Runtime Metric Meets Developer – Building Better Cloud Applications Using Feedback

Jürgen Cito, Philipp Leitner, Harald C. Gall
software evolution & architecture lab
University of Zurich
{cito, leitner, gall}@ifi.uzh.ch

Aryan Dadashi, Anne Keller, Andreas Roth
SAP SE
{aryan.dadashi, anne.keller01, andreas.roth}@sap.com

Abstract

A unifying theme of many ongoing trends in software engineering is a blurring of the boundaries between building and operating software products. In this paper, we explore what we consider to be the logical next step in this succession: integrating runtime monitoring data from production deployments of the software into the tools developers utilize in their daily workflows (i.e., IDEs) to enable tighter feedback loops. We refer to this notion as *feedback-driven development* (FDD).

This more abstract FDD concept can be instantiated in various ways, ranging from IDE plugins that implement feedback-driven refactoring and code optimization to plugins that predict performance and cost implications of code changes prior to even deploying the new version of the software. We demonstrate existing proof-of-concept realizations of these ideas and illustrate our vision of the future of FDD and cloud-based software development in general. Further, we discuss the major challenges that need to be solved before FDD can achieve mainstream adoption.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques; H.3.4 [*Systems and Software*]: Distributed Systems

Keywords Software Development, Cloud Computing, Continuous Delivery, Feedback-Driven Development

1. Introduction

With the widespread availability of broadband Internet, the software delivery process, and, as a consequence, industrial software engineering, has experienced a revolution. Instead of boxed software, users have become accustomed to software being delivered "as-a-Service" via the Web (SaaS [43]). By now, this trend spans various kinds of software, including enterprise applications (e.g., SAP SuccessFactors), office products (e.g., Windows Live), end-user applications (e.g., iCloud), or entire web-based operating systems (e.g., eyeOS). With SaaS, much faster release cycles have become a reality. We have gone from releases every few months or even years to weekly or daily releases. Many SaaS applications are even employing the notion of continuous delivery [20] (CD), where new features or bug fixes are rolled out immediately, without a defined release plan. In the most extreme cases, this can lead to multiple rollouts a day, as for instance claimed by Etsy, a platform for buying and selling hand-made crafts[1]. On the one hand, these circumstances have imposed new challenges to software development, such as the necessity not to postpone quality checks to a dedicated quality assurance phase, as well as necessitating a high degree of automation of the delivery process as well as cultural changes [10, 21]. On the other hand, SaaS and CD have also opened up tremendous new opportunities for software developers, such as to gradually rollout new features and evaluate new ideas quickly using controlled experiments in the production environment [25].

1.1 Feedback-Driven Development

In this paper, we focus on one particular new opportunity in SaaS application development: tightly integrating the collection and analysis of runtime monitoring data (or feedback) from production SaaS deployments into the tools that developers use to actually work on new versions of the application (i.e., into Integrated Development Environments, or IDEs). We refer to this notion as *feedback-driven development* (FDD). FDD includes, but goes way beyond, visualizing performance in the IDE. We consider FDD to be a logical next step in a long succession of advancements in software engineering that blur the traditional boundaries be-

[1] http://www.infoq.com/news/2014/03/etsy-deploy-50-times-a-day

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

tween building and operating software products (e.g., cloud computing [9], DevOps [21], or live programming [32]).

We argue that now is the right time for FDD. Firstly, driving software development through runtime feedback is highly *necessary*, given that the fast release cycles prevalent in SaaS and CD do not allow for long requirements engineering and quality assurance phases. Secondly, the necessary feedback data is now *available*. Most SaaS applications are run using the cloud deployment model, where computing resources are centrally provided on-demand [9]. This allows for central management and analysis of runtime data, often by using Application Performance Monitoring (APM) tools, such as New Relic[2]. However, currently, this runtime data coming from operations (*operations data*) is hardly integrated with the tooling and processes that software developers use in their daily work. Instead, operations data is usually available in external monitoring solutions, making it cumbersome for developers to look up required information. Further, these solutions are, in many cases, targeted at operations engineers, i.e., data is typically provided on system level only (e.g., CPU load of backend instances, throughput of the SaaS application as a whole) rather than associated to the individual software artifacts that developers care about (e.g., lines of code, classes, or change sets). Hence, operations data is available in SaaS projects, but it is not easily *actionable* for software developers.

1.2 Paper Contribution and Outline

In this paper, we discuss the basic idea behind the FDD paradigm based on two classes of FDD tools, namely *analytic* and *predictive* FDD. Analytic FDD tools bring runtime feedback directly into the IDE and associate performance data visually to the software artifacts that they relate to, hence making operations data actionable for software developers. For instance, this allows developers to refactor and optimize applications based on feedback on how the code actually behaves during usage.

Predictive FDD goes one step further, and warns developers about problems based on local code changes prior to even running the application in production. To this end, predictive FDD builds upon static code analysis [1, 13], but augments it with knowledge about runtime behavior. This allows us to generate powerful warnings and predictions, which would not be possible based on static analysis alone. Even more sophisticated predictive FDD tools are able to use knowledge about the concrete system load that various service calls induce to predict the impact of code changes on the cloud hosting costs of the application, warning the developer prior to deployment about changes that will make hosting the application in the cloud substantially more costly.

In Section 2 we give a brief exposition of relevant background, which we follow up by an illustration of an example application that can benefit from FDD in Section 3. We introduce the general concepts in Section 4, and, in Section 5, substantiate the discussion using three concrete case studies of analytic as well as predictive FDD tools, which we have devised and implemented as part of the European research project CloudWave[3]. Further, we elaborate what major challenges remain that might impede the wide-spread adoption of feedback usage in everyday development projects in Section 6. Finally, we discuss related research work in Section 7, and conclude the paper in Section 8.

2. Background

The overall vision of FDD is tightly coupled to a number of other recent advances in Web engineering, some of which we have already sketched in Section 1. We now provide a more detailed background, to allow the reader to understand what kind of applications we assume will be supported by FDD going forward. Specifically, three current trends (cloud computing, SaaS, and continuous delivery) form the basis of FDD.

2.1 SaaS and Cloud Computing

The concept "cloud computing" is famously lacking a crisp and agreed-upon definition. In this paper, we understand "cloud" applications to mean applications that are provided as a service over the Web (i.e., SaaS in the NIST model of cloud computing [34]), in contrast to applications that are licensed and installed on premise of a customer's site, or downloaded and run on the user's own desktop or mobile device. Figure 1 illustrates these different models.

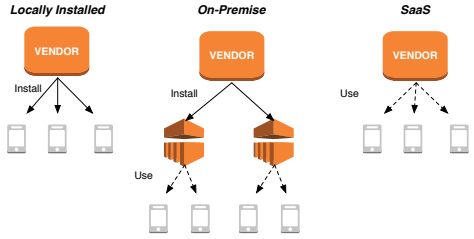

Figure 1. SaaS in contrast to on-premise or on-device software provisioning models. In SaaS, only one instance of the application exists and is accessed by clients directly. The software is never installed outside of the vendor's control.

SaaS has some interesting implications for the evolution and maintenance of applications. Most importantly, there is exactly one running instance of any one SaaS application, which is hosted by and under control of the service provider. This single instance serves all customers at the same time (multi-tenancy [3]). Implementing and rolling out new features in a SaaS environment is at the same time promising (as rollouts are entirely under the control of the service provider) and challenging (as every rollout potentially impacts each customer) [4]. Further, the SaaS model gives

[2] `http://newrelic.com`

[3] http://cloudwave-fp7.eu

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

service providers ready access to a rich set of live performance and usage data, including, for instance, clickstream data, fault logs, accurate production performance metrics, or even production user data (e.g., uploaded videos, number and type of Skype contacts). In addition to supporting traditional operations tasks (e.g., application performance engineering), this abundance of data has also led to the ongoing "big data" hype [5], which promises to generate deep market insight based on production data. However, these analyses are primarily on a strategic level (e.g., which product features to prioritize, which markets to address). Whether, and how, cloud feedback can also be used to support software developers in their daily workflow, i.e., while they are writing or optimizing code, is a much less discussed topic.

2.2 Continuous Delivery

Another recent advance that is tightly coupled to the changed evolution of SaaS applications is continuous delivery (CD). CD has recently gained steam due to the success of companies such as Facebook, Google, or Etsy, all of which claim to employ CD to varying degrees for their core services. The most significant conceptual novelty behind CD is the abolishment of the traditional notion of releases. Instead, each change to the SaaS application may be shipped (i.e., pushed to production) independently of any release cycles "as soon as it is ready" [20].

