee SA, Zhan J, Sun N. Digging deeper into cluster system logs for failure prediction and root cause diagnosis. In *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 103–112, Madrid, Spain, September 2014. IEEE.
23. He S, Lin Q, Lou J-G, Zhang H, Lyu MR, Zhang D. Identifying impactful service system problems via log analysis. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, pages 60–70, Lake Buena Vista, FL, USA, 2018. ACM Press.
24. Lin Q, Zhang H, Lou J-G, Zhang Y, Chen X. Log clustering based problem identification for online service systems. In *Proceedings of the 38th International Conference on Software Engineering Companion - ICSE '16*, pages 102–111, Austin, Texas, 2016. ACM Press.
25. Cito J, Leitner P, Gall HC, Dadashi A, Keller A, Roth A. 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!) - Onward! 2015*, pages 14–27, Pittsburgh, PA, USA, 2015. ACM Press.
26. Islam MS, Pourmajidi W, Zhang L, Steinbacher J, Erwin T, Miranskyy A. Anomaly detection in a large-scale cloud platform. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 150–159, 2021.
27. Gardin F, Gautier R, Goix N, Ndiaye B, Schertzer J-M. Machine learning with logical rules in Python. https://github.com/scikit-learn-contrib/skope-rules, 2020.
28. Friedman JH, Popescu BE. Predictive learning via rule ensembles. Ann Appl Stat. 2008;2(3):916–54.
29. Zhao Y, Nasrullah Z, Li Z. Pyod: A Python toolbox for scalable outlier detection. J Mach Learn Res. 2019;20(96):1–7.
30. Li Z, Zhao Y, Botta N, Ionescu C, Hu X. Copod: Copula-based outlier detection. In *2020 IEEE International Conference on Data Mining (ICDM)*, pages 1118–1123. IEEE, 09 2020.