

Master Thesis

August 7, 2015

Feedback Driven Development

Predicting the Costs of Code Changes
in Microservice Architectures
based on Runtime Feedback

Emanuel Stöckli

of Hofstetten-Flüh, Switzerland (08-727-836)

supervised by

Prof. Dr. Harald C. Gall

Dr. Philipp Leitner



University of
Zurich^{UZH}



Master Thesis

Feedback Driven Development

Predicting the Costs of Code Changes
in Microservice Architectures
based on Runtime Feedback

Emanuel Stöckli



University of
Zurich ^{UZH}



Master Thesis

Author: Emanuel Stöckli, emanuel.stoeckli@gmail.com

Project period: February 09, 2015 - August 09, 2015

Software Evolution & Architecture Lab

Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Prof. Dr. Harald Gall for giving me the opportunity to write this Master's Thesis at the software evolution and architecture lab at the University of Zurich. My deepest gratitude goes to my advisor, Dr. Philipp Leitner. Thank you for all the valuable feedback during the meetings and also for the additional reviews in the end phase of the project. Furthermore, I want to thank Jürgen Cito for introducing me to the topic of feedback-driven development and all interview participants for the inspiring conversations.

Moreover, I am grateful for the excellent workplace in the s.e.a.l lab, which would not have been the same without the helpful inputs of Joel Scheuner and Genc Mazlami. I would like to thank in particular Christian Bosshard who helped me a lot with insights he gained in his Master's thesis, which also served as a source for my thesis.

Also, many thanks go to Alexander Mülli and all other ifi fellows for the coffee and lunch breaks we had during this six months.

Last but not least, I would like to thank my parents, Kurt and Franziska and my brother Raphael for always supporting and encouraging me. Also, I want to express my heartfelt thanks to my dear girlfriend Jeannette.

Abstract

Building highly scalable and automatized software systems is fostered by Cloud Computing and Microservice Architectures. While loosely-coupled and independently scalable services in the cloud are charged according to actual resource consumption, costs of operation can often be traded against performance. On the contrary, autoscaling is often used to keep the response time on a desired level, while scaling resources up and down. Understanding such complex systems and assessing the costs of operation is difficult for developers. The central issue is the lack of tools. While a plethora of tools bring static code analysis to the IDE, dynamic runtime feedback from production systems is limited to performance-related metrics. We developed a novel approach that brings cost-related metrics into the IDE of developers. The so-called CostHat consists of (1) the *Analytical CostHat* that maps dynamic information about a microservice such as the number of incoming requests or running instances to the corresponding service method and (2) the *Predictive CostHat* that predicts the impact of modifying the number of requests to another microservice on its costs of operation. A prototype was implemented and evaluated in a semi-structured interview study that investigated the relationship of software engineers to costs of operation and how they perceive the CostHat tooling. Our findings indicate that costs currently treated in a reactive rather than proactive manner. Also, it is indicated that the overall understanding of the system can be increased by the Analytical CostHat and developers are driven to optimize by the Predictive CostHat. While a serious overhead in build times is shown in the quantitative evaluation, results from the interview study indicate that developers are more concerned about the monitoring overhead.

Zusammenfassung

Skalierbare und automatisierte Softwaresysteme werden durch Cloud Computing und Microservice Architekturen begünstigt. Geld kann jederzeit gegen mehr Ressourcen, sprich mehr Leistung eingetauscht werden, da solch lose gekoppelte und eigenständig skalierbare Services in der Cloud nach dem tatsächlichen Ressourcenverbrauch abgerechnet werden. Im Gegenzug kann automatische Skalierung genutzt werden, um die Reaktionszeit von Services auf einem gewünschtem Niveau zu halten, während dem gebuchte Ressourcen hoch- oder runterskaliert werden. Das Verständnis von solch komplexen Systemen und die Beurteilung von Betriebskosten ist schwierig für Software Entwickler. Das zentrale Problem ist der Mangel an Tools. Eine Vielzahl von Tools bringen statische Code-Analyse in die Entwicklungsumgebungen aber dynamisches Runtime Feedback aus produktiven Systemen beschränkt sich auf Metriken, welche auf die Performance ausgerichtet sind. Wir haben einen neuartigen Ansatz entwickelt, welcher finanzielle Metriken in die Entwicklungsumgebung bringt. Der Ansatz, auch CostHat genannt, besteht aus (1) dem *Analytical CostHat*, welcher dynamische Informationen wie zum Beispiel die Anzahl der laufenden Instanzen oder eingehenden Anfragen zur entsprechenden Microservice-Methode darstellt und (2) der *Predictive CostHat*, welcher die Auswirkungen einer veränderten Anzahl an Anfragen an einen Drittservice auf die gesamten Betriebskosten berechnet. Ein Prototyp wurde entwickelt und in einer semi-strukturierten Interview Studie ausgewertet. Diese untersucht das Verhältnis von Software Entwicklern zu Betriebskosten von Software und erforscht wie die Teilnehmer den Prototypen wahrnehmen. Unsere Ergebnisse zeigen, dass Kosten zur Zeit vor allem rückwirkend und nicht vorausschauend behandelt werden. Auch deuten die Ergebnisse darauf hin, dass das Gesamtverständnis des Systems durch den Analytical CostHat erhöht werden kann und Software Entwickler durch den Predictive CostHat dazu angetrieben werden zu optimieren. Während dem die quantitative Auswertung aufzeigt, dass der Build-Prozess bedeutend länger dauert, deuten Ergebnisse aus der Interviewstudie darauf hin, dass Software Entwickler mehr über den Overhead der Monitoring Komponente besorgt sind.

Contents

1	Introduction	1
1.1	Motivation and Research Questions	1
1.2	Structure of the Thesis	2
2	Background	5
2.1	Cloud Computing	5
2.1.1	Types and Service Models	6
2.1.2	Costs of Cloud Computing Resources	6
2.1.3	Characteristics	9
2.2	Microservice Architectures	11
2.2.1	Definition of a Microservice	11
2.2.2	Characteristics	12
2.2.3	Advantages and Disadvantages	16
2.3	Feedback-Driven Development	18
2.4	Implications for This Work	20
3	Related Work	21
3.1	Capturing Feedback Input Data	21
3.1.1	Capturing Runtime Monitoring Data	21
3.1.2	Capturing Code Changes	22
3.2	Predicting Costs of Code Changes	23
3.3	Bringing Software Metrics to the Developer	24
3.3.1	Visualization	24
3.3.2	Integrating Feedback into the Workflow of Developers	26
4	The Concept of Bringing Cost-Relevant Metrics to the Developer	29
4.1	Explaining the Rationale	29
4.2	Requirements	31
4.3	CostHat in Detail	32
4.3.1	Formal Model	32

4.3.2	Architectural Overview	34
4.3.3	Capturing the Context of a Target Microservice	36
4.3.4	Creating Value for the Developer	38
5	Implementation Details	43
5.1	Towards an FDD Framework	43
5.2	Monitoring Component	44
5.2.1	Aspect-oriented Monitoring Approach	44
5.2.2	Contributions	45
5.3	Server-Side Backend	48
5.3.1	Making Use of the Existing Feedback-Handler	48
5.3.2	Contributions	49
5.4	Eclipse Plugin	52
5.4.1	Introducing the FDD Base Plugin	53
5.4.2	Setting up the Plugin	54
5.4.3	Detecting Code Changes	56
5.4.4	Preparing and Predicting Dynamic Content	57
5.4.5	Making the Content Visible	60
6	Evaluation	63
6.1	Demo Application	63
6.2	Interview Study	65
6.2.1	Study Methodology	65
6.2.2	Status Quo	66
6.2.3	Features of the CostHat FDD Tooling	68
6.2.4	Visual Representation of the CostHat FDD Tooling	71
6.2.5	Integration of the CostHat FDD Tooling	73
6.2.6	Overhead of the CostHat FDD Tooling	74
6.2.7	Roundup	75
6.3	Quantitative Evaluation	76
6.3.1	Methodology	76
6.3.2	Results	76
6.3.3	Discussion	77
7	Closing Remarks	79
7.1	Conclusion	79
7.2	Future Work	80
A	Acronyms	83

B Attachments	85
B.1 Demo Application Guide	85
B.2 Interview Study Questionnaire	91
B.3 Interview Study Results	93
B.4 Quantitative Evaluation	105
B.5 CD Contents	109

List of Figures

2.1	Different Facets of Cloud Resource Pricing	7
2.2	Communication Patterns: Server-Side and Client-Side Load Balancer	14
2.3	Communication Patterns: Orchestration and Choreography using the example of a message-oriented middleware (influenced by [New15])	15
2.4	Communication Patterns: Communication via Gateway or Proxy (influenced by [NSS14])	15
2.5	Principles of Feedback-Driven Development (taken from [Bos15])	19
2.6	FDD in a Continuous Delivery pipeline (taken from [Bos15])	19
3.1	UML Communication Diagram (taken from [Fow04])	26
3.2	Call relations in a circular view (taken from [CHZ ⁺ 07])	26
3.3	Calling Context Ring Charts (taken from [MBV ⁺ 10])	26
4.1	Explaining the Rationale of the CostHat Concept	29
4.2	Requests ordered along a timeline and divided into periods	33
4.3	Architectural Overview with System and Environment Components	35
4.4	Mockup of the Method Declaration	38
4.5	Mockup of the Client Invocation Visualization	40
4.6	Mockup of the Client Invocation What-If Scenario Visualization	41
5.1	Class Diagram of interfaces relevant to filter/aggregate requests between services	48
5.2	Class Diagram of RequestCollector, RequestAggregationStrategy and RequestAggregationValues	50
5.3	FDD Base Plugin as a dependency of the CostHat and the PerformanceHat	53
5.4	Activate/Deactivate CostHat for a certain project	55
5.5	Feedback Handler Client within the CostHat Plugin that allows to register new target microservices	55
5.6	CostHat plugin properties for time range, request interval and feature toggling	55
5.7	Class Diagram of classes and interfaces relevant for cost predictions	58
5.8	Screenshot of the CostHat plugin: prediction of the cost impact	59
5.9	Screenshot of the CostHat plugin: method of a ProductController enriched by the Analytical CostHat	61
6.1	Demo Microservice Architecture (arrows represent request flows)	64
6.2	Avg. Build Times of CostHat Builder with varying number of client invocation annotations	77
6.3	Avg. Build Times of CostHat Builder with varying number of method declaration annotations	77

List of Tables

2.1	Advantages of Microservice Architectures	16
2.2	Disadvantages of Microservice Architectures	17
6.1	Average Build Times of Java/Maven Builder versus CostHat Builder	77

List of Listings

5.1	Information about the procedure and a current timestamp is collected at every join point	45
5.2	ProcedureCallJoinPoint	46
5.3	Annotations of each joint point are mapped to the procedure domain model	47
5.4	Constants related to HTTP headers	51
5.5	CostHat Participants harness the overloaded visit method of an ASTVisitor	56
5.6	ContextBuilder that collects dynamic content before it is passed to the template engine	57
6.1	Unix shell commands to debug Eclipse builders	76

Introduction

Cloud computing enables increased scalability, business agility and faster innovation [BFKB⁺14], while huge operating efforts and large capital investments are decreasing [AFG⁺09]. As operating software in the cloud becomes easier, the process of building and operating software tends to be more and more aligned [CLG⁺15], which fosters evolutions like Continuous Delivery and DevOps. But faster delivery and deployment often conflicts with monolithic large-scale software systems, where updates are accumulated and the deployment of the whole system has to proceed all in one. This encourages the need for more service-oriented systems of small and loosely coupled services allowing for more independent scaling and deployment. Microservice Architectures fulfil those needs while increasing complexity and raising the challenge to understand the system as a whole [CZG11]. Also, performing tests and understanding the impact of code changes becomes more difficult. Feedback-Driven Development (FDD) creates an additional form of back-coupling by bringing runtime monitoring data directly from the production system into the daily workflow of a software developer [CLG⁺15]. Analytical FDD directly provides the developer with runtime information from production and predictive FDD goes even beyond by using this runtime information together with local code changes as input to predict the impact. This enables developers to better assess the impact of their code changes before deployment and prevents undesired implications. Bosshard [Bos15] applied the concept of FDD on performance metrics, which reveals possible performance bottlenecks. This thesis aims at bringing cost-relevant metrics to the developer and pointing out code changes that cause an increase of costs of operation.

1.1 Motivation and Research Questions

Imagine the following notional scenario in the context of a microservice architecture that runs in the cloud: software developer Joe is responsible for the whole lifecycle of a frequently accessed microservice *A*; he adds a few lines of code, which includes new requests to another microservice *B*. Everything works fine, service *B* indeed provides the desired response. In the meanwhile, new instances are started based on autoscaling units in order to keep the desired response time constant despite the fact that the number of incoming requests to service *B* has been doubled

due to Joe's requests. Maybe Joe would have implemented the communication flow in another way if he would have been aware of this impact. Code changes of a single microservice can have an enormous impact on other microservices. Either performance decreases, in other words response times increase, or the number of instances increases. Notional scenarios like this are quite reasonable because ongoing hypes like cloud computing and microservices foster scalability and automatization. To date, feedback-driven development is mainly focused on providing performance data and estimates to the developer. Thus, today's software development environments provide no support in assessing the situation described in the scenario before. This is used as spawn point of this work and leads to the first research question:

Research Question 1

How can cost-related metrics be integrated into software development environments in order to enable software developers to assess the financial impact of their code changes?

A systematic study conducted by Cito et al. not only investigates how professional software developers build applications but also on how they use tools and data on top of cloud computing [CLFG14]. Among other things, this study included financial aspects such as the costs of operation. They found that although developers consider cost aspects to be important, it is not a tangible factor in their everyday life. They further conclude that methods and tools for developers will need to adapt to new requirements of developing applications for the cloud. Assuming that cost-relevant metrics are available in software development environments, there is still some doubt if developers are interested in such metrics at all and how they perceive such metrics. Hence, a second research question can be derived:

Research Question 2

How do software developers perceive the availability of cost-relevant metrics within a software development environment?

1.2 Structure of the Thesis

In the following, Chapter 2 enables the reader to understand the subject of this work and is structured according to the thesis title "*Feedback Driven Development (iv): Predicting the Costs (iii) of Code Changes in Microservice Architectures (ii) based on Runtime Feedback (i)*".

- (i) *Runtime Feedback* indicates that we focus on software systems that already have been deployed to a production environment and are equipped with a monitoring component which

provides us with runtime data. Those preconditions are fulfilled in cloud computing environments and therefore we further elaborate this topic in Section 2.1.