In other release models (e.g., release trains [24], as used by the Firefox project), new features are rolled out according to a defined release plan. If a new feature is not ready in time for a feature cut-off date, it is delayed to the next release, which may in the worst case take months. At companies like Etsy, features are rolled out as soon as they are deemed ready, independently of a defined release plan. Facebook claims to occupy a middle ground between release trains and strict CD, where most features are rolled out into production the same day they are ready, while a fraction of particularly sensitive changes (e.g., those related to privacy) are rolled out once a week [14]. In practice, these models result in frequent, but tiny, changes to the production environment, in the most extreme cases multiple times a day. This practice increases both, the business agility of the application provider, as well as the likelihood of releasing badly-performing changes to production [42]. A consequence is that SaaS applications with such release models tend to be in a state of perpetual development [14] – there is never a stable, released and tagged version which is not supposed to be touched by developers.

3. Illustrative Example

In the rest of this paper, we base our discussions on a (fictitious) enterprise application *MicroSRM*. *MicroSRM* is based on a microservice architecture [37], of which a small excerpt is shown in Figure 2a. *MicroSRM* consists of a purchase order service (*PO*) and a supplier service (*Supplier*). The application allows companies to manage its purchases and suppliers, such that its business users can create, modify or delete purchase orders, or view individual purchase orders including all details, such as order items. Following the notion of microservices, each of those services consists of a frontend, which acts both as a load balancer and API for the clients of the service, a data storage, and a number of backend instances, which implement the main business logic of the service. The concrete number of backend instances can be adjusted dynamically, based on load and following the idea of auto-scaling [30]. This is depicted in Figure 2(b).

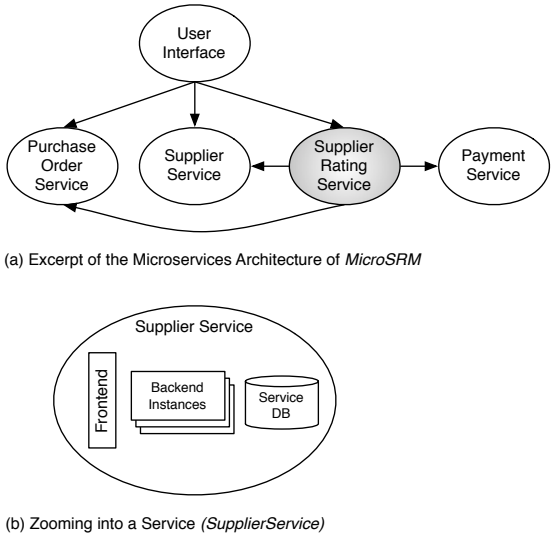

Figure 2. Architecture overview of *MicroSRM* consisting of a network of Microservices.

Let us now assume that *MicroSRM* has been running successfully for a while, and the application is supposed to be extended with an additional service, a supplier rating service (*Rating*). The new service utilizes information accessed through the APIs of both, *PO* and *Supplier*. It calculates and rates how well suppliers have performed in the past, evaluating delivery performance, comparing prices, as well as user ratings persisted in the *Rating* service itself. After the service has been deployed in production, it can be observed (through standard monitoring tools), that the new *Rating* service shows poor response time behavior, which had not been noticed through tests during development. The root cause for this performance problem has been that the complete data to recalculate the supplier rating based on orders from this supplier has been fetched at runtime from the *PO* service. With a growing number of orders this had slowed down the *Rating* service. Moreover, the algorithm used to calculate the rating was linear in the number of order items per supplier. To identify the root cause of the performance problem, the *Rating* service developers had to perform a manual analysis. They had to log the data volume accessed from the *PO* service and analyze the usage of this data inside the rating

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

algorithm. Only then they could start to re-engineer the algorithm, replicate data from the *PO* service, or use a data aggregation API at the *PO* service.

At the time the performance problem was discovered and the analysis was conducted, it already had an impact on the end user. We argue that *feedback* on the application can be provided much earlier, already during the development of the *Rating* service. All operations data relevant for identifying the root cause of the performance problem (e.g. data volume inside the PO service) has already been available. It has just not been pushed to the appropriate level of abstraction, namely the development artifacts, such as the involved REST calls in the program code. Additionally, the available feedback has not been integrated into the daily workflow and tools of developers.

As we will detail in the following sections, FDD is about enabling the automation of this feedback process: aggregating operations data, binding it to development artifacts, and predicting and visualizing it in the IDE.

4. Feedback-Driven Development

In this section, we introduce our approach for *Feedback-driven Development* (FDD), a paradigm for systematically exploiting operations data to improve development of SaaS applications. We discuss that FDD is about tightly integrating the collection and analysis of feedback from production deployments into the IDEs that developers use to actually work on new versions of the application. By nature, FDD is particularly suited to support more service-oriented and CD-based projects, but the underlying ideas are useful for the development of any kind of SaaS application.

FDD is not a concrete tool or process. Rather, FDD is an abstract idea, which can be realized in different ways, using different kinds of operations data and feedback, to support the developer in different ways. We refer to these different flavors of the same underlying idea as *FDD use cases*. In this section, we discuss general FDD concepts, while concrete use cases and prototypical example implementations on top of different cloud and monitoring systems are the scope of Section 5.

4.1 Conceptual Overview

A high-level overview that illustrates the core ideas behind FDD is given in Figure 3. At its heart, FDD is about making operations data, which is routinely collected in the runtime environment, actionable for software developers. To this end, we transform *source code artifacts* (e.g., services, methods, method invocations, or data structures) into one or more graph representations modeling the dependencies between different artifacts (*dependency graphs*). Each node in these dependency graph represents a source code artifact $a_i \in \mathcal{A}$. The concrete semantics of these graphs, and what they contain, differ for different FDD use cases. For example, for feedback-based performance prediction, the dependency graph is a simplified abstract syntax tree (AST) of the application, containing only method definitions, method calls, and control structures (`ifs` and `loops`).

Operations data (e.g., service response times, server utilization, but also production data, such as the number of purchase orders as in our example) is collected from *runtime entities* (e.g., services or virtual machines running in the production environment). Every operations data point, d_i, is represented as a quadruple, $d_i = \langle t, \tau, e, v \rangle$, where:

- t is the time the operations data point has been measured,
- τ is the type of operations data (as discussed in Section 4.2.1),
- e refers to the runtime entity that produced the data point,
- v is the numerical or categorical value of the data point.

Operations data deriving from the measurement of a runtime entity usually come in the form of a series $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$. In our illustrative example, relevant runtime entities are, for instance, the services, virtual machines, and databases. Operations data can be delivered in various forms, for instance through execution logs or events. *Feedback control* is the process of filtering, integrating, aggregating, and mapping this operations data to relevant nodes in the dependency graphs generated from code artifacts. Therefore, we define *feedback* as the mapping from source code artifacts to a set of operations data points, $\mathcal{F} : \mathcal{A} \mapsto \{\mathcal{D}\}$.

This process is steered by *feedback mapping*, which includes specifications (tailored to expected use cases) that contain the knowledge which operations data is mapped to which entries in the dependency graphs, and how. In our example, the services would need to be mapped to their invocation in the program code.

This *feedback-annotated dependency graphs* then form the basis of concrete FDD use cases or tools, which either *visualize* feedback in a way that is more directly actionable for the software developer (left-hand side example use case in Figure 3), or use the feedback to *predict* characteristics of the application (e.g., performance or costs) prior to deployment (right-hand side example).

4.2 Operations Data and Feedback

We now discuss the collection, aggregation, and mapping of operations data to feedback in more detail.

4.2.1 Types of Operations Data

In Figure 4, we provide a high-level taxonomy of types of operations data. Primarily, we consider *monitoring data*, i.e., the kind of operational application metadata that is typically collected by state-of-the-art APM tools, and *production data*, i.e., the data produced by the SaaS application itself, such as placed orders, customer information, and so on.

Monitoring data can be further split up into *execution performance data* (e.g., service response times, database query times), *load data* (e.g., incoming request rate, server

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

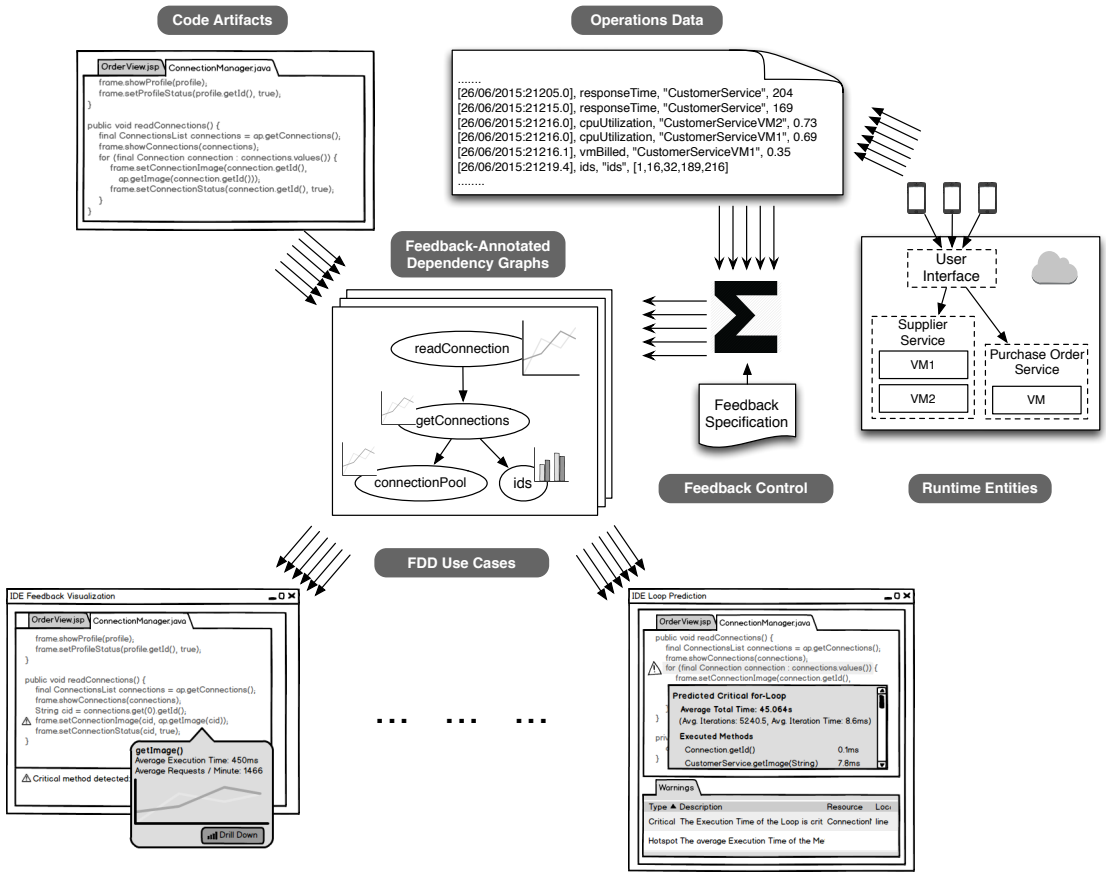

Figure 3. Conceptual overview of Feedback-Driven Development. Code artifacts are transformed into use case specific dependency graphs, which are enriched with feedback harvested in the production cloud environment. The annotated dependency graphs are then used to visualize feedback directly in the IDE, as well as predict the impact of changes to the program code.

utilization), *costs data* (e.g., hourly cloud virtual machine costs, data transfer costs per 10.000 page views), and *user behavior data* (e.g., clickstreams).

4.2.2 Feedback Control

All this operations data is, in principle, already available in today's cloud solutions, either via built-in cloud monitoring APIs (e.g., CloudWatch[4] in Amazon Web Services) or through external APM solutions. However, operations data by itself is typically not overly interesting to developers without proper analysis, aggregation, and integration. This is what we refer to as feedback control.

Feedback control is steered by feedback specifications, which are custom to any specific FDD use case. Further, software developers typically are able to further refine feedback specifications during visualization (e.g., via a slider in the IDE which controls the granularity of feedback that is visualized). Feedback control encompasses five steps, (1) data collection, (2) data filtering, (3) data aggregation, (4) data integration, and (5) feedback mapping, as depicted in Figure 5.

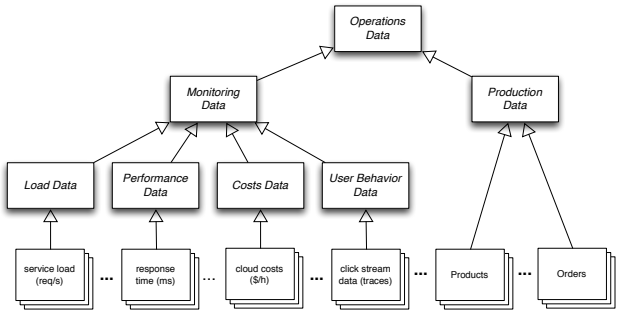

Figure 4. Overview of the types of operations data we consider in FDD and concrete examples. We distinguish between production data (i.e., the payload of the application, for instance, placed orders) and the monitoring information delivered by APM tools (e.g., response times or service load). A special type of APM data is usage data (e.g., click streams).

[4] `http://aws.amazon.com/cloudwatch/`

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

Data collection controls how operations data is monitored and collected on the target system. This may include instrumenting the application to produce certain operations data which would be unavailable otherwise (e.g., by sending required production data to the APM tool), or by configuring the APM tool (e.g., to collect operations data for a given percentage of users or only during system load below x percent).

Data filtering controls how the data is filtered for a specific use case. That is, FDD use cases often differ in the types and resolution of required operations data. This includes selecting the type of operations data relevant for the specific FDD use case. For resolution, for instance, performance visualization use cases often require fine-grained data to display accurate dashboards and to allow developers to drill down. Performance prediction, on the other hand, does not require data on the same resolution, and can work on a more coarse-grained sampling.

Data aggregation controls how operations data should be compressed for a use case. For some use cases, basic statistics (e.g., minimum, maximum, arithmetic mean) are sufficient, while other use cases require data on a different level of granularity.

Data integration controls how different types of operations data (or operations data originating from different runtime entities) should be integrated. For instance, in order to calculate the per-request costs of a service, the hourly costs of all virtual machines hosting instances of this service need to be integrated with request counts for this service.

Finally, ***feedback mapping*** links collected, filtered, aggregated, and integrated operations data to development-time source code artifacts. This final step transforms operations data into *feedback*, i.e., data that is immediately actionable for developers.

Each of the first four steps takes the form of transformation functions, taking as input one or more series of operations data $\mathcal{D}$, and produces one or more series of operations data $\mathcal{D}'$ as output. In *feedback specification*, these functions can be represented using, for instance, the complex event processing [29] (CEP) abstraction (i.e., using Esper Pattern Language (EPL[5]). The final step, *feedback mapping*, is typically encoded in FDD tools. That is, the knowledge which series of operations data should be mapped to which source code artifacts is generally hard-coded in FDD implementations for a specific use case.

4.2.3 Feedback Freshness

One of the challenges with integrating feedback is identifying when the application has already sufficiently changed so that feedback collected before the change should not be considered anymore (i.e., the feedback became "old" or "stale"). A naive approach would simply ignore all data that had been gathered before any new deployment. However, in a CD process, where the application is sometimes deployed multiple times a day, this would lead to frequent resets of the available feedback. This approach would also ignore external factors that influence feedback, e.g., additional load on a service due to increased external usage.

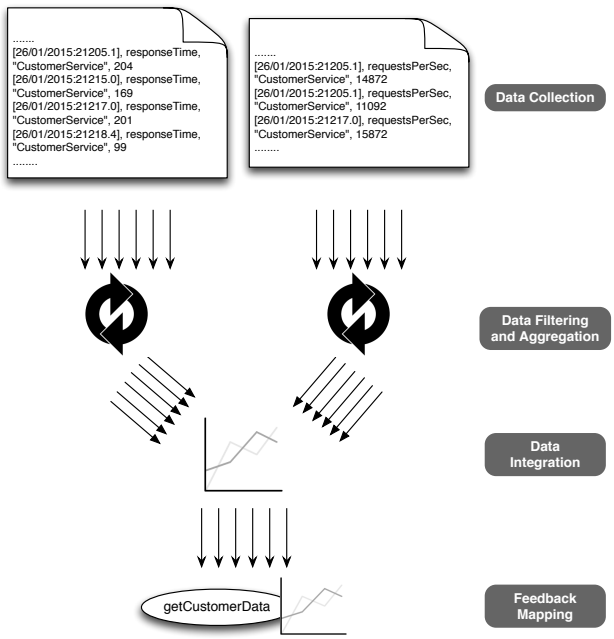

Figure 5. Feedback is filtered, aggregated, and integrated operations data, which has been mapped to source code artifacts (e.g., method calls).