- (ii) *Microservice Architectures* and its implications are examined in Section 2.2.
- (iii) *Costs* arising from the use of cloud resource are highlighted in Section 2.1.2.
- (iv) *Feedback Driven Development* is a field of research that is concerned with bringing those runtime metrics to the software developer. Its software is based on insights from the production environment and minimizes the effort needed to get those insights.

Thereupon Chapter 3 discusses related work and demonstrates how this work differs. Chapter 4 reveals the main contribution of this work consisting of a new conception that brings cost aspects into software development environments. In order to answer the research questions defined above, this concept has been implemented with a prototype presented in Chapter 5. Afterwards, this prototype has been used to evaluate the concept in Chapter 6. The work ends with final remarks in Chapter 7.

Background

2.1 Cloud Computing

Cloud Computing has changed the way computing is practiced. Cloud providers manage hardware, allocate resources and provide them to cloud users, which again, can offer services to their end users [JS14]. Since computing became an utility, large capital outlays in hardware and operation staff expenses are past for direct customers of cloud providers [AFG⁺09]. Beside financial reasons, the improvement of business agility and faster innovation are motives to adopt cloud computing [BFKB⁺14]. This work follows the definition of the National Institute of Standard and Technology:

“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.”

– Mell and Grance (2011), *The National Institute of Standards and Technology*, [MG11b]

It is also adopted by [ZCB10] and is widely accepted (cited by 2994 others according to Google Scholar on June 30, 2015).

The mindset is changing as well, [AFG⁺09] reports three arising aspects:

1. "The illusion of infinite computing resources on demand, ..."
2. "The elimination of an up-front commitment by Cloud users, ..."
3. "The ability to pay for use of computing resources on a short-term basis ..."

This has led to new service and pricing models.

2.1.1 Types and Service Models

Usually two types of clouds are distinguished, the *Public Cloud*, which is available to the general public and internal datacenters of organizations as *Private Clouds* [AFG⁺09]. Both, users of public and private clouds choose the desired service model with resources they want to access in the cloud. There are three frequently mentioned (e.g., [MG11b], [RCL09] and [YBDS08]) service models with different levels of abstraction:

- **Software-as-a-Service (SaaS):**
provisioning of software as cloud application that is deployed at the providers computing infrastructure [YBDS08]. Users usually just access the application.
- **Platform-as-a-Service (PaaS):**
provisioning of platform layer resources, including operating system support and software development frameworks [ZCB10]. Users usually just deploy their application.
- **Infrastructure-as-a-Service (IaaS):**
provisioning of infrastructural resources, usually in terms of virtual machines [ZCB10].

Recently, containerization implementations like Docker¹ or Warden² became more popular. Docker containers build a wrapper around software that encapsulate a complete filesystem, which is nicely separated but still contains everything needed to run the software anywhere [Doc15]. Dua et al. [DK⁺14] already mentions containers as a new type of cloud service, the so-called Containers-as-a-Service (CaaS). Containers lie between IaaS, which provide virtualized compute resources, and PaaS, which provide application specific runtime services [PDCB15]. Compared to virtual machines they still offer application isolation of network, computation and storage but have performance improvements and reduced startup time because not a full operating system is running [DK⁺14]. Containers are considered as the future of commoditizing application deployment and scalability [Kra14].

2.1.2 Costs of Cloud Computing Resources

The most common entry point into costs of cloud computing resources might be the claim that there is no difference between using 100 servers for one hour and using one server for 100 hours. While this creates huge incentives to run batch jobs in the cloud [AFG⁺09], this work rather has its focus on microservice architectures that are deployed in the cloud and therefore make continuously use of cloud resources.

From a cloud customer's point-of-view, arising costs depend on the pricing strategy of the respective service provider. Compared to the pricing of traditional software with limitless usage, the pricing of cloud computing resources is more flexible and thus various service providers having their own pricing strategies and charging structures [SP11]. Nevertheless, prevalent providers

¹<https://www.docker.com/>

²<https://github.com/cloudfoundry/warden>

use to bundle cloud resources (Amazon [Ama15b], Rackspace [Rac15], Microsoft Azure [Mic15] and DigitalOcean [Dig15]). Bundles are often optimized for certain use cases (general purpose, memory optimized, I/O optimized, compute optimized, etc.), which allows the customer to select a base quality of service and adjusts single resources in order to get the desired quality of service (e.g., adding more storage to the bundle). They often provide a variety of additional services like load balancers, the management of images or security services that are offered beside the typical cloud resources like computation (clock hour of server time), storage or data transfer.

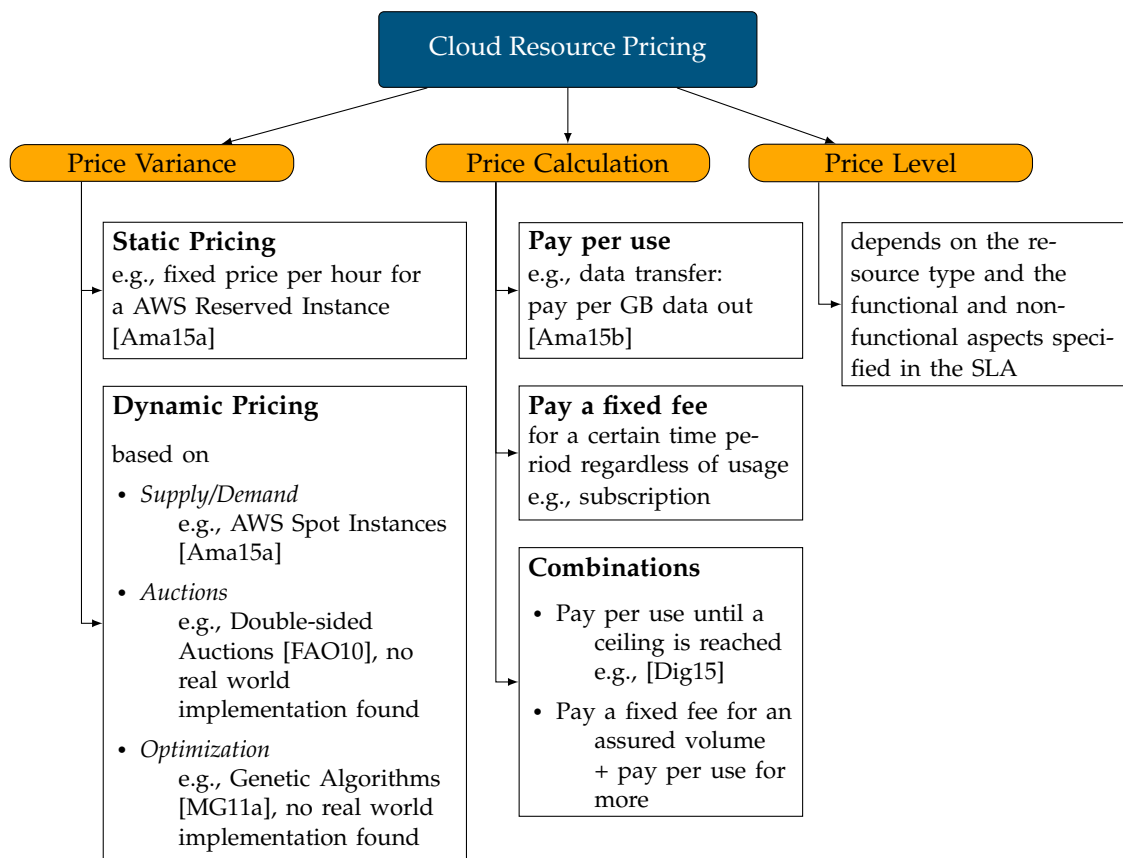


Figure 2.1: Different Facets of Cloud Resource Pricing

In the end, the sum of all costs for allocated resources and services will result in the total price that a cloud user has to pay [ARAEB13]. Collecting all costs that arise from leasing different resources with different bases of calculation tend to be confusing, therefore present a clean overview of important facets of cloud resource pricing in Figure 2.1.

Price Level

The customer and the cloud provider usually agree on a service-level agreement (SLA), which defines all functional and non-functional aspects of the offered service in formal and quantitative terms [JS14]. Both, functional and non-functional aspects influence the price level. Functional and resource-dependent aspects such as the quality and capacity of resources (which cpu, how much memory etc.), the age of the resources and the service provider's initial costs for the resources are influential factors [ARAEBA13]. On the other side, responsibilities and commitments of both, customer and cloud provider influence the price as well [ARAEBA13]. A typical example is special prices for long-term commitment, e.g., Microsoft Azure [Mic15] offers more attractive rates for organizations that are willing to make an upfront monetary commitment that is thereupon consumed throughout the year by using any combination of the Azure cloud services. Amazon [Ama15b] has a similar offer called "Amazon EC2 Reserved Instances".

Price Variance: Static versus Dynamic Pricing

A pricing strategy of a service provider can be either static or dynamic, where in the later the price charged changes dynamically [ARAEBA13]. Today, static pricing is still the norm although dynamic pricing has considerable advantages for service providers such as more revenue and a better utilization of unused capacity because the price influences the demand [XL12]. An example for dynamic pricing in practice is Amazon [Ama15b], which offers spare EC2 instances as "Spot Instances" based on supply and demand. While such simple dynamic pricing models are indeed tested in practice, more complex pricing models discussed in research, are often limited to theory [ARAEBA13]. Samini et al. [SP11] captures auction-based pricing approaches such as double-sided auctions where both, service providers and customers, have to bid in order to settle the price.

Others address competitive markets where service providers do not know the pricing strategy of others, e.g., a genetic algorithm that optimizes the pricing function is introduced by [MG11a]. It boosts earnings up to 100% compared to utility-based dynamic pricing providers and up to 1000% more than static prices. From a customer's point of view dynamic pricing might hamper cost prediction and estimation. [STT⁺12] proposes the usage of financial option theory and treats cloud resources as call options. On the other hand, advantages may appear as well. For example [JTB11] observed Amazon EC2 spot prices and found the decreasing in prices on weekends.

Calculation Methods

Cloud service providers usually price their products on a per-use basis, which means that cloud users are charged according to their resource consumption [ARAEBA13]. E.g., every Amazon EC2 instance type has a price per hour and the user pays the number of hours he used this instance [Ama15b]. Beside consumption-based prices, there are fixed fees such as subscriptions, in which the user pays a fixed unit price for a certain period of time (e.g., per month) [ARAEBA13]. One might think of different combinations of the models above. For example Digitalocean [Dig15]

offers five virtual machine plans that are paid by hour of usage but as soon as the cloud user reaches a certain cap the costs will be fixed to a monthly subscription fee.

2.1.3 Characteristics

Scaling and Load Balancing

Cloud computing enables dynamic resource provisioning and de-provisioning as required [HKR13]. Depending on the scaling behaviour of all affected applications, cloud computing facilitates to start small and scale up and down in case of rising and falling demand [ZCB10]. Scaling can be handled manually or automatically, but the decision how to scale is always imminent.

Horizontal versus Vertical Scaling based on [VRMB11]. Horizontal scaling is about increasing or decreasing the number of instances, e.g., the number of virtual machines in the context of cloud computing. Theoretically, it is also possible to scale horizontal by varying the number of servers or containers. When multiple instances are available, incoming requests have to be assigned to instances somehow. For this reason the entry point of an application is often a load balancer, which routes the load to available instances (e.g., the Elastic Load Balancer³ of Amazon). Second, vertical scaling will change a current entity, e.g., resizing resources like the CPU or the memory of a running instance.

Scale Cube based on [AF15]. The AKF Scale Cube with its x-axis, y-axis and z-axis introduced in [AF15] is an alternative perspective on scaling. As expected, x-axis scaling has the same meaning as horizontal scaling as mentioned above. But y-axis scaling differs from vertical scaling. In this context, y-axis scaling is about functional decomposition and leads to a separation of responsibilities or data meaning among multiple entities [AF15]. Furthermore, z-axis scaling leads to a split of responsibilities based on certain values like the country or customer segment. Especially y-axis scaling is interesting to look at when it comes to Microservice Architectures where separation of concerns is a major pillar (see Section 2.2).

Faster Releases

There is an ongoing evolution of open and closed source organizations towards faster release cycles and reduced time-to-market [MKA⁺13]. This often goes along with establishing Continuous Integration, Continuous Delivery or even Continuous Deployment, which relies on a high degree of automatization [Swa14].

“Continuous Delivery, as the name suggests, is a way of working whereby quality products, normally software assets, can be built, tested and shipped in quick succession thus delivering value much sooner than traditional approaches”

³<http://aws.amazon.com/elasticloadbalancing>

– Swartout (2014), [Swa14]

This automated process is often represented in a so-called Deployment Pipeline [HF10]. It starts with pushing changes to the version control and ends when it reaches the end user. Continuous Delivery quickly goes through the pipeline and ensures that every change could be deployed, where Continuous Deployment goes one step beyond by actually deploying every change. To link everything together, cloud computing is by no means prerequisite to implement faster releases but it eases automated provisioning and encourages the adoption of Continuous Delivery [Swa14]. Only working weeks, days or even hours to release a feature to the customers rather than months, as practiced before, also means getting faster feedback [MKA⁺13]. This serves as a basis to discuss the role of the contribution of this work later on.

DevOps

Automation does not only lead to faster releases and reduced incidence of errors, but also influences the responsibilities of the operations team [BWZ15].

“DevOps is a way of working whereby developers and IT system operators work closely, collaboratively, and in harmony towards a common goal with little or no organizational barriers or boundaries between them”

– Swartout (2014), [Swa14]

Development teams used to aim at delivering software as rapidly as possible, whereas operations teams wanted stability [HF10]. Also, most projects fail due to people problems rather than technical problems [HF10]. Thus, the accomplishment of DevOps tasks that range from development and delivery to maintenance and support by software developers removes a significant coordination step from the deployment process [BWZ15]. The ease provisioning of cloud computing encourages and facilitates the adoption of DevOps [Swa14].

Considering the emergence of DevOps and changing responsibilities of developers, then, the integration of cost-relevant metrics based on runtime feedback seems more fruitful than ever.

2.2 Microservice Architectures

This chapter aims at pointing out the view on microservices on which the presented work is based. This is especially important because it is a fast-moving area and this chapter should enable the reader to follow the thoughts of this work even though the view on microservices might change in the future.