Hence, we propose the usage of statistical changepoint analysis on feedback to identify whether data should still be considered "fresh". Changepoint analysis deals with the identification of points within a time series where statistical properties change. For the context of observing changes of feedback data, we are looking for a fundamental shift in the underlying probability distribution function. In a time series, we assume that the observations come from one specific distribution initially, but at some point in time, this distribution may change. Recalling the series of observations in operations data in Section 4.1, $\mathcal{D} = \{d_1, d_2, \ldots, d_n\}$, a changepoint is said to occur within this set when there exists a point in time, $\tau \in \{1, ..., n-1\}$, such that the statistical properties of $\{d_1, ..., d_\tau\}$ and $\{d_{\tau+1}, ..., d_n\}$ exhibit differences. The detection of these partition points in time generally takes the form of hypothesis testing. The null hypothesis, H_0, represents no changepoint and the alternative hypothesis, H_1, represents existing changepoints. In previous work, we have already shown that changepoint analysis can be successfully employed to detect significant changes in the evolution of operations data [11].

[5] `http://www.espertech.com/esper/index.php`

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

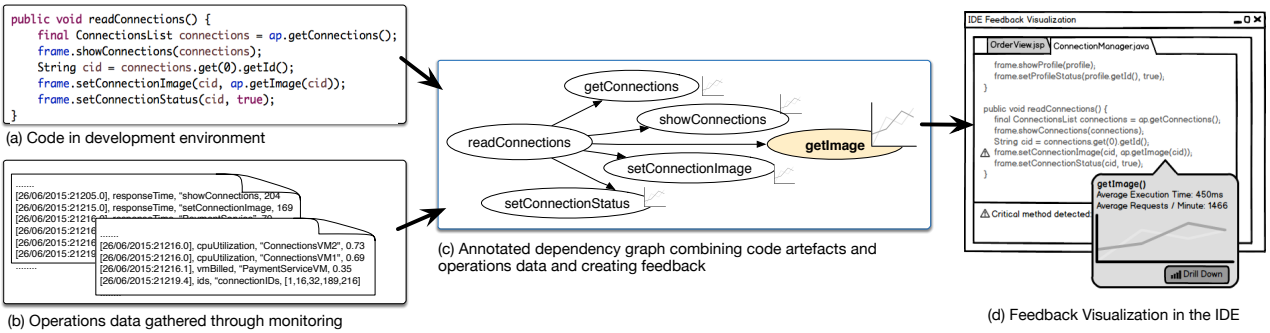

Figure 6. How visualization works: (a) A developer examines the method *readConnections*. (b) The application is in production and operations data is available from concrete traces. (c) The dependency tree is annotated with operations data, creating feedback. (d) The feedback visualization is integrated in the code view of the IDE.

4.3 Supporting Developers With Feedback

The purpose of feedback is thus by definition to support developers in reasoning about the future behavior of the systems they create. We can distinguish two types of FDD use cases: analytic FDD and predictive FDD. The former deals with analyzing operations data and its relation to development artifacts *a-posteriori*. The latter provides a *prediction* of future system behavior under certain assumptions. In practice, analytic FDD often takes the form of feedback visualization, while predictive FDD is primarily concerned with inferring (as of yet unknown) operations data prior to deployment of a change.

4.3.1 Feedback Visualization

After feedback data is collected it needs to be displayed to the developer in a meaningful context to become actionable. As with all visualizations, the chosen visualization techniques should be appropriate for the data at hand, allowing the user to interactively explore the recorded feedback data and quickly identify patterns. In the context of FDD, the more interesting challenge is to put the implicitly existing link between feedback data and the development-time artifacts (*feedback mapping*) to good use. Figure 6 illustrates the process of feedback visualization starting from the developer's code view and execution traces to how standard IDE views are being enriched with feedback.

A developer is examining `readConnections()`, consisting of a small number of method calls. Different kinds of operations data on these methods have been made available by monitoring solutions. A combination of simple static analysis (extracting a simplified AST and call graph in Figure 6a) and dynamic analysis (extracting relevant metrics from concrete traces in Figure 6b) results in the annotated dependency graph seen in Figure 6c. Note that the `getImage` node is highlighted, as it is the only artifact deemed relevant in this scenario. Relevance is determined from a combination of parameters of *feedback control* and statistics calculated from the values of the attached feedback data (e.g., methods with an average execution time over a developer-defined threshold). These artifacts are then highlighted in the IDE through warnings and overlays in the exact spot the identified issue occurred, as depicted in a mockup in Figure 6d. Developers are now able to utilize this targeted feedback to guide their design decisions and improve the application. Exemplary concrete visualization techniques that we have experimented with in our concrete tooling are shown in Section 5.

4.3.2 Predicting Future Behavior Based on Feedback

Predictive FDD aims at deriving the impact of changes during development in an application based on existing feedback. Figure 7 illustrates the steps leading to the prediction of future behavior of an application. A developer changes the code of the method `overallRating()`, adding an iteration over suppliers (`getSuppliers()`) and a call to a different service (`getPurchaseRating()`). Figure 7b shows how this code change transforms the dependency graph (described in Section 4.1). The change introduced 3 new nodes where feedback is already available from existing traces: (1) `getSuppliers`, (2) collection size of `suppliers`, and (3) `getPurchaseRating`. The steps "Statistical Inference" and "Feedback Propagation", illustrated in 7c, complete the prediction cycle. Feedback for the iteration node `Loop:suppliers` is inferred using the feedback of its child nodes (`size:suppliers` and `getPurchaseRating`) as parameters for a statistical inference model, identifying it as a *critical entity*. The concrete statistical model is specific to the use case. For instance, for one of our use cases (*PerformanceHat* in Section 5.2), we chose to implement a quite simplistic model for loop prediction. We model the total execution time of a loop, $\tau(l)$, as the sum of the average execution times, $\bar{\tau}$, of all methods within the loop, $\{l_{m,1}, \ldots, l_{m,n}\}$, times the average collection size, $|l|$, the loop is iterating over: $\tau(l) = \sum_{i=1}^{n} \bar{\tau}(l_{m,i}) \times |l|$. Depending on the specific application nature, the complexity of these inference models can vary greatly. In a last step, all de-

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

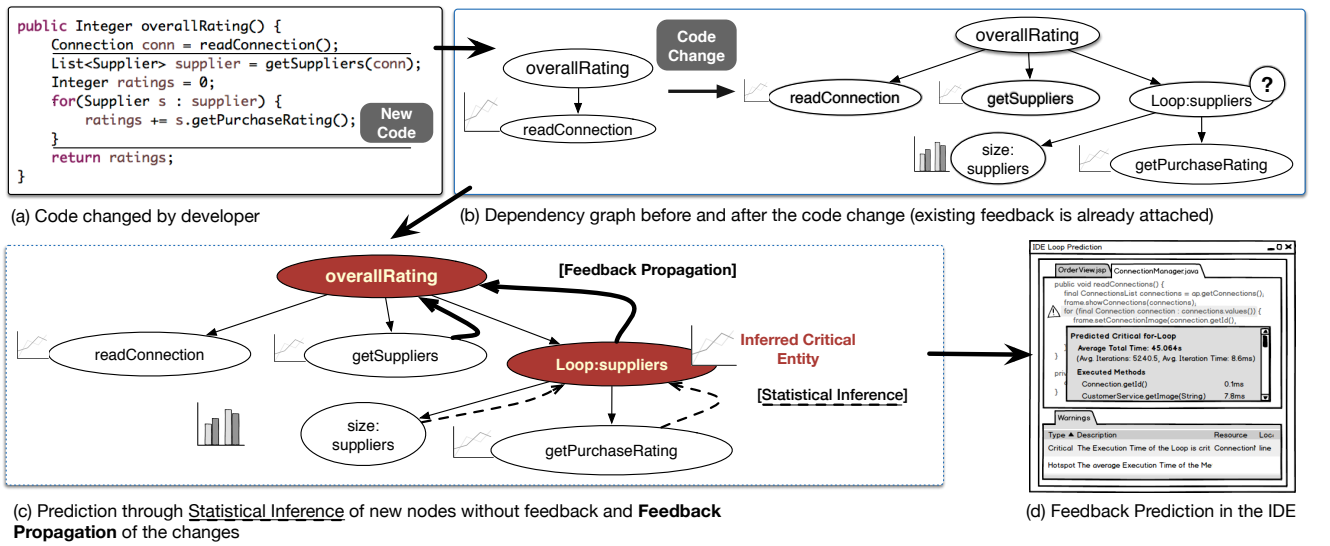

Figure 7. How Prediction works: (a) A developer changes the program and adds a loop within the method `overallRating()`. (b) The annotated dependency graph changes accordingly with the change, adding the loop (Loop:suppliers) and its nodes (*size:suppliers* and *`getPurchaseRating`*). (c) Prediction and change impact analysis is achieved through statistical inference and feedback propagation. (d) The prediction is integrated in the code view of the IDE, warning the developer about a preformance-critical change.

rived feedback from changes are propagated in the nodes of the graph following its dependency edges. This kind of prediction allows us to warn developers about possible issues of their code changes prior to even running the application in production, as shown in a mockup in Figure 7d.

5. Case Studies

As discussed in Section 4, the abstract idea of FDD can be instantiated in various ways, and through various concrete developer tools. We now discuss three concrete tools which implement the FDD paradigm in different ways.

5.1 FDD-Based Expensive Artifacts Detection

Performance Spotter is an experimental set of tools developed on top of the SAP HANA Cloud Platform[6]. The aim of *Performance Spotter* is to help the developers in finding expensive development artifacts. This has been achieved by creating analytic feedback based on collected operations data (see Section 4.2) and mapping this feedback onto corresponding development artifacts in the IDE. Other information such as the number of calls and the average execution time are derived by aggregating the collected data. *Performance Spotter* provides ways to promote performance awareness, root cause analysis, and performance analysis for external services. Hence, *Performance Spotter* is one instantiation of an analytic FDD tool.

Performance Awareness. *Performance Spotter* helps developers to become aware of potential performance issues by observing, aggregating and then visualizing the collected metrics of artifacts. Figure 8, illustrates *Performance Spotter*'s Functions Performance Overview. On the left side of the figure, a Javascript code fragment is depicted. On the top right, a list of functions with their relative execution times to other functions is visualized. The blue bars represent the relative execution times. On the bottom right, a diagram illustrates the average execution times of selected functions over time. We can identify poorly performing functions by using the given overview. For instance, Figure 8 shows that the average execution time of `getPOs()` is very large in relation to other functions, and that the performance of this function has recently decreased, Furthermore, execution times have increased significantly at particular application runs (*problematic sessions*) and stayed almost stable afterwards. Identifying problematic sessions enables developers to analyze the impact of code as well as resource changes on certain artifacts' performance behavior. In this particular case, the cause of increasing the execution times was code changes in `getPOItems()` before each problematic sessions. However, discovering the root cause of the problem required more insight into the collected monitoring data.

Root Cause Analysis. However, knowing that a development artifact suffers from poor performance is by itself not sufficient. Developers need to find the root cause of such issues. Thus, another feature of *Performance Spotter* is Functions Flow, which builds an annotated dependency graph of functions. The nodes of this graph are function calls and there is an edge between two nodes if one function calls an-

[6] http://hcp.sap.com

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

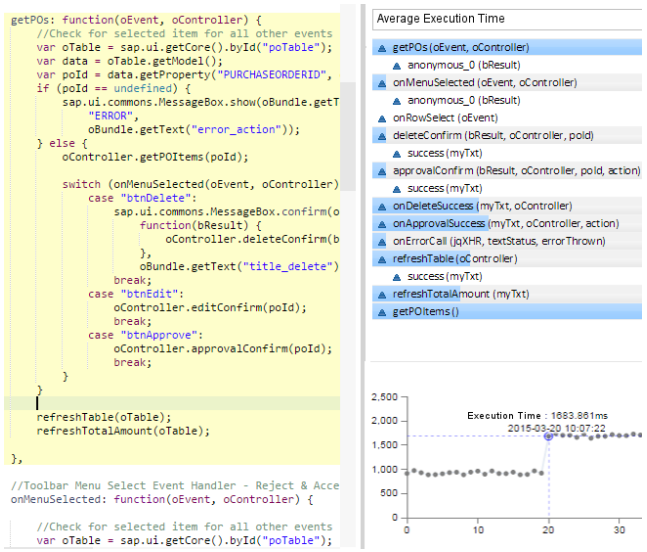

Figure 8. *Performance Spotter*'s Functions Performance Overview.

other (i.e., a call graph). The nodes are annotated with relevant feedback (e.g., execution time). Having a visualization of the dependency graph of functions, we can find the root cause of performance issues by traversing the graph and following the poorly performing nodes. The Functions Flow of the artifact `getPOs()` is depicted in Figure 9, showing that `getPOItems()` is the most expensive function call, i.e., it is the root cause of the performance problem.

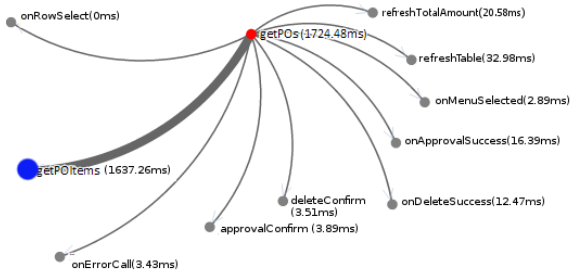

Figure 9. *Performance Spotter*'s Functions Flow helps the developer to find expensive functions.

External Service Performance Analysis. Since in many cases an external service (e.g., database) is called within an application, it is necessary to keep track of its behavior from an application's perspective. Previously, we have detected `getPOItems()` as the cause for the high execution time of `getPOs()`. Internally, this method uses a number of database calls, which indicates that improving the database calls would improve the overall performance of the function. *Performance Spotter* provides a feature to analyze the database statements directly from where they are called. Figure 10 shows a piece of code (on the top) and the Database Performance Overview (on the bottom). This feature enables the developer to find the expensive database statements by sorting and/or filtering them.

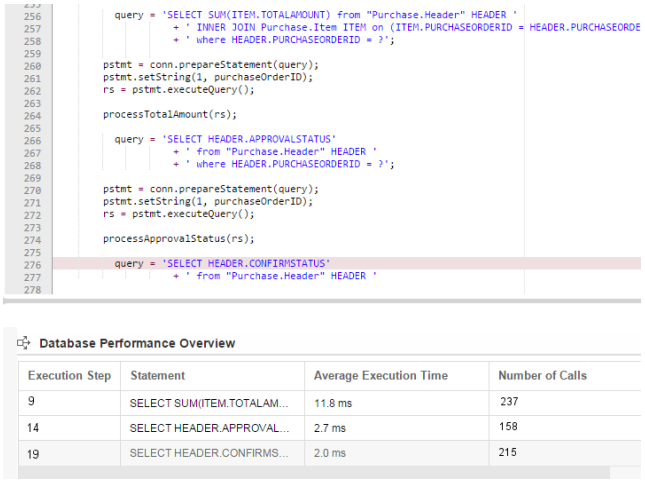

Figure 10. *Performance Spotter*'s Database Performance Overview helps the developer to find expensive database calls.

5.2 FDD-Based Prediction of Performance Problems

A second concrete implementation of the FDD paradigm is *PerformanceHat*, a prototypical Eclipse IDE plugin that is able to predict performance problems of applications during development in the IDE (i.e., prior to deployment). It works with any underlying (cloud) platform, as long as the required operations data (see discussion below) is available. *PerformanceHat* is consciously designed in a way that it can easily interface with a variety of monitoring solutions as a backend for operations data (e.g., APM solutions such as NewRelic). The current proof-of-concept version of *PerformanceHat* provides two main features, *hotspot detection* and *critical loop prediction*.