2.2.1 Definition of a Microservice

The term "microservice" is a wide-spread buzzword. A plethora of definitions exist in both, books and the web but definitions in research are scarce. However, along similar definitions, [New15] considers Microservices as a distributed system of collaborating and finely grained services that have their own lifecycles. Later he comes up with the following concise definition:

"Microservices are small, autonomous services that work together."

– Newman (2015), *O'Reilly Media, Inc.*, [New15]

Klang uses a broader approach:

"A Microservice is a discrete, isolated, and named piece of logic that consumes 0N inputs and produces 0N outputs and which is executed for the benefit of an invoker - it is performed as a service."

– Klang (2014), [Kla14]

Discrete in this context means that a service has only one single responsibility and is therefore called micro. It does one thing and does it well. It is *isolated* because it is separately operated, maintained and also fails in isolation. It has to be *named* because the location of the physical hardware does not matter as long as we can refer to the service [Kla14].

[NSS14] uses other words, which basically mean the same but in contrast he puts great emphasis on "well-defined interfaces" between those microservices. This well-defined interfaces can be interpreted as application of the Liskov Substitution Principle [Mar03] on Microservices: a service is replaceable as long as the API remains the same.

"A micro-service is a lightweight and independent service that performs single functions and collaborates with other similar services using a well-defined interface"

– Namiot and Sneps-Sneppé (2014), [NSS14]

Breaking down those definitions and grouping all parts with the same meaning back leads to four main pillars that describe the view of this work on microservices:

1. **Size:**
small [New15], fine-grained [New15], lightweight [NSS14], micro (derived from the term "Microservice")
2. **Purpose:**
discrete [Kla14], performing a single function [Kla14], single responsibility (derived from [Mar03])
3. **Coupling:**
autonomous [New15], isolated [Kla14], own lifecycles [New15], independent [NSS14], decoupled [New15]
4. **Communication:**
collaborate together [NSS14], work together [New15], executed for the benefit of an invoker [Kla14]

2.2.2 Characteristics

Size and Purpose

If it comes to granularity, the question remains how small a single Microservice should be. Microservice Architectures have been described as fine grained service-oriented architectures and also compared to the Unix philosophy of small and simple programs [Fow14]. We therefore deduce that a specific size or number of lines of code are irrelevant, it is more about the simplicity and the specific dedication of a service.

[Mar03] describes the *Single Responsibility Principle (SRP)* of classes in the context of Object-oriented programming. He considers one single responsibility to be one reason for change. Applied to Microservices this means that a service should only do exactly one thing. [New15] states that for a given piece of functionality it should be obvious for everyone where this code lives and it should be possible to rewrite it within weeks. The pace of change, different scaling characteristics, a heterogeneous team structure or a different degree of sensitivity of the underlying data are all reasons to further split an application into separate microservices [New15].

Coupling

A Microservice should be maximally cohesive and loosely coupled with the result that it can be changed without changing other services [New15]. [Fow14] propounds the view of dumb pipes and smart endpoints, which receive requests, apply some logic and produce a response. A decoupled service allows us to change it and independently deploy it right away whenever we want. This might often only work in theory because more than one service is affected but there are still advantages compared to monolithic applications, which are usually deployed as a single

package. All components are deployed together and as soon as a rollback of one broken feature is required, changes to all components are affected.

Communication

Microservices need to communicate among themselves and with the outer world, usually based on some lightweight mechanisms [NSS14]. Relating to client-to-service communication, both, synchronous or asynchronous communication over HTTP is quite common (e.g., REST). Apart from that, server-to-server communication often requires lower latency and smaller messages and therefore relying on HTTP (and consequently TCP) does not fit very well [New15]. Other service-to-service communication approaches are under discussion, e.g., Docker open sourced an ultra-lightweight networking library that offers transport based on in-memory Go channels, Unix sockets, Raw TCP, TLS, HTTP2/SPDY and Websockets⁴. Also, many Remote Procedure Call (RPC) Frameworks can run on top of different networking protocols such as UDP [New15]. But still, REST over HTTP seems to be the default choice for service-to-service interactions [New15].

For the remainder of this work, a Microservice Architecture is seen as architectural style that does not imply how communication mechanisms are implemented (neither specific protocols nor serialization formats such as JSON, XML, Avro Thrift or Google Protocol Buffers can be assumed). This decision depends on the context (e.g., what delay can be tolerated?) and on how those services are going to be deployed (e.g., is it the same physical machine?). Furthermore independent implementation decisions are one of the biggest advantages of microservices (see Section 2.2.3). It follows that developers of a two particular services should have the opportunity to decide how they exchange data.

For this reason we continue in more abstract terms by looking at communication patterns that often occur. As commonly known, communication between two services can be...

- direct (one hop from a service to another) or indirect (more than one hop)
- synchronous (blocks until completed) or asynchronous (operations continue)

Handling multiple instances. Oftentimes multiple instances of a particular service are deployed for several reasons such as performance, reliability or testing [BWZ15]. As soon as two instances of a service are deployed, direct communication quickly leads to questions like "How do I identify other services?", "Where is it located?", "Which instance should be called?" or "Does another service respond if one service fails?". Demand for a service registry/discovery, failure detection and load balancing arises.

The most simplistic approach is to directly refer to a load balancer, e.g., the Amazon Elastic Load Balancer⁵, which has the responsibility to decide about the instance that processes the request. This server-side approach is represented on the left side of Figure 2.2.

⁴<https://github.com/docker/libchan>

⁵<http://aws.amazon.com/elasticloadbalancing/>

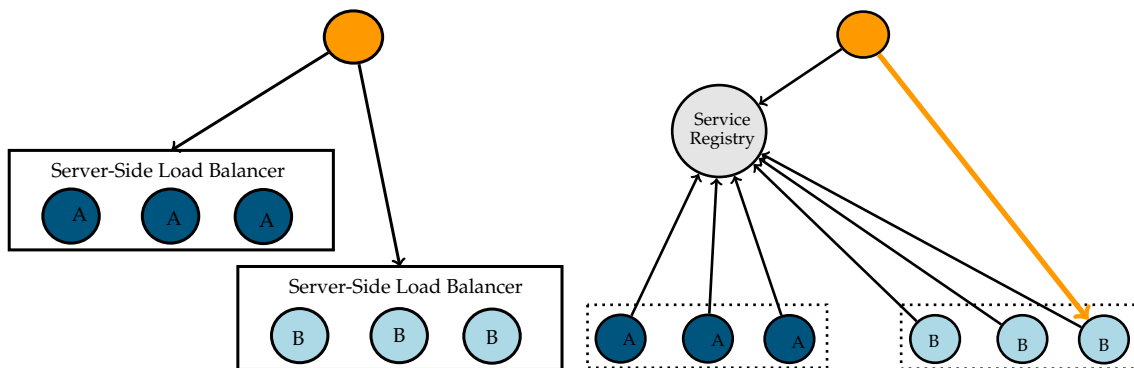


Figure 2.2: Communication Patterns: Server-Side and Client-Side Load Balancer

Often, a load balancer is used together with a Service Registry/Discovery [BWZ15] such as Netflix Eureka⁶ or AirBnB SmartStack⁷ (based on AirBnB Nerve for registry and AirBnB Synapse for discovery). Service instances can register themselves in order to be discoverable and by periodically renewing this registration they indicate that they are still working properly [BWZ15]. The illustration on the right of Figure 2.2 shows a client-side load balancer, where the orange service receives multiple IP addresses. If one instance fails, the client can use the next one. For example Netflix offers a client-side library called Ribbon⁸ that allows clients to still use direct communication with services after initial discovery. There is a variety of other approaches available, e.g., using DNS to return multiple IP addresses or having a registry that acts as a load balancer and redirects requests [BWZ15].

Orchestration versus Choreography. Applying business logic mostly affects multiple microservices, [New15] suggests two ways to handle that: First, the orange service in the first illustration of Figure 2.3 directly communicates with all the three other services that are affected (either synchronously or asynchronously). Sometimes this might be desired but often, this leads to overloaded services with too much business logic.

Choreography is second way to handle this issue: services emit events and on the other side, consuming services need to find out that those events have happened [New15]. Message brokers like RabbitMQ⁹ can offer such functionality. But it is crucial to keep the middleware as lightweight as possible, otherwise history will repeat itself: as [New15] outlines, the problem with Enterprise Service Bus (ESB) was that vendors overloaded their software and pushed a lot of business logic and functionality into the middleware. This conflicts with the principles of a Microservice Architecture.

⁶<https://github.com/Netflix/eureka>

⁷<http://nerds.airbnb.com/smartstack-service-discovery-cloud/>

⁸<https://github.com/Netflix/ribbon>

⁹<https://www.rabbitmq.com/>

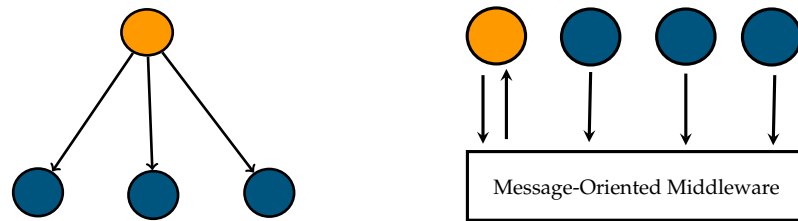


Figure 2.3: Communication Patterns: Orchestration and Choreography using the example of a message-oriented middleware (influenced by [New15])

Communication via Gateway or Proxy. For client-to-service communication, API Gateways, which act as intermediary between clients and backend services are widespread and often used for monitoring, backwards compatibility and caching in order to decrease the amount of remote calls between services [NSS14]. Both, commercial vendors (e.g., apigee¹⁰, 3scale¹¹ or Intel Mashery¹²) and Open-Source projects (e.g., WSO2¹³, strongloop¹⁴ or tyk.io¹⁵) offer a wide range of functionalities: e.g., API management, API documentation, API lifecycle management, monitoring, predictive analytics and statistics, authentication and security, quotas and rate limits, billing.

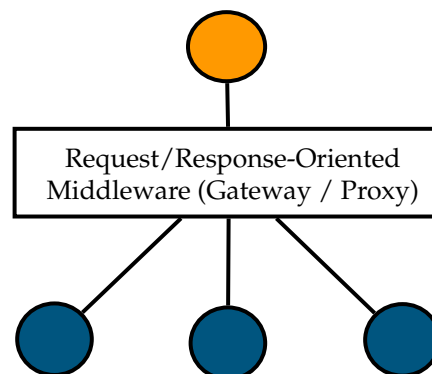


Figure 2.4: Communication Patterns: Communication via Gateway or Proxy (influenced by [NSS14])

No particular communication pattern or mechanism can be derived from microservices. For the remainder of the thesis, this is seen as architectural and implementational detail with the implication that the proposed concept should not require a specific communication pattern such as an API Gateway in order to be functional.

¹⁰<http://apigee.com/>

¹¹<http://www.3scale.net/>

¹²<http://www.mashery.com/>

¹³<http://wso2.com/api-management/try-it/>

¹⁴<https://strongloop.com/>

¹⁵<https://tyk.io/>

2.2.3 Advantages and Disadvantages

First, we capture advantages and disadvantages of Microservice Architectures, afterwards we summarize them in Table 2.1 and 2.2.

Advantages

Advantages	
Scalability	Independently scaling up and down [New15]
Deployment	Separate updates, maintenance and deployment of services [New15] Faster deploys, enables Continuous Delivery [New15]
Technology Choice	Use of programming language, framework, data store that fits the service [New15] Use of hardware that suits the service [NSS14] Removes long term commitment, smaller legacy systems [NSS14]
Understandability	Easier to understand and rewrite services [NSS14] Easier to add and replace developers [NSS14] Unambiguous responsibilities, it is obvious where the code lives [New15]
Fault Isolation and Resilience	Failures stop at service boundaries [New15] Increased resilience because the rest of the system stays alive [New15]
Evaluation	A/B testing of different versions in order to evaluate [HK14] Improved mapping of runtime metrics and user feedback to code changes

Table 2.1: Advantages of Microservice Architectures

Microservices can be independently scaled: scaled up when more resources are needed but also scaled down when they are not used anymore [New15]. As a consequence services with memory intense operations can be deployed on memory optimized R3 Amazon instance types¹⁶ while others stay small. In contrast, large monolithic software systems can only scale as a whole and different application components may have a demand for different resources such as more memory or cpu [NSS14]. Thus, scaling monolithic applications is often more costly but only when microservices are split up according to scaling characteristics [New15].

Furthermore, small services allow multiple developers and teams to update, maintain and deliver relatively independently of each other [New15]. Changes get deployed faster and do not build up and up until a new version is ready [New15]. As mentioned before, cloud-based technologies can encourage the adoption of Continuous Delivery [Swa14]. It follows that Microservices Architectures combined with cloud computing might reinforce Continuous Delivery. This also leads to fast problem isolation and ease rollbacks [New15]. On the contrary, development and deployment efforts in monolithic applications have to be coordinated [NSS14] and changing one line of code requires the whole application to be deployed again [New15].

Both, hardware and software technologies are moving fast and monolithic architectures make it difficult to incrementally adopt new technologies because it is often not possible to replace the

¹⁶<http://aws.amazon.com/ec2/instance-types/>

underlying technology of a single component [NSS14]. However, Microservice Architectures allow to select the right technologies for each service, prevent the use of one-size-fits-all approaches and increase replaceability [New15]. Services that handle network-oriented data can use graph databases whereas others use a relational database.

Also, [NSS14] argues that applications get more difficult to understand and modify as more they grow and thus, it's more difficult to replace developers and include new ones. Microservices remain small by definition and consequently, it should always be obvious where code lives and they should be easily rewritable [New15].

[New15] further states that Microservice Architectures should prevent that everything stops working as soon as one service fails. Both, networks and machines can and will fail and thus service failures need to be handled. If implemented appropriately, this will lead to increases resilience where failures stop at service boundaries while the rest of the system stays alive.

[HK14] state that cloud services enable the so-called A/B testing of multiple versions of services in order to evaluate the impact which specific product changes actually have on customer behavior. This might be even easier and cheaper with small services and therefore we argue that it can be seen as advantage. As mentioned above, Microservice Architectures enable faster and smaller deployments [New15]. It follows that opportunities to map runtime metrics and user feedback to code changes improve.

Disadvantages

Everything has downsides and so have microservice architectures. It is apparent that with an increasing amount of services it might be hard to keep an overview. Inline comments alone are not sufficient because developers might be limited to access code of a particular service. Thus, a clean documentations of the architecture and an overview about public interfaces are needed

Disadvantages	
Increased Complexity	Difficult to keep an overview (own statement)
	Trade-off between consistency, availability and partition tolerance [New15]
	Versioning of services [Kra14]
	Refactoring and moving code across service boundaries [Fow14]
	End-to-end testing is more difficult [New15]
Performance Issues	Debugging and reproducing bugs [LGW ⁺ 08]
	Unreliable networks [RGO06]
	Network latency [RGO06]
Inefficiency of Workflows	Limited bandwidth and transport costs [RGO06]
	Trade-off between sharing code and loose coupling [New15]
	Duplication of efforts (own statement)

Table 2.2: Disadvantages of Microservice Architectures

(Swagger¹⁷ and readme.io¹⁸ might be interesting therefore).

The CAP theorem mentioned in [New15] proves the trade off between consistency, availability and partition tolerance in distributed systems. Two of them need to be kept in failure mode, which increases complexity.

As mentioned above, services are deployed independently and have their lifecycle. The need for versioning of services is created because two dependent services might get deployed independently. As soon as a new version of a service is not backwards compatible and two different versions have to be deployed complexity and operational pain increases a lot [Kra14].

Furthermore moving and refactoring code across service boundaries becomes a challenge due to the coordination overhead for changing interfaces, the need of backwards compatibility and difficulties in testing [Fow14]. Especially end-to-end testing is more complex because it runs against the entire distributed system of microservices and requires multiple services to be deployed [New15]. In addition, debugging and reproducing bugs is more complex because they often appear only after a certain sequence of events, typically involving machine or network failures [LGW⁺08].

There are quite a few performance challenges known from distributed systems and derived from the eight fallacies of distributed computing [RGO06]: the network is not reliable, there is always latency, the bandwidth is limited, transport costs are not zero and the network topology will certainly change.

The prevalent credo of "do not repeat yourself" (DRY) to avoid code duplication becomes a trade-off between sharing code and keeping the coupling between services low. Therefore a rule of thumb is suggested "don't violate DRY within a microservice, but be relaxed about violating DRY across all services" [New15].

Beside the problem that sharing code increases coupling, it is often not possible to share code when harnessing the opportunity to use different technologies for different services. This might lead to duplication of efforts.

2.3 Feedback-Driven Development

[CLG⁺15] present the concept of Feedback-Driven Development (FDD) that points out the need for holistic and cloud-specific engineering methods, which tackle the issue that developers often build cloud applications without having insights about how those applications behave in production. The goal is to shorten the loop of software development, deployment, operation, and feedback by making real-time usage information available during software development.

Figure 2.5 illustrates the basic principle: In a first step, the cloud application gets deployed to a cloud platform, which is continuously monitored. As soon as monitoring data is available, filtered and aggregated data is handed back to the developers.

¹⁷<http://swagger.io/>

¹⁸<http://readme.io/>

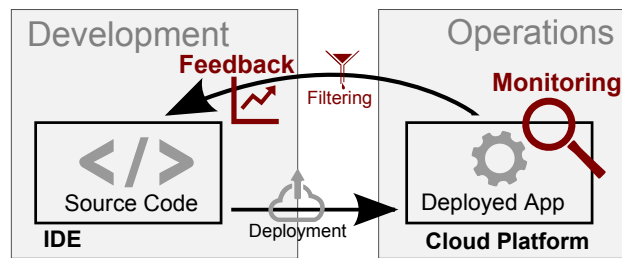


Figure 2.5: Principles of Feedback-Driven Development (taken from [Bos15])

Thus, the manual process of software development is supported by runtime data that is made available in a valuable form. The challenge is to refine the right amount of data that is most relevant during development. In order to take full advantage of such timely data, short development cycles are required but in the same time, incremental and usage-driven development cycles are also enabled thereby [BFKB⁺14]. Given that, developers have the opportunity to make their decisions during development based on runtime metrics, rather than based on intuition as [CLFG14] describes the status quo.

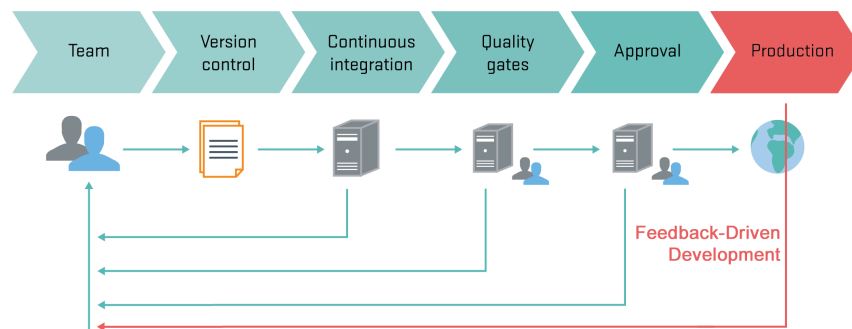


Figure 2.6: FDD in a Continuous Delivery pipeline (taken from [Bos15])

FDD is tightly interrelated to the trends of Continuous Delivery and DevOps, as described in Section 2.1.3. [Bos15] explains the connection between FDD and Continuous Delivery by embedding the concept of FDD into a Continuous Delivery pipeline (see Figure 2.6). [HF10] looks on feedback as the heart of the software delivery process and therefore the feedback cycle should be as short as possible. Hence, Feedback-Driven Development adds value by providing an additional feedback loop that includes operational runtime information [Bos15]. There is also a link between FDD and DevOps: both promote communication, collaboration, and integration between developers and operational teams [BFKB⁺14]. Monitoring at every stage of the pipeline is the most important way to ensure that Continuous Delivery and DevOps is working [Swa14]. Given that, the impact of any change can be validated fast and no hidden surprises should arise [Swa14]. In that sense, an additional feedback loop from the production environment to the development team also increases validation whether Continuous Delivery and DevOps are actually working.

2.4 Implications for This Work

To sum up in terms of the famous saying *"The whole is more than the sum of its parts"*, the goal of Microservice Architectures is to map the complex real world to a multitude of services that are small, loosely coupled, easy to maintain and separately deployable. The complexity is shifting from a large monoliths to the coordination and communication between those services. Cloud resources such as computation and storage are released when needed. Due to pay-per-usage pricing models, the total costs are now more dependent on how much cloud services are being used. The DevOps movement blurs the line between operations and development. Thus, developers are more and more involved into the deployment of applications and therefore have a stronger impact on deployment costs. Assuming a microservice architecture that is continuously deployed in the cloud, the increase of number of calls from one to another service will have an impact on its load factor and will somewhen lead to the need of more computation power. More computation power can mean scaling up horizontally or vertically, but it will increase the costs of this service. Keeping services small, makes sure that new developers can quickly learn the robes of a particular service they have to change. At the same time, other services might be like black-boxes during development unless they have access to it. This will raise the need of software developer for new forms of assistance during the software development process.

Related Work

3.1 Capturing Feedback Input Data

Our predictive FDD system requires operations data of set of target microservices and code changes of a particular microservice as input in order to predict the impact of the given code change on the overall costs that arise from running this set of microservices in the cloud. In their taxonomy, [CLG⁺15] segment operations data into production and monitoring data, where we focus on the latter, which includes execution performance data, load data and costs data (user behaviour data is irrelevant in the context of this work). The collection of this monitoring data is subject of the first subsection and ways to capture code changes are discussed in Section 3.1.2.

3.1.1 Capturing Runtime Monitoring Data

Bosshard [Bos15] and Röthlisberger [RHB⁺12] use an aspect-oriented approach to capture runtime monitoring data. By injecting code into each method of the target application at compile time, they can collect data such as names of the caller/callee, a current timestamp or memory allocation during runtime. Later, [Bos15] calculates aggregated performance metrics based on this monitoring data. The aspect-oriented approach also suits our needs due to its non-intrusiveness, which is even more important when it comes to multiple microservices. Our work differs from [RHB⁺12] and [Bos15] in the fact that we need to draw a connection between monitoring data of all those independently deployed microservices. Beside research, Application Performance Monitoring (APM) tools such as Ruxit¹ and NewRelic² are widely used, often with the possibility to add custom metrics.

[Bos15] reports a large overhead of the monitoring component. The monitoring framework Kieker [VHWH12] tackles this issue of overheads in continuous monitoring by sampling. Ganglia [MCC04] and Monalisa [NLG⁺03] are further research examples in the area of distributed monitoring systems. Therefore, Stadler et al. [SDGW08] provide input on protocols such as tree-based aggregation or gossip that keep the overhead at an acceptable level. While we bear in mind that dis-

¹<https://ruxit.com/>

²<http://newrelic.com/>

tributed monitoring fits microservice architecture, we focus on a feedback handler that collects monitoring data in order to simplify the access of monitoring data from the IDE.

Furthermore, a variety of API gateway vendors offer monitoring and analysis, e.g., apigee³, 3scale⁴, StrongLoop⁵ or MuleSoft Anypoint⁶. By redirecting all interaction between services to a central gateway enables the calculation of metrics such as response times, failure rates, etc. In contrast, we follow a holistic approach that covers the whole microservice architecture without a limitation to public APIs behind a gateway. We aim at monitoring interactions between multiple back-end services.

Moreover, prior research exists that considers service level agreements (SLA) and quality of service (QoS) aspects between a service provider and consumer, e.g., Michlmayr et al. [MRLD09] discuss server- and client-side monitoring approaches and propose a framework that combines them. In contrast, monitoring of services in this work is limited to gathering information about invocations between services.

Research also tackles monitoring on the cloud service provider side, e.g., [ABDDP13] and [MLW11]. While this is crucial for their own operations, many cloud providers also pass this information further to their customers, e.g. with cloud platform monitoring tools such as Amazon CloudWatch⁷. However, compared to our needs the data is too coarse-grained.

3.1.2 Capturing Code Changes

Gu et. al [Gu12] memorize all interactions of developers in the IDE and use this information to create valuable aggregations in the form of plug-ins such as DevTime, which reveals for what the developer has spent time. They capture all edits while a version control system (VCS) is limited to differences between commits. Interaction with the IDE and temporary code changes are irrelevant for this work. Preferably, we need to capture the code changes before they get committed to the VCS.

Bosshard [Bos15] creates an Eclipse Project Builder, the so-called Feedback Builder, which is called during the build process of a project (incremental, auto, full or clean build). He queries metrics and predictions that are calculated in the back-end and maps them to source code with the help of this Feedback Builder. We adopt the approach of Bosshard and create a custom builder but our work differs in the way that it requires the detection of code changes rather than just mapping known hotspots to certain line of code. We aim at capturing changes that include the communication with other microservices in order to predict the associated cost impact.

³<http://apigee.com/about/products/api-management>

⁴<http://www.3scale.net/api-management/report-analyze/>

⁵<https://strongloop.com/>

⁶<https://developer.mulesoft.com/anypoint-platform>

⁷<http://aws.amazon.com/cloudwatch/>

3.2 Predicting Costs of Code Changes

In cloud computing, costs arise from using cloud resources as seen in Section 2.1.2. This work concentrates on predicting costs that result from modifying the source code of a microservice that is deployed to IaaS cloud resources leased from a cloud provider in order to provide them to developers. Research with the same focus is rare but there is a broad diversity of related research. Jennings and Stadler [JS14] conducted a literature survey and state-of-the-art review in the area of resource management in clouds. They state that managing cloud resources is a hard problem because of the heterogeneity and interdependencies of resource types and the variability and unpredictability of the load.

This is certainly a challenge for both, cloud providers to predict future demand in order to operate their business and allocate their resources, but also for IaaS customers to predict the future workload in order to make sure cloud resources are available when needed (they might have to care about instance reservation, provisioning, booting up, etc. in advance). Similar to our work, they also aim at shifting from reactively towards proactively managing cloud resources (increasing developer's awareness of resource needs might be the most extreme form of proactive resource management). Still, our situation is slightly different: given a code change, we estimate the cost impact based on the additional load that results from this code change. This is crucial to bear in mind when looking at related areas, e.g., resource demand profiling or resource utilization estimation, because it eliminates the problem of unpredictable and highly variant load.

Related literature about resource demand profiling addresses both, cloud providers that use the predicted demand as input for their scheduling and pricing, and the IaaS user needs to care about workload management [JS14]. Despite of the different requirements of the predictions mentioned above and a different utilization of the predicted demand, existing research about profiling still provides us with valuable guidance.

To increase the accuracy of predictions, statistical time series analysis of incoming requests are conducted: e.g., Gong et al. [GGW10] use Fast Fourier Transform (FFT) and Markov chains, Chandra et al. [CGS03] a autoregressive model and Jung et al. [JHJ⁺10] an autoregressivemoving-average model. In order to predict upcoming resource demand, Islam et al. [IKLL12] compare Neural Networks and Linear Regression. Compared to all these approaches, we apply a simplified approach based on average values of incoming requests. Within the context of this thesis, the focus and priority lies on increasing awareness by bringing the predicted value to the developer rather than maximizing the accuracy.

Furthermore, with the prototype Predico created by Singh et al. [SSN⁺11] brings up a query language to conduct a what-if analysis to predict the impact of certain workload changes on the performance of data centers. The system is modelled by the help of a directed acyclic graph (DAG), queuing theory to model response times and influence graphs with the application of a change propagation algorithm to execute the query.

Moreover, there is work which uses predictions for executing a given workload in the cloud, e.g., Mian et al. [MMVP13] predict the resource costs of executing a workload on a given configura-

tion C of virtual machines and data storage in a public cloud in regard to penalties resulting from harming Service Level Agreements. While their aim is to determine the most cost-effective configuration, we rather focus on increasing the awareness of Software Developers regarding the future resource costs influenced by their code changes. If a code change is based on a conscious decision and the developer is totally aware of increased resource costs, then he will certainly make sure that the SLA is not violated and no penalty costs arise. Therefore penalty costs will be ignore in this work.

Based on a cloud customer's budget and the assumption of limited cloud resources Tsakalozos et al. [TKS⁺11] determine a suitable number of virtual machines to execute a given workload. But compared to [TKS⁺11], this work has its focus on Software Engineers during software development and does not assume having a fixed budget for executing a given workload. Our environment is coined by a microservice architecture where all services are continuously deployed in the cloud rather than executing batch jobs. Of course, there might be a budget for every service but we do not assume it because especially with this type of architecture other requirements like performance in the sense of desired response times seem natural as well.

To sum up, there is a variety of work that predicts future resource demand, e.g., in order to allocate resources. Therefore, time series of resources consumption (cpu, memory, storage, etc.) and incoming requests are considered. Our work differs in that it makes the strongly simplified assumption that average resource costs based on average consumption of every microservice is given and is proportional to incoming requests.