Detecting Hotspots. Hotspots refer to methods that, in the production execution of the application, make up a substantial part of the total execution time of the application (cp. "expensive artifacts" in the previous case study discussion). In a traditional environment this information could be looked up in dashboards of APM solutions, requiring a context switch from the development environment to the operations context of performance dashboards. It also requires further navigation to a specific location in these dashboards. In the FDD paradigm, such hotspot methods are reported as warnings attached to a software artifact within the development environment. Figure 11 gives an example of a hotspot. Notice that hotspot methods are identified both at method definition level (e.g., `private void login()`) and method call level (e.g., `Datamanager(b).start()`) in Figure 11. When hovering over the annotated entities a tooltip displays

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

summary statistics (currently the average execution time) as a first indicator of the performance problem, as well as a deep link to a dashboard visualizing the time series of operational metrics that led to the feedback.

For hotspot detection, *PerformanceHat* requires only method-level response times in `ms`, as delivered by state-of-the-art monitoring solutions. For program statements for which no response times are available, a response time of `0 ms` is assumed. In practice this means that, oftentimes, we primarily detect hotspots of statements that implement interactions with external components or services (e.g., database queries, remote method invocations, JMS messaging, or invocations of REST interfaces). In Figure 11, `login()` is identified as a hotspot, as `DataManager.start(..)` invokes an external REST service in the Microservice based architecture of the *MicoSRM* illustrative example (see Section 3). Other statements, for instance `b.getPassword()`, are ignored as the response times for these statements are negligible.

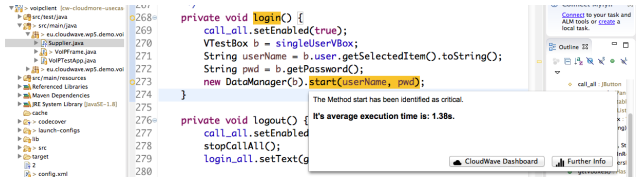

Figure 11. The *PerformanceHat* plugin warning the developer about a "Hotspot" method.

Critical Loop Prediction.

Performance problems are often related to expensive loops [23]. As described in Section 4.3.2, in FDD we predict the outcome of changes in code by utilizing existing data to infer feedback for new software artifacts. In *PerformanceHat* we are able to do so for newly introduced loops over collections (i.e., *foreach* loops). In the initial stages of our prototype we propose a simplified model over operations data on collection sizes and execution times of methods to infer the estimated execution time of the new loop (as discussed in Section 4.3.2). Figure 12 gives an example of a so-called *Critical Loop*. When hovering over the annotated loop header a tooltip displays the estimated outcome (*Average Total Time*), the feedback parameters leading to this estimation (*Average Iterations* and *Average Time per Iteration*), and the execution times of all methods in the loop. This information enables developers to dig further into the performance problem, identify bottlenecks and refactor their solutions to avoid poor performance even *before* committing and deploying their changes.

We are in the process of releasing *PerformanceHat* as an open source project at GitHub[7], as a plugin compatible with Eclipse Luna (Version 4.4.1) and onwards. A screencast demonstrating *PerformanceHat*'s capabilities can be found online[8] as well.

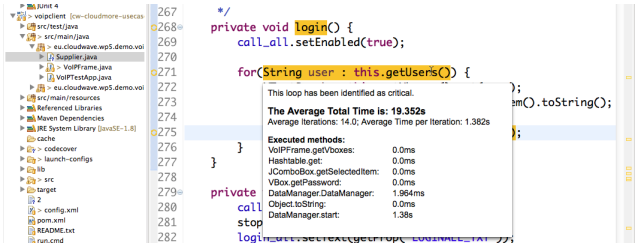

Figure 12. Prediction of Critical Loops in the *PerformanceHat* plugin.

5.3 FDD-Based Prediction of Costs of Code Changes

It is often assumed that deploying applications to public, on-demand cloud services, such as Amazon EC2 or Google Appengine, allows software developers to keep a closer tab on the *operational costs* of providing the application. However, in a recent study, we have seen that costs are still usually intangible to software developers in their daily work [12]. To increase awareness of how even small design decisions influence overall costs in cloud applications, we propose integrated tooling to predict costs of code changes following the FDD paradigm. We present our ideas on how we can predict costs *induced by introducing a new service* and by *replacing existing services*. Unlike the *Performance Spotter* and *PerformanceHat* use cases, our work on this idea is still in an early stage. We are currently in the process of building a proof-of-concept implementation for these ideas on top of Eclipse, NewRelic and AWS CloudWatch.

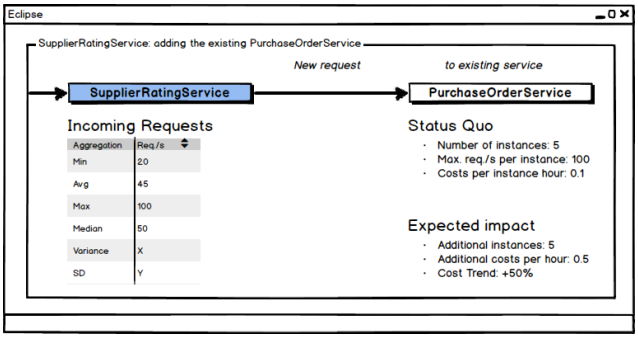

Figure 13. IDE Tooling displaying how the new service SupplierRating has an impact on the cost of existing service of the PurchaseOrder.

Costs through new Services. When considering services deployed on cloud infrastructure, increasing load usually leads to the addition of compute instances to scale horizontally or vertically. In this scenario, illustrated in Figure 13, we introduce a new service *SupplierRatingService* that invokes the existing *PurchaseOrderService*. The IDE tooling

[7] http://www.github.com/sealuzh/PerformanceHat

[8] http://bit.ly/PerformanceHat

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

provides information on the deployment and cost structure of the existing service (*StatusQuo*) and provides an impact analysis on how this structure would change (*Expected Impact*) based on load pattern parameters (*Incoming Requests*). The load pattern parameters would be estimated by leveraging monitoring data for similar services in the application and can be adjusted by the developer to perform sensitivity analysis.

Costs induced by Replacing Services. Another, similar, scenario in Figure 14 considers the replacement of an invocation within an existing service (*OtherPaymentService*) with a new service (*NewPaymentService*). In this case, the load pattern parameters are known and pre-populated in the interface (*Incoming Requests*). The impact analysis (*Expected Impact*) differs from the previous case in that the model takes into account partial rollouts (*Simulation*). This allows for more complex sensitivity analysis scenarios.

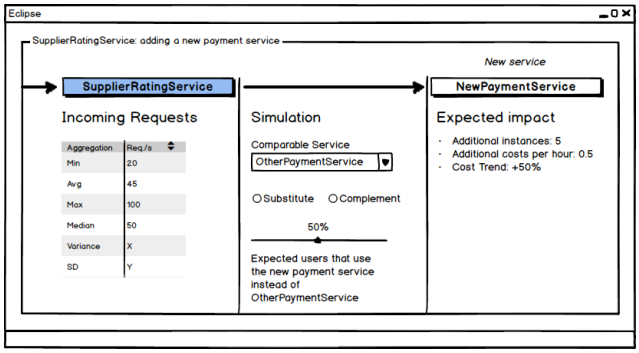

Figure 14. IDE Tooling displaying how the replacement of an existing service (OtherPaymentService) by a new service (NewPaymentService) has an impact on overall costs.

6. Challenges Ahead

Based on our case study implementations, as well as based on initial experiments and discussions with practitioners, we have identified a small number of interesting challenges that need to be addressed before the FDD idea can be deployed on a larger scale.

Data Access and Privacy. The availability of rich, up-to-date, and correct operations data is the cornerstone upon which FDD builds. While modern cloud and APM solutions already provide a wealth of data, we have still encountered concerns regarding the availability of some of the data discussed in Figure 4. Most importantly, production data (e.g., user information, order data) will, in many cases, be unavailable to engineers due to privacy and data protection concerns. Consequently, our initial use cases in *PerformanceHat* and *Performance Spotter* did not make use of these types of operations data. Relatedly, when deploying the FDD idea on Web scale, we will also face the orthogonal problem that there will in many cases actually be *too much* data available to use directly in developer-facing FDD tools.

For both cases, it will become necessary to invest more research into how operations data is actually harvested in production. In terms of privacy protection, we envision future monitoring solutions to be privacy-aware, and be able to automatically anonymize and aggregate data to the extent required by local data protection laws or customer contracts. We expect that many lessons learned from privacy-preserving data mining [45] can be adapted to this challenge. Further, we need to devise monitoring solutions that are able to sample operations data directly at the source. While it is easy to only generate data for a subset of calls (e.g., only track 1% of all service invocations), doing so while preserving the relevant statistical characteristics of the underlying distribution is not trivial.