3.3 Bringing Software Metrics to the Developer

The novelty of the proposed approach consists of the combination of predicting cost-related dynamic metrics and integrating them into the daily workflow of developers. Therefore, related work about how software metrics are visualized and integrated is relevant.

3.3.1 Visualization

Visualizations are utilized to increase the understanding of programs for a long time. Therefore, Reiss [Rei03] suggests five major requirements for a dynamic visualization system: *Minimize overhead*, *Maximize information*, *Emphasize real time*, *Maximize displayed information*, *Provide a compact display*. Those are crucial for our work, since we are limited in space (visualization within the IDE) and time (developers should not be hindered to continue working), plus emphasizing real time fits together with the FDD goal of bringing runtime information from production to developers.

A vast body of research about source code analysis and corresponding visualizations of software system has emerged and, e.g., was brought together by [Bin07]. Static code analysis is accomplished without executing the program, while dynamic code analysis shows how software really

behaves by calculating dynamic metrics during runtime. With the focus on microservice architectures the target codebase of a service is substantially smaller compared to traditional monolithic applications. Therefore, pure static code analysis is only limited useful because a single microservice is comprehensible and developers are not reliant upon visualizations in order to get an understanding. Complexity arises with the interconnection of multiple microservices (see Section 2.2).

Nevertheless our work shares the commonality of software visualization, e.g., metaphors used in static code analysis are also relevant in the context of visualizing dynamic metrics. For example, Wettel and Lanza [WL08] use a city metaphor where classes are buildings and packages are districts to visualize large software systems in 3D.

Lanza and Ducasse [LD03] propose a lightweight software visualization technique called polymetric view, which allows to visualize five different software metrics on one node/rectangle. Along with that, [Rei03] uses box displays where the height, the width, the hue, the saturation and the brightness of the rectangular are proportional to a certain metric. This can serve as a benchmark in order to visualize multiple metrics.

The challenge of understanding dynamic and loosely coupled systems such as microservice architectures, service-oriented systems or other forms of distributed systems has been identified (e.g., by [CZG11], [FWWH13] and [MC⁺01]) but research on visualizing such systems and interactions between services is sparse.

Hence, the Unified Modeling Language (UML) can serve as a basis, which, according to Fowler [Fow04], represents interactions with four diagram types: sequence, communication, interaction overview and timing diagrams. Communication diagrams as represented in figure 3.1 are mostly related to our work because they put stress on the visualization of how participants connect rather than when they communicate (timing diagram) or in what order they communicate (sequence diagram).

Furthermore, research about trace visualization exists, e.g., Cornelissen et al. [CHZ⁺07] visualize execution traces with a combination of circular bundle views and massive sequence views. The circular view that shows interactions is represented in figure 3.2. The program trace visualization approach of Fittkau et al. [FWWH13] combines a 2D visualization for software landscapes and a 3D city metaphor to visualize applications.

In order to ease analysis of relationships between caller and callees, often, trees are used to visualize a sequence of method invocations; they are called Calling Context Trees [MBV⁺10]. As an alternative, Moret et al. [MBV⁺10] introduce Calling Context Ring Charts to visualize such sequences of method invocations, either with or without corresponding dynamic metrics. In contrast to common tree visualizations, callees are represented by ring segments of either equal length or of a length proportional to the aggregated dynamic metric (see Figure 3.3). This is also relevant in the context of our work because the need to visualize caller/callee relationships in microservice architectures is still present but shifts to inter-process calls rather than intra-procedural flows in imperative programming or inter-procedural flows in object-oriented programs. Those ring charts allow a neatly exploration of dynamic metrics in hierarchical structures. One might

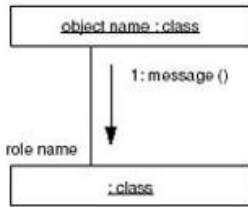


Figure 3.1: UML Communication Diagram (taken from [Fow04])

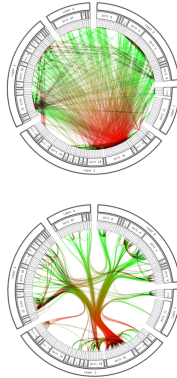


Figure 3.2: Call relations in a circular view (taken from [CHZ⁺07])

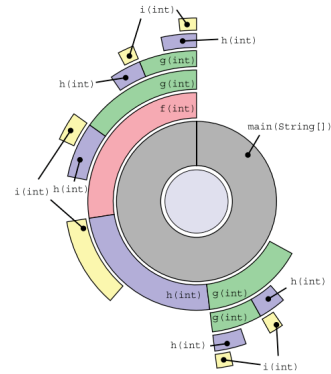


Figure 3.3: Calling Context Ring Charts (taken from [MBV⁺10])

consider to apply those ring charts to visualize dynamic metrics of inter-service method invocations, e.g., the ring length could be used to visualize how often certain service method is called.

Compared to the approaches above, this work aims at increasing the comprehension of microservices related to the one, which the developer is currently working on. In contrast to [MBV⁺10] and [CHZ⁺07], it is concerned with inter-service calls rather than inter-procedure calls. Furthermore, only related microservices are the center of attention and not the whole landscape as approached by [FWWH13].

3.3.2 Integrating Feedback into the Workflow of Developers

A vast amount of data is collected by cloud platforms and application performance management (APM) tools and it is usually available in external tools and thus, is not well integrated in the daily workflow [CLG⁺15]. In their empirical study, Cito et al. (2014) [CLFG14] found that software developers fail to harness monitoring data during development. Furthermore, they found that costs are not an issue in the daily work of software developers. This work aims at tackling those issues.

Prior research on information needs of software developers, e.g., in bug reports [BPSZ10] or in collocated development teams [KDV07]. But research on needs of developers for cost-relevant software metrics is sparse.

Different ways to integrate feedback into the workflow of software developers are discussed here. Brandtner et al. [BGG15] created a mashup framework for continuous integration that acts as single point of access and provides users with quick and accurate feedback about the state of a software system.

Others, as discussed subsequently, integrate feedback directly into integrated development envi-

ronments (IDE) of developers. The work from [Zel07] about the future of programming environments serves as basis and underlines the relevance of assisted decisions in IDEs. In contrast to Zeller's suggestions, our proposed FDD concept goes one step beyond and does not only allow to learn from the past to better understand the future but also from the present. The goal is to bring the latest runtime information from production into the IDE.

In their prototype called "Senseo", Röthlisberger et al. [RHV⁺09] augment static perspectives of IDEs with dynamic metrics. They address the issue that profilers and debuggers collect dynamic runtime information but do not feed aggregated information back into the editor of the IDE. Although it is possible to see the performance of a particular run, it might be more interesting to see the average execution time. The same situation occurs with abstract interfaces, which are implemented by various concrete implementations: a debugger displays the concrete implementation during a particular run but it is often more interesting to see how often which concrete implementation is used. This work is related in the sense of bringing aggregated metrics into IDEs of developers but differs regarding the kind of metric (focus on cost-relevant metrics) and also the type of software system (focus on distributed microservice architectures). In a controlled experiment with 30 professional developers, Röthlisberger et al. [RHB⁺12] show that integrating dynamic information into IDEs significantly decreases the time spent by 17.5% and increases the correctness of the solutions by 33.5%.

Blincoe et al. [BVD15] predict developer's need for coordination based on related tasks they are working on. Their IDE plugin encourages developers to communicate and the underlying guidelines they elaborated share some similarities with this work: They also pursue an automatic and non-intrusive approach, which allows developer to easily gather the designated information without interrupt their work. Furthermore, one of the elaborated guidelines states that the plugin should consider the experience level of the developer when making recommendations. This might be taken up by this work, especially regarding the experience of a developer with two affected microservices.

Lemma and Lanza [LL13] are developing Moon, a live programming language and its IDE. Extreme forms of live programming shift away from the classical edit-compile-run cycle to directly modifying running systems, which leads to direct feedback. FDD corresponds to live programming, as it also aims at fast feedback but the feedback is of predictive nature and the goal is to be able to estimate the impact before changing the running system. One main pillar of the moon IDE is continuous state visualization, which provides live feedback on the state of the overall system in order to allow developers to spot errors early. The focus on microservice architectures requires us to go beyond and not only show the state of the system that is modified but also from other services.

To conclude, while there is broad range of available software visualization approaches, this work differs from prior approaches regarding the following aspects

- many small software systems rather than one particular, large software systems
- dynamic runtime information collected from production environments

The Concept of Bringing Cost-Relevant Metrics to the Developer

This chapter aims at introducing our concept, referred to as CostHat, on an abstract level. It enables the reader to gain an understanding of our approach to bring cost-relevant metrics to the developer.

4.1 Explaining the Rationale

The presented concept aims at bringing cost-relevant metrics to the developer. This includes the goal to enable the developer to make wise design and implementation decisions with regards to the implicated future costs of operation of the whole microservice architecture.

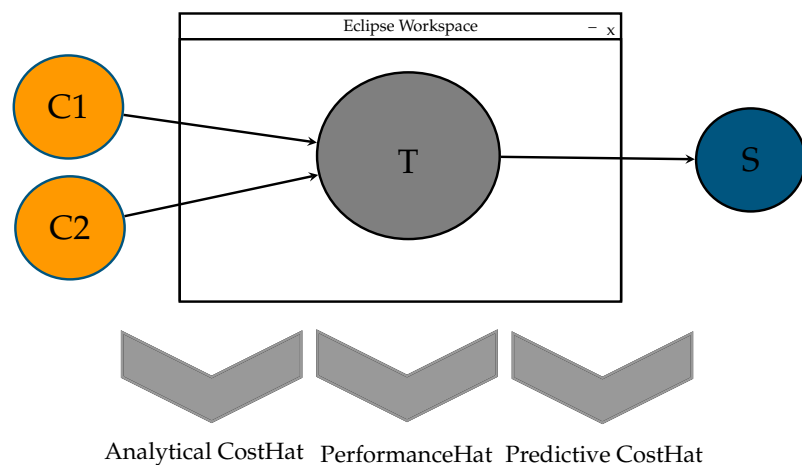


Figure 4.1: Explaining the Rationale of the CostHat Concept

The CostHat is illustrated with the help of Figure 4.1, which represents a simple scenario of a developer working on target microservice T that depends on microservice S and serves two microservices, *caller C1 and C2*. As introduced in the background in Chapter 2, we focus on microservice architectures deployed in the cloud, and so are microservices C1, C2, T and S. Cost impacts from changing microservice T can occur either *directly* or *indirectly*:

- A direct impact on costs of operation means that microservice T has to be horizontally or vertically scaled, either up or down.
- Indirect cost impacts result from calling other microservices, i.e., microservice S, which then have to be scaled.

In order to enable developers to be aware of both, direct and indirect cost impacts, we introduce the two basic building blocks *Analytical CostHat* and *Predictive CostHat*. In the following we look at both building blocks and highlight synergies with existing FDD systems.

Analytical CostHat. As covered by related work (see Section 3.3.1), it is a challenge to understand a loosely coupled system as is given in a microservice architecture. Therefore, we argue that the prerequisite for developers to assess the cost impact of their code changes is to have a detailed overview how microservice T is used in production. Therefore, we introduce an *Analytical CostHat* that provides the developer with dynamic and cost-relevant information about the status of microservice T. More precisely, the *Analytical CostHat* enables the developer to answer the following questions:

- How often is microservice T invoked in total?
- Which services invoke microservice T? How often do they call microservice T?
- How many instances of T are running?
- What is the price for an additional instance?
- How many requests per second can be served by one instance?

Synergies between existing FDD systems. An existing FDD systems, the so-called PerformanceHat, detects performance problems such as hotspots or critical loops based on runtime information [Bos15]. A combination with the presented *Analytical CostHat* enables the developer to link runtime performance issues to runtime cost-relevant metrics, e.g., not only a critical loop of a microservice method is visible but also that only a fraction of request are passed to this method and the low costs of the running instance. When considering the DevOps trend (see Section 2.1.3), which gradually blurs the border between developers and operations, this combination rises new questions such as "Is it worth to make this critical loop more efficient or is horizontal scaling a plausible alternative?". But it can also highlight the relevance to optimize because the costs of operating a service is assessable.

Predictive CostHat. Furthermore, it is often quite hard to assess the cost impact on other microservices, while working on a specific microservice. Therefore, the presented concept includes a predictive part called *Predictive CostHat* that aims at providing the developer with predictions how other services are affected. Other microservices are affected as soon as inter-process communication comes into play. The basic idea is to capture code changes that include inter-process communication to other microservices and to predict the financial impact based on the available information from the *Analytical CostHat*. Considering Figure 4.1, when everything remains constant but inter-process communication is added from service T to service S, then it is possible to calculate the number of additional requests from T to S based on the number of incoming requests to T. When the number of requests per second, which service S can handle to achieve a desired response time, is given, the prediction of the number of additional instances becomes possible. The same approach is applicable when service-to-service communication is reduced.

4.2 Requirements

Requirements for the CostHat are based on the knowledge gathered in the background (Chapter 2) and related work presented in Chapter 3. The characteristics of cloud computing and microservice architectures as well as advantages and disadvantages of microservices play an important role when defining requirements for the presented concept.

[R0] Holistic View It requires that the CostHat provides a holistic view on the microservice architecture. Compared to different API gateway vendors discussed in Section 3.1.1, the feedback of the CostHat includes all services and is not limited to the public API.

[R1] Architecture Independency Concluding from Section 2.2, the focus on microservice architectures does not infer specific communication mechanisms. Therefore, the CostHat should be applicable regardless of whether the communication is implemented via gateway, message-oriented architecture, other forms of middleware or direct service-to-service communication.

[R2] Framework Independency A major advantage of microservices mentioned in Section 2.2 is the independent choice of technology used to implement each microservice. While the focus on a programming language to implement the CostHat is inevitable, at least the framework should be free to choose. This decision is based on the idea that frameworks change more often than programming languages and are often focused on a specific purpose. Each microservice accomplishes a specific purpose and has a single responsibility, consequently different frameworks might be used.

[R3] IDE Integration Derived from Section 2.3, which explains the concept of FDD, the proposed concept should make runtime information available during software development in order

to shorten the feedback loop. The process of developing software process usually takes place in an integrated development environment (IDE), hence, the CostHat should be integrated there.

[R4] Ease of use Cito et al. [CLFG14] find that costs are not a tangible factor in the current workflow of a developer. Based on this finding, the barriers of entry of the CostHat should be as low as possible. The setup has to be easy, just as the operation itself.

[R5] Appropriate Visualization Derived from Section 3.3.1, the five requirements for a dynamic visualization system proposed by Reiss [Rei03]: "Minimize overhead", "Maximize information", "Emphasize real time", "Maximize displayed information" and "Provide a compact display". As our visualization is within the IDE, we are limited in space and, thus, maximization of the displayed information is crucial. We are also limited in time because developers should not be hindered to continue working and therefore, minimization of the overhead is important.

[R6] Provide Context Information A major disadvantage of microservices mentioned in Section 2.2 is the increased complexity due to its distributed and loosely-coupled characteristics. This leads to difficulties keeping an overview. Therefore, it is important to provide the developer with sufficient context information in order to make the feedback valuable.

4.3 CostHat in Detail

The rationale and the derived requirements are building the foundation of the CostHat, which will be described in detail in the following.

4.3.1 Formal Model

The Underlying Environment Given a microservice architecture, the environment consists of a finite set S of n services $s_1, s_2, \dots, s_n \in S$:

$$\text{Environment} \quad S = \{s_1, s_2, \dots, s_n\} \quad (4.1)$$

Assuming the deployment of one or more instances of every service x leads to a finite set I of m instances $i_{x1}, i_{x2}, \dots, i_{xm}$ of each service $s_x \in S$:

$$\text{Instances} \quad I(s_x) = \{i_{x1}, i_{x2}, \dots, i_{xn}\} \quad (4.2)$$

Despite the fact that microservices only have a single responsibility, they can offer multiple methods to other microservices in order to fulfill this responsibility (sometimes also referred as endpoints). Thus, every service $s_x \in S$ has a finite set M of methods

$m_{x1}, m_{x2}, \dots, m_{xn}$.

$$\text{Methods} \quad M(s_x) = \{m_{x1}, m_{x2}, \dots, m_{xn}\} \quad (4.3)$$

Microservices receive requests, apply some logic and finally produce a response [Fow14]. Therefore, we further define a graph as ordered pair $G = (M, R)$, where M is a set of methods as defined in 4.3 and R is a finite edge-set of u requests from a service method a to a service method b at a particular time t defined in 4.4.

$$\text{Requests} \quad R(m_{s_a}, m_{s_b}, t) = \{r_{s_a s_b 1}, r_{s_a s_b 2}, \dots, r_{s_a s_b u}\} \quad (4.4)$$

Simplifying Assumptions It is assumed that all instances of each service $s_x \in S$ are deployed in the cloud based on the IaaS service model as described in Section 2.1.1. Furthermore, an autoscaling unit assures a constant response time of each service $s_x \in S$. Hence, the actual response time of a service is abstracted and cannot be considered by the CostHat. As described in Section 2.1.2, the total costs of an IaaS instance is often the sum of costs arising from all cloud resources such as computation time, storage or data traffic. The price model strongly varies by cloud provider. Thus, the composition of the costs per instance is abstracted and is assumed to be given for each service $s_x \in S$. Altogether, we assume the following constants to be given in order to generate valuable cost feedback:

1. average costs per instance per hour (including all associated cloud resources and services)
2. maximum number of requests per time period per instance to assure constant response time

Feedback Model Individual requests by itself do not generate great value as long as they are not grouped by a time period such as seconds, minutes, hours, and so on. Figure 4.2 shows several requests from a particular service to another service, ordered along a timeline.

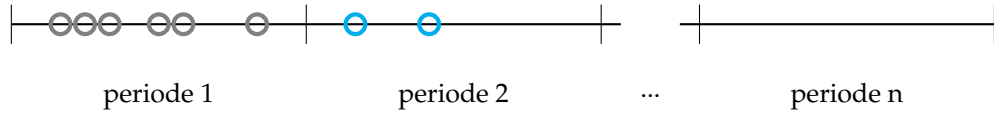


Figure 4.2: Requests ordered along a timeline and divided into periods

This leads to the transformation of the graph G to a weighted directed graph, where each request has a weight (e.g., average requests per second). Therefore, we define a Mapping Function 4.5 from a set of requests R to a set of groups of requests G and an Aggregation Function 4.6 that takes a set of grouped requests and returns an aggregation value $n \in \mathbb{R}_{\leq 0}$, i.e., an average value.

$$\text{GroupedRequests} : R \mapsto \{G\} \quad (4.5)$$

$$\text{AggregationFunction} : \{G\} \mapsto n \quad (4.6)$$

All in all, the aggregation of requests is a weighted graph defined by the triple (M, R, n) .

The proposed CostHat consists of two components:

1. **Analytical CostHat:** Provides the developer with the given constants and with context information from the weighted graph (M, R, n) , filtered by the corresponding services.
2. **Predictive CostHat:** Takes a code change as input in form of an aggregated request delta. Based on the given constants it calculates the simplified linear cost delta and provides this to the developer.

$$CodeChange = (M, R, \Delta n) \quad (4.7)$$

4.3.2 Architectural Overview

This section introduces the proposed architecture consisting of three CostHat system components and several other components in the immediate environment. In order to build the concept introduced in Section 4.1, all architectural decisions are based on the requirements elaborated in Section 4.2 and the formal model in Section 4.3.1.

Figure 4.3 is used to outline the system and environment components on a conceptional level:

- **Target Microservices**

The spawn point is always an FDD user, which builds a software system based on a microservice architecture, called the *Target Microservice Architecture*. Each microservice of the underlying target microservice architecture is a *Target Microservice* and forms the basis for the feedback-driven development. In Figure 4.3 this is visualized as *Target 1* and *Target 2*.

- **CostHat Monitoring Component**

Every single target microservice has an attached *Monitoring Component*. This component collects statistics about the service-to-service communication flow independent of the actual communication mechanism (e.g., direct request, API gateway, message queue, etc.). Furthermore, it collects statistics and measurements of the executed program flow serving as basis to calculate metrics in the future (e.g., execution time of a procedure). A crucial point is the ease of attaching this component to an application. The goal is that the FDD user only has to perform an initial setup so that the monitoring component autonomously starts to monitor. One approach by which this can be achieved, is by using aspect-oriented programming to inject the monitoring component at compile time as proposed by [Bos15]. Based on requirement R1 and R2, the only burden the FDD user has to accept is to mark how service-to-service communication is implemented (discussed in detail in Section 4.3.3).

- **Cloud Platform**

Each target microservice is deployed on a cloud platform. The proposed FDD system is independent from the actual cloud provider. In Figure 4.3, every target microservice is visualized within the integrated development environment (IDE) and deployed in the cloud while the color stays the same.

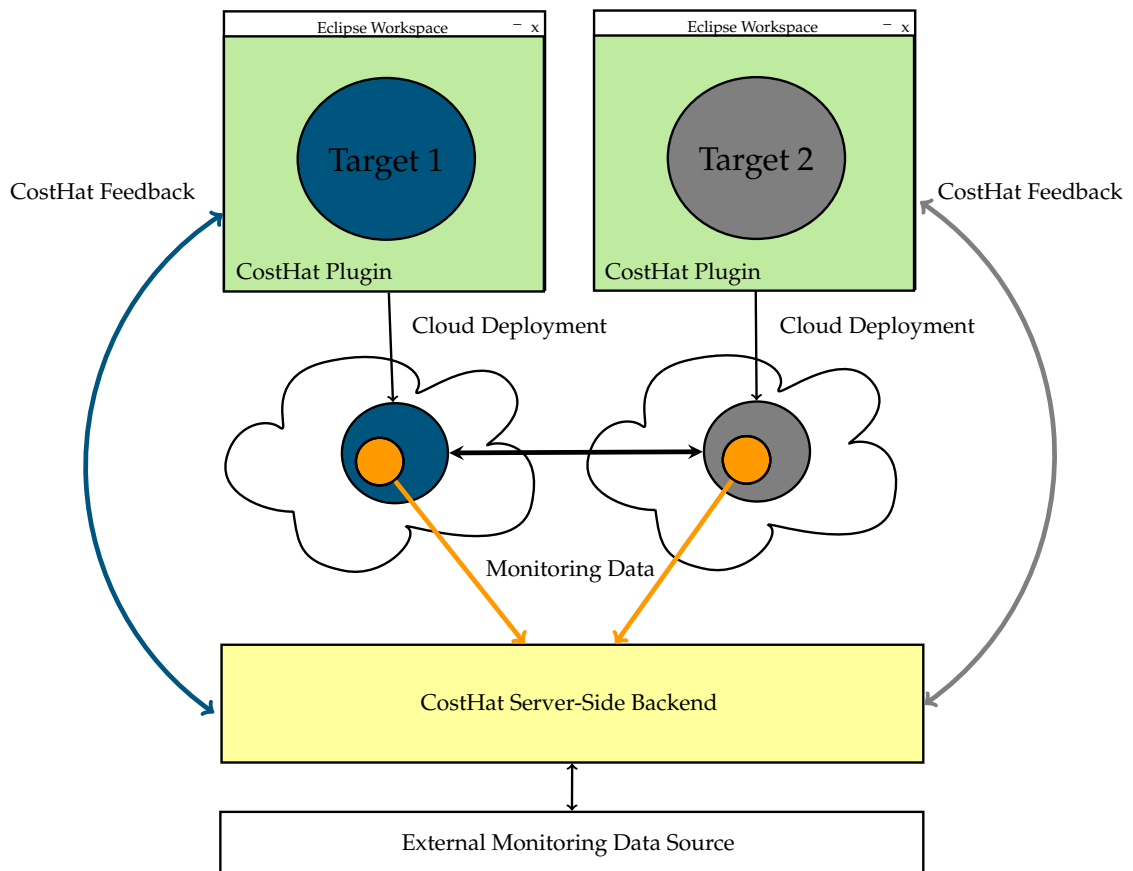


Figure 4.3: Architectural Overview with System and Environment Components

- **CostHat Server-Side Backend**

The monitoring component of every target microservice sends the collected runtime data to a backend. The first step towards value generation is accomplished by bringing monitoring data of all target applications together. Then, a RESTful web service API provides clients with filtered and aggregated monitoring data. More precisely, this means the backend needs to be capable of aggregating all incoming requests to a target microservice and all outgoing requests from a target microservice.

- **CostHat IDE Client**

Software developers working on one or more target microservices use an IDE with integrated CostHat client. Most likely this will be an IDE plugin but on the conceptual level this can be any client. Having mentioned this, we will use the term *CostHat Plugin* for the remainder of this work. The *Analytical CostHat* feature queries the aggregated and filtered data from the feedback-handler and brings them to the developer. The Predictive CostHat feature goes one step beyond and uses the aggregated requests together with code changes

of the developer as input to predict the financial impact. This is performed in the plugin since costs are often highly dependent on the application and on the cloud platform. This architecture allows a future extension of the CostHat Client, e.g., to allow the specification of a pricing model for the corresponding cloud platform. To exchange such application-dependent parameters with multiple developers, a version control system (VCS) could be used. As a consequence, predictions are separated from the common monitoring data with the purpose that the backend could be exchanged.

On the conceptional level, the architectural design decisions of the CostHat differ from other FDD systems such as the PerformanceHat [Bos15] in the following points:

- It creates value by bringing together multiple target microservices rather than one target application.
- The main responsibility of the monitoring component is to gather the interactions of the target microservices rather than the internal program flow.
- The CostHat applies a strict separation between the aggregation/filtering of requests and the prediction of the cost impact. The former is handled by the backend, the latter by the CostHat Client.

4.3.3 Capturing the Context of a Target Microservice

Imagine the use of the proposed FDD system within the context of a target microservice architecture. The CostHat needs to capture the following:

1. The methods of each target microservice that constitute the API and serve as entry points for incoming requests. Both, the target microservice and the methods need an identifier, e.g., a RESTful url.
2. How target microservices communicate with each other, in other words, how external calls are implemented.

This is the foundation for both, the *Monitoring Component* to capture incoming and outgoing requests and for the *CostHat plugin* to provide the developer with cost-relevant metrics. The given requirements imply that the concept is independent from the architecture and independent from the underlying application framework. This has the following implications:

- No particular labels and tags such as annotations can be assumed. In Java, it is common that web frameworks use annotation-based approaches to map incoming requests to Java methods, e.g., Java Spring uses `@RequestMapping`¹ and DropWizard uses `@Path`² annotations.

¹<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html>

²<https://dropwizard.github.io/dropwizard/manual/core.html>

- An identifier of the target microservice and identifiers for the available methods that serve as entry points for incoming requests need to be captured. Although RESTful URLs are commonly used to identify public services, the identification of microservice methods can be different, e.g., some developers may use the Java package names when registering a service at a service registry for internal service-to-service communication.
- Outgoing requests to other microservices need to be captured.

Therefore the CostHat introduces two custom tags listed below. While we suggest to use forms of syntactic metadata specific to each programming language to implement those custom tags, the concept is explained in a Java-oriented notion. Therefore, we take up an annotation-based approach, which is widely used by web frameworks. However, the following two custom annotations are going to be introduced:

1. **@MicroserviceMethodDeclaration to capture microservice methods**

with following attributes:

- microservice identifier
- method identifier

2. **@MicroserviceClientMethodDeclaration to capture microservice client methods**

with following attributes:

- microservice identifier of the caller
- method identifier of the caller
- microservice identifier of the callee
- method identifier of the callee

How it is used by the monitoring component and the backend. The monitoring component tracks the complete program flow associated with timestamps. This also includes all annotations of procedures with all attributes of the annotations. Incoming and outgoing requests within the microservice architecture can then be aggregated by the backend based on the corresponding annotation.

One way to increase the ease of use for the FDD user would be to offer framework-dependent extensions of the monitoring component, which automatically use the annotations of the corresponding framework in order to bypass the use of custom annotations.

How it is used by the CostHat plugin. Each *@MicroserviceMethodDeclaration* annotation provides the plugin with the corresponding microservice and method identifiers. This can be used to query the backend in order to provide the developer with Analytical CostHat information, e.g., the average incoming requests to this method.

The *@MicroserviceClientMethodDeclaration* annotation can be used to detect if code changes contain service-to-service communication. In that case, the plugin can read the identifier of the callee and

the identifier of its method in order to query the backend. Based on the fact that the monitoring component gathered the whole program flow, the backend is aware of how often this kind of service-to-service communication is taking place in this procedure. Hence, the financial impact can be calculated based on the request delta.