Confounding Factors. All prototypical FDD use cases discussed in Section 5 are operating on the simplified assumption that feedback is largely free from confounding factors, such as varying service performance due to external influences (e.g., variance in networking performance, external stress on the service, or a service scaling up or down due to changes in customer behavior). This problem is amplified by the fact that cloud infrastructures are known to provide rather unpredictable performance levels [26]. When deploying FDD in real projects, such confounding factors will lead to false positives or negatives in predictions, or produce misleading visualizations.

More work will be required on robust data aggregation methods that are able to produce useful feedback from noisy data. These methods will likely not only be statistical, but integrate heterogeneous operations data from different levels in the cloud stack (e.g., performance data, load data, scaling information) to produce clearer and more accurate feedback. First steps in this direction have already been conducted under the moniker 3-D monitoring [31].

Information Overload. Another challenge that all FDD use cases discussed in Section 5 face is how to display the, and only the, information that is relevant to a given developer. Ultimately, visualizations and predictions generated by FDD tools need to not only be accurate, but also be relevant to the developer. Otherwise, there is a danger that developers start ignoring or turning off FDD warnings. This problem is amplified by the fact that different developers and project roles care about different kinds of problems.

A technical solution to this challenge is customization support. Feedback control (see Section 4.2.2) enables developers to turn specific feedback on and off, to restrict feedback to certain services or methods, or to change the thresholds for warnings. For instance, in *PerformanceHat*, developers can change the threshold from which a method is displayed as a hotspot, either globally or on a per-method level. However, in addition, more research is necessary to know which kinds of feedback, warnings and predictions are typically useful for which kinds of developers and projects. Existing research in software engineering on information

needs [7, 16] and developer profiles [6] can serve as a basis for this work.

Costs as Opportunity. Our aim to integrate monetary considerations within the FDD paradigm is to increase awareness about the costs design decisions have when developing for the cloud. However, this awareness should go beyond considering costs solely as a burden, but rather as an opportunity (e.g., better performance through more computing power leads to more sales).

Possible solutions to this challenge include associating cost models in FDD with business metrics, as well as proper information visualization to underline opportunities.

Technical Challenges. Finally, there are a number of technical challenges that need to be tackled when realizing the FDD paradigm in industry-strength tools. Specifically, in our work on *PerformanceHat* and SAP HANA's *PerformanceSpotter*, we have seen that implementing FDD in a light-weight way, such that the additional static analysis, statistical data processing, and chart generation does not overly slow down the compilation process or the perceived responsiveness of the IDE, is not trivial from a technical perspective. *Feedback control*, as discussed in Section 4.2.2, mitigates this challenge somewhat by minimizing the nodes in the generated dependency graphs. However, particularly prediction of the impact of code changes (see Section 4.3.2) is currently taxing in terms of performance, as predicted changes to the feedback associated to one method need to be propagated through the entire application AST. We are currently investigating heuristics to speed up this process.

7. Related Work

A substantial body of research exists in the general area of cloud computing [19], including work on cloud performance management and improvement (e.g., [17, 22, 40], among many others). However, as [2] notes, software development aspects of cloud computing and SaaS are currently not widely explored. We have recently provided a first scientific study that aims to empirically survey this field [12].

One of the observations of this study was that cloud platforms and APM tools (e.g., the commercial solutions NewRelic or Ruxit[9]) make a bulk of data available for software developers, but that developers currently struggle to integrate the provided data into their daily routine. Existing research work, for instance in the area of service [28, 36, 39], cloud [31, 35], or application monitoring [44], provides valuable input on how monitoring data should be generated, sampled, and analyzed. There is very little research on how software developers actually make use of this data to improve programs. FDD is an approach to address this gap. However, FDD is not a replacement of APM, but rather an extension that makes use of the data produced by APM tools. To this end, our work bears some similarities to research in the area of live trace visualization [15, 18]. However, our work goes beyond the scope of visualization of program flow.

Some parts of FDD, specifically the use cases more geared towards predicting performance problems before deployment, are based on prior work in the area of performance anti-patterns [41, 46] and their automated detection. Our implementation of these predictive FDD use cases also leans heavily on prior work in static and dynamic program analysis [38].

It is possible to view FDD as a pragmatic approach to bring cloud development closer to the idealistic view of *live programming* [32, 33]. Live programming aims to entirely abolish the conceptual separation between editing and executing code. Developers are editing the running program, and immediately see the impact of any code change (e.g., on the output that the program produces). Hence, developers can make use of immediate feedback to steer the next editing steps [8]. Existing work on live programming has also stressed the importance of usable IDEs and development environments [27]. FDD is an approach to take similar core ideas (bringing feedback, e.g., in terms of execution time, back into the IDE), but plugging them on top of existing programming languages, tools and processes, rather than requiring relatively fundamental changes to how software is built, deployed, and used. FDD also acknowledges that, for enterprise-level SaaS applications, local execution (as used in live programming) is not necessarily a good approximation for how the application will perform in a Web-scale production environment. Hence, the production feedback used in FDD is arguably also of more use to the developer than the immediate feedback used in live programming.

8. Conclusions

In this paper, we presented our vision on Feedback-Driven-Development, a new paradigm that aims at seamlessly integrating feedback gathered from runtime entities into the daily workflow of software developers, by associating feedback to code artifacts in the IDE. We discussed how operations data produced by APM solutions is filtered, aggregated, integrated, and mapped to relevant development-time artifacts. This leads to annotated dependency trees, which can then be utilized for different use cases. In this paper, we have focused on two types of FDD use cases. Analytic FDD focuses on displaying relevant collected feedback visually close to the artifacts that it is relevant for, hence, making the feedback actionable for developers. Predictive FDD makes use of already collected feedback in combination with static analysis to predict the impact of code changes in production. We have exemplified these concepts based on two implementations, on top of Eclipse as well as on top of SAP's HANA cloud solution.

We believe that the general idea of FDD will, in the upcoming years, become more and more important for software developers, tool makers, and cloud providers. Cur-

[9] https://ruxit.com

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015

rently, we are arguably only seeing the tip of the iceberg of possibilities enabled by integrating operations data into the software development process and tools. However, we also see a number of challenges that need to be addressed before the full potential of FDD can be unlocked. Primarily, we need to address questions of privacy and correctness of data. Further, more academic studies will be required to identify which kinds of feedback developers actually require in which situations. Finally, our practical implementations have also shown that there are numerous technical challenges to be addressed when trying to make Web-scale data useful for the developer in (or close to) real-time.

Acknowledgments

We thank Christian Bosshard for his implementation efforts in *PerformanceHat* and Emanuel Stöckli for the initial mockups in the Cost Prediction use case. The authors would also like to thank Ivan Beschastnikh for the input on how to model feedback inference and propagation. The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 610802 (CloudWave).

References

[1] N. Ayewah, D. Hovemeyer, J. Morgenthaler, J. Penix, and W. Pugh. Using Static Analysis to Find Bugs. *IEEE Software*, 25(5):22–29, Sept 2008. ISSN 0740-7459.

[2] A. Barker, B. Varghese, J. S. Ward, and I. Sommerville. Academic Cloud Computing Research: Five Pitfalls and Five Opportunities. In *Proceedings of the 6th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 14)*, Philadelphia, PA, June 2014. USENIX Association. URL `https://www.usenix.org/conference/hotcloud14/workshop-program/presentation/barker`.

[3] C.-P. Bezemer and A. Zaidman. Multi-tenant SaaS Applications: Maintenance Dream or Nightmare? In *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, IWPSE-EVOL '10, pages 88–92, 2010. ISBN 978-1-4503-0128-2.

[4] C.-P. Bezemer, A. Zaidman, B. Platzbeecker, T. Hurkmans, and A. t. Hart. Enabling multi-tenancy: An industrial experience report. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–8, Washington, DC, USA, 2010. IEEE Computer Society. ISBN 978-1-4244-8630-4. . URL `http://dx.doi.org/10.1109/ICSM.2010.5609735`.

[5] C. Bizer, P. Boncz, M. L. Brodie, and O. Erling. The Meaningful Use of Big Data: Four Perspectives – Four Challenges. *SIGMOD Record*, 40(4):56–60, Jan. 2012. ISSN 0163-5808.

[6] M. Brandtner, S. C. Müller, P. Leitner, and H. Gall. SQA-Profiles: Rule-Based Activity Profiles for Continuous Integration Environments. In *Proceedings of the 22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER'15)*, 2015.