4.3.4 Creating Value for the Developer

Looking at the conceptional architecture and the capturing of target microservice context has put the focus on the functional part so far. The next step is to harness this functional part to features that add value for the developer. In the following, three CostHat features are introduced by applying five dimensions adopted from Maletic et al. [MMC⁺02].

Feature 1: Enrich entry point methods of a service with the Analytical CostHat

1. *Who will use the feature?*

Developers who work on a microservice that provides an API for other microservices.

2. *Why is the feature needed?*

IDEs usually provide keyboard shortcuts to show type and call hierarchies or to open the declaration of a given method. In loosely coupled distributed systems other challenges are present, namely there is a lack of assistance to understand the inter-service call trace and to assess the relevance of a method regarding the costs of operation. The PerformanceHat provides assistance to assess the compute intensity of a particular method. This feature provides the frequency of incoming calls to a particular method in the production environment. We argue that this, together with the current operation costs, assists developers to assess the impact of code changes on direct costs to operate this service. A critical hotspot

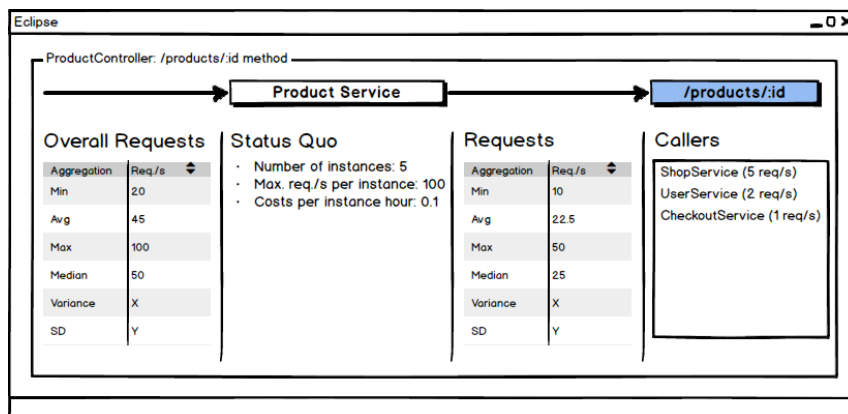


Figure 4.4: Mockup of the Method Declaration

method might get really expensive when called hundreds of times a second but might be a minor issue when called once a day.

3. *What is the data source to represent?*

Analytical CostHat data based on the attributes of the `@MicroserviceMethodDeclaration` and fetched from the feedback handler.

Deriving from the formal model, a microservice has one or more methods. An absolute load measure is provided in form of overall incoming requests to the corresponding microservice aggregated by desired time period, e.g., requests per second. By additionally showing the number of incoming requests to the corresponding method, the developer also gets insights how often the method is called relative to the microservice. Adding the number of instances and costs per instance for this microservice finalizes the cost assessment.

4. *How to represent it?*

The visualization is based on the idea of a sequence flow of the incoming requests from the controller to the respective methods (see Figure 4.4).

We decided to choose this kind of representation because it shares some similarities with the UML communication diagram discussed in Related Work (Section 3.3.1), which is widely used by developers. Therefore we assume faster recognition of the represented data. This should be further encouraged by the use of contrastive colors.

Each arrow and each box build a column of information. Arrows represent requests and the source and destination of the request is represented by boxes. The valuable information is visualized underneath this request flow in the corresponding column.

5. *Where to represent the visualization?*

The exact way to place this visualization depends on the particular IDE but the concept emphasizes to place this visualization as close as possible to the corresponding method. Mapping feedback directly to source code artefacts is one of the main pillars of FDD because otherwise the required effort of the developer increases.

One way to represent this visualization is to use hovers on top of each `@MicroserviceMethodDeclaration` annotation (common in the Eclipse IDE).

Feature 2a: Adding a new invocation of a service to another service

1. *Who will use the feature?*

Developers who work on a microservice and invoke another microservice from the target microservice architecture through a client.

2. *Why is the feature needed?*

Assessing indirect cost implications on other microservices is even harder compared to direct cost impacts. Also, it goes beyond working on a specific microservice. Therefore predic-

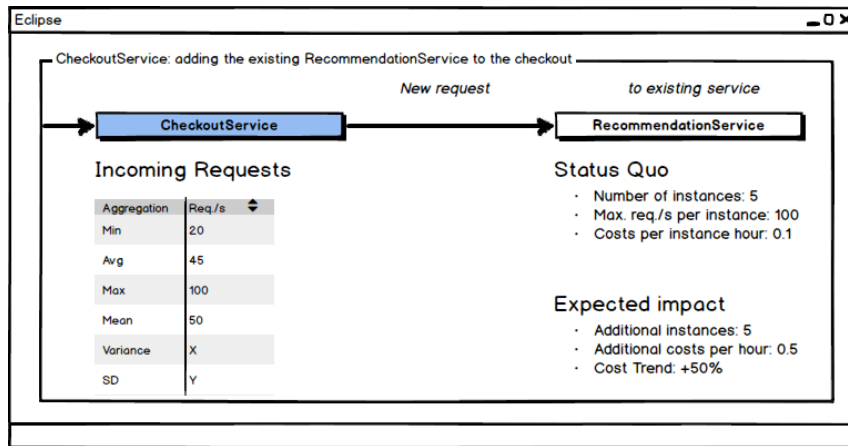


Figure 4.5: Mockup of the Client Invocation Visualization

tions and, in case of significant cost impacts, indications are needed to increase the awareness of the developer.

3. What is the data source to represent?

Analytical and Predictive CostHat data based on the idea that cost implication comes from inter-process communication indicated by `@MicroserviceClientMethodDeclaration` annotations. With the identifiers of the two services captured with the annotation, the CostHat plugin can query the backend.

4. How to represent it?

The visualization is based on the idea of a sequence flow of requests. It differs from Feature 1 in the context, which is concerned with outgoing requests from one microservice to another (see Figure 4.5). Detailed information is once more displayed below the request flow. The major feedback here is the expected impact on the other microservice, which is, consequently, visualized in the column of the other microservice.

5. Where to represent the visualization?

In order to warn the developer of the upcoming cost implication, it is crucial to place this kind of information directly in the source code where the invocation is taking place. Once again, the exact visualization depends on the IDE, e.g., using hover information on top of the corresponding method invocation is a possible approach for the Eclipse IDE.

Feature 2b: Replacing an existing service by a new service

1. Who will use the feature?

Developers who work on a microservice and replace the invocation of one microservice with another microservice from the target microservice architecture.

2. *Why is the feature needed?*

If it comes to software maintenance, often, one microservice is replaced by another one. Developers are reliant on assistance in order to assess the impact on the other service, especially when going beyond by thinking of different implementation scenarios. Beside the scenario of substituting a service by another one, a complementary usage of both services according to certain criteria can be considered, e.g., free and premium users. This feature assists developers to analyse such scenarios.

3. *What is the data source to represent?*

Similar to Feature 2a, Analytical and Predictive CostHat data based on `@MicroserviceClient-MethodDeclaration` annotations. Both services need to be caught, the one used before and the new one. Hence, the load of all affected services are queried from the backend.

4. *How to represent it?*

The visualization of this feature is represented in Figure 4.6 and is an extension of Feature 2a. In addition, it provides a select box to choose another microservice whereof the load pattern is used for the predictions. The default is the microservice that is replaced, but the developer can use any other microservice from the target microservice architecture as a model. Furthermore, radio buttons gather the type of replacement: substitute (fully replaced) or complement (partially replaced). A scrollbar is used to modify the percentage of requests, which will use the new microservice. This enables the analysis of different scenarios and allows the developer to see the impact.

5. *Where to represent the visualization?*

Representation in the same way as Feature 2a.

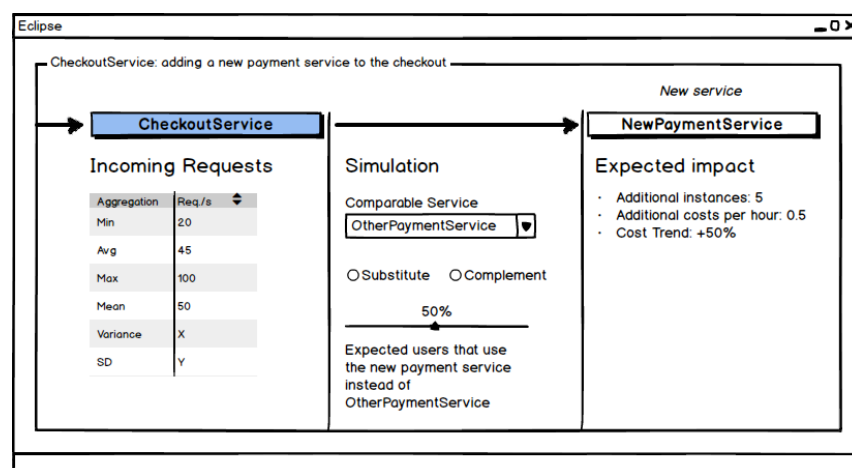


Figure 4.6: Mockup of the Client Invocation What-If Scenario Visualization

Implementation Details

Chapter 4 introduced the concept of bringing cost-relevant metrics to the developer on a high level of abstraction without focusing on concrete technologies. In a next step, a concrete implementation of the proposed concept is presented. The prototype is implemented in Java and contains the system components presented in Section 4.3.2: a monitoring component, a monitoring backend and the CostHat plugin. After specifying the architectural and technical decisions that have been made, each component of the prototype is described in more detail.

5.1 Towards an FDD Framework

The most significant implementation decision is based on the idea that the greatest benefit for developers arises from a complete range of FDD systems. Section 4.1 demonstrated synergies between the PerformanceHat and CostHat and shows one benefit from taking both views, the performance and the cost view. But FDD is not limited to performance and costs, one can also think of other scopes of bringing runtime feedback to the developer, e.g., mapping runtime failures to source code. Therefore, we argue that when designing a new FDD system three key points are crucial to be considered:

1. keep the new FDD system compatible with existing systems
2. use synergies as often as possible
3. enable developers to setup those, and only those, FDD systems required in a certain project.

Implementing an additional FDD system with a new monitoring component would lead to the utilization of two monitoring components in parallel and thus, to an additional overhead. Therefore, using the same monitoring component by extending the PerformanceHat monitoring component seems fruitful to prevent an increase of the overhead.

Extending an existing FDD system also matches Requirement 4 (Ease of use). An initial setup of the monitoring component for all FDD systems limits the effort required to setup multiple FDD systems.

When using the same monitoring component for different FDD systems, the use of the same backend system suggests itself, because otherwise a distinction of which data should be sent to which backend would be needed and the overhead would be increased even more.

As a consequence, we decided to extend the implementation of the PerformanceHat FDD system presented by Bosshard [Bos15]. The goal is not only to implement a CostHat prototype but also to move towards an extendable FDD framework that allows future FDD systems to be added more easily. On the one hand, the basic prerequisites for a such an extensible framework have already been created: The architecture of the PerformanceHat is properly designed to be adaptable and a high level of abstraction keeps the coupling loose. Concrete implementations of interfaces can be easily exchanged because dependency injection is managed with the framework Google Guice¹. On the other hand, the PerformanceHat plugin has been implemented as one single component although many classes that could be reused from other FDD systems. Therefore, a joint base plugin is introduced in Section 5.4. However, details about the implementation of the CostHat prototype are divided into multiple subsequent sections, whereof each section describes one component. In the monitoring and the backend component, the overall description of the component is well separated from the contribution of this thesis.

5.2 Monitoring Component

The monitoring component has to be attached to every single target microservice. This is achieved by extending the aspect-oriented monitoring approach of the PerformanceHat with the ability to track all requests between multiple target microservices, which attach the monitoring component. While the existing approach is shortly outlined in Section 5.2.1, the focus lies on Section 5.2.2, which describes how incoming and outgoing requests between microservices are monitored during runtime.

5.2.1 Aspect-oriented Monitoring Approach

In object-oriented programming (OOP), common concerns are grouped together to classes and structured in hierarchical inheritance trees and thus, concerns are well separated [EFB01]. This is powerful and e.g., increases reusability and maintainability, but it falls short when it comes to monitoring because monitoring code is typically scattered all over the software system. Aspect-oriented programming (AOP) tackles this problem by treating scattered concerns as first-class elements and enabling code injection at multiple points in the target software system [EFB01].

AspectJ² brings aspect-oriented programming to Java and is used by the PerformanceHat to inject the monitoring component at compile-time [Bos15]. It introduces four constructs required to comprehend how this monitoring component is implemented [KHH⁺01]:

¹<https://github.com/google/guice>

²<https://eclipse.org/aspectj/>

- A *join point* is a point in the target software, where the monitoring code is injected and is often considered as node in the runtime object call graph.
- A *pointcut* matches a set of join points and can be combined with other pointcuts by the help of operators (and, or and not).
- An *advice* specifies when certain code should be executed at each join point of a pointcut (before, after or around).
- An *aspect* is a modular unit, which includes the pointcuts and advices.

The PerformanceHat, in brief, declares a *monitoring aspect* with an underlying *pointcut* that contains all method and constructor calls of the target system but without those of the monitoring component itself. The *around* advice is used to execute the monitoring code before and after the actual method or constructor is invoked. [Bos15] uses this to record the whole execution trace and to measure metrics like the execution time for each procedure (method or constructor).

Listing 5.1: Information about the procedure and a current timestamp is collected at every join point

```

1  protected void before(final ProcedureCallJoinPoint joinPoint) {
2      TraceStorage.INSTANCE.add(new
        RunningProcedureExecution(joinPoint.getTargetProcedure(),
        System.currentTimeMillis());
3      setBeforeTimes();
4  }
```

A current timestamp (`System.currentTimeMillis()`) and metadata about the corresponding procedure (`joinPoint.getTargetProcedure()`) are collected at every joint point before the corresponding procedure is invoked (see Line 2 in Listing 5.1). This implies for the CostHat that the timestamp of every invocation of every procedure of all target microservices is already given. It also includes both, procedures that perform calls to other microservices and procedures which serve as entry points for incoming requests from other microservices. To date, the values returned from the `getTargetProcedure()` method are limited to class name, the procedure name, the type of procedure and its arguments. Annotations attached to a procedure are currently ignored. The subsequent section examines how this existing approach can be used to build a monitoring component for the CostHat.