[7] S. Breu, R. Premraj, J. Sillito, and T. Zimmermann. Information Needs in Bug Reports: Improving Cooperation Between Developers and Users. In *Proceedings of the 2010 ACM Conference on Computer Supported Cooperative Work*, CSCW '10, pages 301–310, 2010. ISBN 978-1-60558-795-0.

[8] S. Burckhardt, M. Fahndrich, P. de Halleux, S. McDirmid, M. Moskal, N. Tillmann, and J. Kato. It's Alive! Continuous Feedback in UI Programming. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 95–104, 2013. ISBN 978-1-4503-2014-6.

[9] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computing Systems*, 25:599–616, 2009.

[10] L. Chen. Continuous Delivery: Huge Benefits, but Challenges Too. *IEEE Software*, 32(2):50–54, 2015.

[11] J. Cito, D. Suljoti, P. Leitner, and S. Dustdar. Identifying Root-Causes of Web Performance Degradation using Changepoint Analysis. In *Proceedings of the 14th International Conference on Web Engineering (ICWE)*. Springer Berlin Heidelberg, 2014.

[12] J. Cito, P. Leitner, T. Fritz, and H. C. Gall. The Making of Cloud Applications – An Empirical Study on Software Development for the Cloud. *ArXiv e-prints*, Mar. 2015.

[13] P. Cousot and R. Cousot. Modular static program analysis. In R. Horspool, editor, *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, pages 159–179. Springer Berlin Heidelberg, 2002. ISBN 978-3-540-43369-9. . URL `http://dx.doi.org/10.1007/3-540-45937-5_13`.

[14] D. G. Feitelson, E. Frachtenberg, and K. L. Beck. Development and Deployment at Facebook. *IEEE Internet Computing*, 17(4):8–17, 2013. ISSN 1089-7801.

[15] F. Fittkau, J. Waller, C. Wulf, and W. Hasselbring. Live Trace Visualization for Comprehending Large Software Landscapes: The ExplorViz Approach. *2013 First IEEE Working Conference on Software Visualization (VISSOFT)*, pages 1–4, 2013. .

[16] T. Fritz, J. Ou, G. C. Murphy, and E. Murphy-Hill. A Degree-of-knowledge Model to Capture Source Code Familiarity. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ICSE '10, pages 385–394, 2010. ISBN 978-1-60558-719-6.

[17] A. Gambi and G. Toffetti. Modeling Cloud performance with Kriging. In *Proceedings of the 2012 34th International Conference on Software Engineering (ICSE)*, pages 1439–1440, June 2012.

[18] O. Greevy, M. Lanza, and C. Wysseier. Visualizing Live Software Systems in 3D. In *Proceedings of the 2006 ACM Symposium on Software Visualization*, SoftVis '06, pages 47–56, 2006. ISBN 1-59593-464-2.

[19] L. Heilig and S. Voss. A Scientometric Analysis of Cloud Computing Literature. *IEEE Transactions on Cloud Computing*, 2(3):266–278, July 2014. ISSN 2168-7161.

[20] J. Humble and D. Farley. *Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition, 2010. ISBN 0321601912, 9780321601919.

[21] M. Hüttermann. *DevOps for Developers*. Apress, 2012.

[22] A. Iosup, S. Ostermann, N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. Performance Analysis of Cloud Computing Services for Many-Tasks Scientific Computing. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):931–945, June 2011. ISSN 1045-9219.

[23] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu. Understanding and Detecting Real-world Performance Bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 77–88, 2012. ISBN 978-1-4503-1205-9.

[24] F. Khomh, T. Dhaliwal, Y. Zou, and B. Adams. Do Faster Releases Improve Software Quality?: An Empirical Case Study of Mozilla Firefox. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, MSR '12, pages 179–188, 2012. ISBN 978-1-4673-1761-0.

[25] R. Kohavi, R. M. Henne, and D. Sommerfield. Practical Guide to Controlled Experiments on the Web: Listen to Your Customers Not to the Hippo. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '07, pages 959–967, 2007. ISBN 978-1-59593-609-7.

[26] P. Leitner and J. Cito. Patterns in the Chaos - a Study of Performance Variation and Predictability in Public IaaS Clouds. *ArXiv e-prints*, 2014.

[27] R. Lemma and M. Lanza. Co-Evolution As the Key for Live Programming. In *Proceedings of the 1st International Workshop on Live Programming*, LIVE '13, pages 9–10, 2013. ISBN 978-1-4673-6265-8.

[28] Y. Liu, A. H. Ngu, and L. Z. Zeng. QoS Computation and Policing in Dynamic Web Service Selection. In *Proceedings of the 13th International World Wide Web Conference on Alternate Track Papers &Amp; Posters*, WWW Alt. '04, pages 66–73, 2004. ISBN 1-58113-912-8.

[29] D. C. Luckham. *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems*. Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0201727897.

[30] M. Mao and M. Humphrey. Auto-scaling to Minimize Cost and Meet Application Deadlines in Cloud Workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 49:1–49:12, 2011. ISBN 978-1-4503-0771-0.

[31] C. Marquezan, D. Bruneo, F. Longo, F. Wessling, A. Metzger, and A. Puliafito. 3-D Cloud Monitoring: Enabling Effective Cloud Infrastructure and Application Management. In *Proceedings of the 2014 10th International Conference on Network and Service Management (CNSM)*, pages 55–63, Nov 2014.

[32] S. McDirmid. Living It Up with a Live Programming Language. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, pages 623–638, 2007. ISBN 978-1-59593-786-5.

[33] S. McDirmid. Usable Live Programming. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, pages 53–62, 2013. ISBN 978-1-4503-2472-4.

[34] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical Report 800-145, National Institute of Standards and Technology (NIST), Gaithersburg, MD, September 2011. URL `http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf`.

[35] S. Meng, L. Liu, and T. Wang. State Monitoring in Cloud Datacenters. *IEEE Transactions on Knowledge and Data Engineering*, 23(9), Sept 2011. ISSN 1041-4347.

[36] A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Comprehensive QoS Monitoring of Web Services and Event-based SLA Violation Detection. In *Proceedings of the 4th International Workshop on Middleware for Service Oriented Computing*, MWSOC '09, pages 1–6, 2009. ISBN 978-1-60558-848-3.

[37] P. Newman. *Building Microservices*. O'Reilly, 2015.

[38] C. Y. Park. Predicting Program Execution Times by Analyzing Static and Dynamic Program Paths. *Real-Time Systems*, 5 (1):31–62, Mar. 1993. ISSN 0922-6443.

[39] F. Rosenberg, C. Platzer, and S. Dustdar. Bootstrapping Performance and Dependability Attributes of Web Services. In *Proceedings of the International Conference on Web Services (ICWS)*, pages 205–212, Sept 2006.

[40] A. Shieh, S. Kandula, A. Greenberg, and C. Kim. Seawall: Performance Isolation for Cloud Datacenter Networks. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, 2010.

[41] C. U. Smith and L. G. Williams. Software Performance Antipatterns. In *Proceedings of the 2Nd International Workshop on Software and Performance*, WOSP '00, pages 127–136, 2000. ISBN 1-58113-195-X.

[42] G. Spreitzer and C. Porath. Creating Sustainable Performance, 2012. ISSN 0017-8012.

[43] M. Turner, D. Budgen, and P. Brereton. Turning Software into a Service. *Computer*, 36(10):38–44, Oct. 2003. ISSN 0018-9162.

[44] A. van Hoorn, J. Waller, and W. Hasselbring. Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ICPE '12, pages 247–248, 2012. ISBN 978-1-4503-1202-8.

[45] V. S. Verykios, E. Bertino, I. N. Fovino, L. P. Provenza, Y. Saygin, and Y. Theodoridis. State-of-the-art in Privacy Preserving Data Mining. *SIGMOD Record*, 33(1):50–57, Mar. 2004. ISSN 0163-5808.

[46] A. Wert, M. Oehler, C. Heger, and R. Farahbod. Automatic Detection of Performance Anti-patterns in Inter-component Communications. In *Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures*, QoSA '14, pages 3–12, 2014. ISBN 978-1-4503-2576-9.

PeerJ PrePrints | https://dx.doi.org/10.7287/peerj.preprints.985v1 | CC-BY 4.0 Open Access | rec: 15 Apr 2015, publ: 15 Apr 2015