5.2.2 Contributions

In order to track all requests between two target microservices, the two annotations introduced in Section 4.3.3 are harnessed. Procedures used as entry point for incoming requests are annotated with `@MicroserviceMethodDeclaration`. Whenever a procedure with this annotation is invoked, we can associate it with a distinct microservice identifier and a method identifier given in the attributes of the annotation. Therefore, the number of incoming request to a distinct microservice but also to a distinct method of a microservice can be tracked.

Client methods used for microservice-to-microservice communication are annotated with an `@MicroserviceClientMethodDeclaration` annotation. Based on the given attributes, a distinct identification of the caller and callee is possible. Moreover, this approach is independent from the actual implementation of the request (e.g., no matter if it is REST over HTTP, HTTP2/SPDY or a Remote Procedure Call). Therefore Requirement 1 (Architecture Independency) and Requirement 2 (Framework Independency) are fulfilled. Even so, the corresponding annotations have to be set by developers or helper scripts.

Considered Approaches. Two different approaches are considered in order to ensure that the monitoring component keeps track of all procedures with the mentioned annotations:

- Creating a new aspect with a pointcut that matches the CostHat annotations
- Using the same pointcuts as the PerformanceHat but extending the monitoring code to keep track of annotations

The advantage of creating a new aspect is that it well separates the CostHat from the PerformanceHat. While it can be deployed as a single monitoring component, different monitoring code can be injected for both FDD systems. The second advantage results from the finding of Bosshard [Bos15] that the PerformanceHat monitoring component causes a huge overhead. Thus, the overhead could be kept small, when using the CostHat without the PerformanceHat because only the procedures with certain annotations are subject of the monitoring. On the other side, it is only possible to use one *around* advice per procedure because it is invoked in between. The overlap of collected information of both FDD systems is another disadvantage: both collect information about a procedure that is invoked, associated with a timestamp but the CostHat needs to keep track of the annotations as well. It is assumed that the CostHat FDD system will not remain the only FDD system that harnesses annotations of procedures and, therefore, it seems useful to extend the procedure domain model by annotations. A weighting of all these advantages and disadvantages has led to the decision to apply the latter approach of adding annotations to procedures.

Listing 5.2: ProcedureCallJoinPoint

```
1 public Procedure getTargetProcedure() {  
2     return new ProcedureImpl(getTargetProcedureClassName(),  
        getTargetProcedureName(), getTargetKind(), getTargetArguments(),  
        getMyAnnotations());  
3 }
```

Implementation. Listing 5.2 shows a code snippet of the `getTargetProcedure()` method, which is called at every joint point in order to collect information of the corresponding procedure before it is invoked. The domain model has been extended by an *Annotation Interface*, which consists of a `Map<String, Object>` of attributes referred as members and a name of the annotation. The *Proce-*

procedure model now has a list of annotations and the *getTargetProcedure()* method returns a procedure instance with annotations.

Listing 5.3: Annotations of each joint point are mapped to the procedure domain model

```

1  public List<Annotation> getMyAnnotations() {
2      List<java.lang.annotation.Annotation> javaLangAnnotations = getAnnotations();
3
4      List<Annotation> customAnnotations = javaLangAnnotations
5          .stream()
6          .map((annotation) -> {
7              Map<String, Object> members = Maps.newHashMap();
8              // Loop through members of the current annotation
9              for (Method method : a.annotationType().getDeclaredMethods()) {
10                 // ! Insert autocomplete code here
11                 members.put(method.getName(), value);
12             }
13             return new AnnotationImpl(annotation.annotationType().getName(), members);
14         })
15         .collect(Collectors.toList());
16     return customAnnotations;
17 }

```

As revealed on line 6 in Listing 5.3, all ordinary Java annotations³ are mapped to the procedure domain model. In regard to requirement R4 (Ease of use) it is preferable that certain members of annotations are autocompleted in order to relieve the developer from repeatedly adding the same attributes to annotations. The loop on line 9 that iterates through all members can be used to autocomplete, e.g., the CostHat prototype auto completes the identifier of the target microservice, which is set during the setup of the monitoring component.

Review of the Requirements. The setup of the CostHat monitoring component does not differ from the PerformanceHat due to the fact that it is an extension. The proposed approach to implement the CostHat monitoring component is able to fulfill requirement R0 (Holistic View), R1 (Architecture Independency) and R2 (Framework Independency). It does not depend on specific Java frameworks but it depends on a build automation tool that, either itself or with the help of available plugins, is compatible with AspectJ to ensure a proper injection of the monitoring component. When using Apache Maven⁴ together with the aspectj-maven-plugin⁵ the initial setup consists of adding the monitoring component as Maven dependency and pasting AspectJ-related configurations to the build section. However, a step-by-step manual is provided by Bosshard [Bos15]. To date and to the best of our knowledge, plugins for other build automation tools such as Gradle⁶ are either non-existent or not as stable as the the aspectj-maven-plugin. Therefore, up to today (August 2015), building target microservice with Maven is a de-facto requirement. Given a maven project, requirement R4 (Ease of use) is fulfilled.

³<https://docs.oracle.com/javase/8/docs/api/java/lang/annotation/Annotation.html>

⁴<https://maven.apache.org/>

⁵<http://mvnrepository.com/artifact/org.codehaus.mojo/aspectj-maven-plugin>

⁶<https://gradle.org/>

5.3 Server-Side Backend

The server-side backend is referred to as *Feedback Handler*, a term introduced by [Bos15]. It can be seen as nerve centre of the FDD framework. The monitoring data, which is conveyed from all target microservices by their monitoring component (see Section 5.2) or alternatively by external metric sources, is brought together to be consumed in a filtered and aggregated way, e.g., by the CostHat plugin. In the future, filtered and aggregated data could also be dispatched from the feedback handler to other systems, e.g., Bosshard [Bos15] mentions the idea to pass the data further to the monitoring service NewRelic⁷.

The first subsection outlines how the existing feedback handler is harnessed and the subsequent sections introduce the reader to the contributions of this work.

5.3.1 Making Use of the Existing Feedback-Handler

The implementation is based on the Java web framework *Spring*⁸ together with a document-oriented *MongoDB*⁹ database. The corresponding spring-based framework *spring-data-mongodb*¹⁰ is used to map the domain model to the document store. Its fluent API is harnessed to perform query and update operations on the MongoDB. Then, the monitoring data is processed and provided as RESTful HTTP API based on the Spring Web MVC framework¹¹.

While this work builds upon the base architecture of the feedback handler, it also continues with the existing authentication based on an application id and access token in order to facilitate the developer to use multiple FDD systems with a single initial setup. Otherwise a developer would have to register the target application several times for each use case.

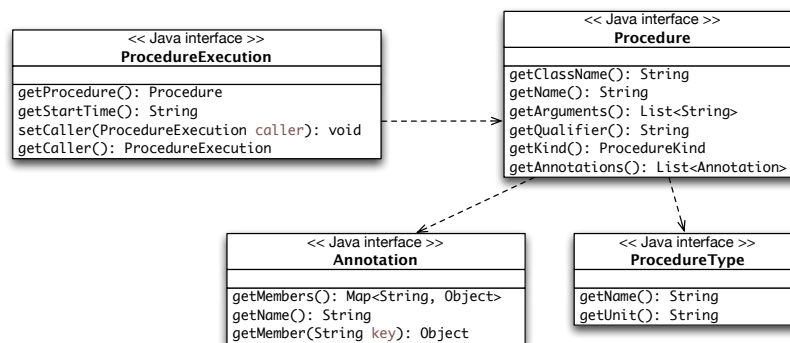


Figure 5.1: Class Diagram of interfaces relevant to filter/aggregate requests between services

⁷<http://newrelic.com/>

⁸<http://projects.spring.io/spring-framework/>

⁹<https://www.mongodb.org/>

¹⁰<http://projects.spring.io/spring-data-mongodb/>

¹¹<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html>

During runtime of a target microservice, the execution of a procedure results in a *ProcedureExecution* entry (see Figure 5.1). It is conveyed by the monitoring component to the feedback handler and stored in the MongoDB.

5.3.2 Contributions

Section 5.2 outlined the extension of the monitoring component by extending the commonly used model by an *Annotation* interface. Figure 5.1 shows how it is linked to the interfaces *Procedure* and *ProcedureExecution*. As a consequence, the database of the feedback handler contains the whole runtime procedure execution trace of all target microservices including their CostHat-related annotations. This results in a huge amount of internal procedure executions entries irrelevant for the CostHat use case. Hence, filtering is required to obtain only those procedure execution entries that are concerned with the microservice-to-microservice communication and aggregation is needed to generate meaningful feedback. The former is examined in the next section and the latter afterwards.

Filtering ProcedureExecution Entries

As outlined in Section 5.2.2, outgoing requests to other microservices are annotated with *@MicroserviceClientMethodDeclaration* and incoming requests to a microservice are marked with *@MicroserviceMethodDeclaration*. Filtering by procedures with corresponding annotation allows to reduce the *ProcedureExecution* entries to either incoming or outgoing requests. But most likely, neither all incoming requests nor all outgoing requests available within the microservice architecture are needed. Therefore three types of filters are considered: filtering by time period, by callee and by caller. Set a start and end timestamp within requests is considered by with filtering by time period. Filtering by callee is used to limit incoming requests to a certain microservice (or to a specific method of it) and outgoing requests to a certain destination, e.g., for the overview how often methods within a microservice are called. Filtering by caller limits outgoing requests to a certain source. But most often, those filters are combined, e.g., to check if a certain microservice has already performed requests to another microservice.

Aggregating Requests

Each procedure execution entry contains a timestamp of when it was started during runtime. In addition to the filtering of *ProcedureExecution* entries, the resulting requests are grouped. Depending on the case, they are grouped by identifier of the caller, callee and/or methods of those. When multiple procedures are grouped, the accumulated timestamps per group are pushed into a list. So far, everything is accomplished directly on the database by the help of the fluent *spring-data-mongodb* API, which creates appropriate queries with *\$match* and *\$group* operations.

Figure 5.2 contains, among others, the *RequestCollector* interface, which represents the result of the described queries. While the concrete implementation may differ depending on how they

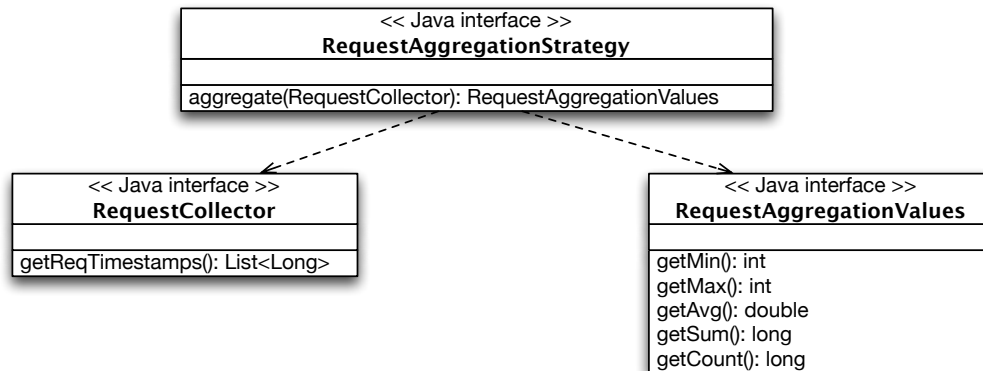


Figure 5.2: Class Diagram of RequestCollector, RequestAggregationStrategy and RequestAggregationValues

were grouped, both concrete implementations, *IncomingRequestCollector* and *ClientRequestCollector* contain a list of accumulated Java timestamps representing the number of milliseconds since the epoch.

While a list of timestamps allows to return the absolute number of requests in the corresponding time period, it is not sufficient to calculate minimum, average and maximum values directly. First, the timestamps need to be grouped again, namely by a certain aggregation interval, which is represented by the enum *AggregationInterval*. An aggregation of requests can occur per second, minute, hour, day or month but it is most common to aggregate requests per second. This kind of aggregation is conducted by a concrete implementation of the *RequestAggregationStrategy* showed in Figure 5.2.

The implemented strategy for the CostHat aggregates timestamps by rounding them to the next lower aggregation period, e.g., second. Based on the time period filter, it calculates the number of expected groups, e.g. given a time period of two days and an aggregation interval of hours, it expects 48 groups. Missing groups are assumed to have zero requests. In a next step, statistics (e.g., minimum, average, maximum) are calculated and returned as *RequestAggregationValues*. The reason to implement a strategy pattern for the aggregation of requests is that it leaves room for further approaches, appropriate to the respective software system, e.g., systems that are only periodically switched on.

Exposing a REST API

The RESTful API is provided by the *CostController*. Adopted from the PerformanceHat, it makes use of the Spring Web MVC¹² annotations such as `@RestController` to mark a class as controller, `@RequestMapping` to map URLs to methods, `@RequestHeader` to get the HTTP headers from the

¹²<http://docs.spring.io/spring/docs/current/spring-framework-reference/html/mvc.html>

request and *@ResponseStatus* to indicate the HTTP status of the response.

The CostHat is affected by the following controllers of the feedback handler:

- **Abstract Controllers:** *AbstractBaseController*, *AbstractBaseRestController*, *AbstractUiController*
- **CostHat:** *CostController*
- **Common FDD Controllers:** *MainController*, *MonitoringController*

The *MainController* serves a newly introduced front-end application to register and manage target applications. It is specially highlighted in the subsequent section. Once a target microservice is registered, its monitoring component makes use of the *MonitoringController*, which serves as entry point to send monitoring data to the feedback handler. The *MonitoringController* also exposes an API to register applications. It was extended in order to allow the front-end to modify and remove existing target applications. Also, the model of applications was extended to cover the number of running instances, the maximum number of requests per second and the price per instance.

The *CostController* is build upon the *AbstractBaseController* and *AbstractBaseRestController* from the base plugin. They provide base features such as the commonly used authentication to the *CostController* and originate from the *PerformanceHat*. The *CostController* provides several endpoints based on the filters and aggregation outlined above. Among others, there is an endpoint that returns all outgoing requests of all registered target microservices and one that returns all outgoing requests filtered by caller or by callee. Also, there is an endpoint that returns all incoming requests to a certain microservice, either grouped by method or not. Furthermore, there is an endpoint that can be used to determine if a particular procedure of a microservice has already called another microservice before, or if this kind of invocation is new.

Listing 5.4: Constants related to HTTP headers

```
1 public class Headers {  
2     ...  
3     public static final String AGGREGATION_INTERVAL = "Aggregation-Interval";  
4     public static final String TIME_RANGE_FROM = "Time-Range-From";  
5     public static final String TIME_RANGE_TO = "Time-Range-To";  
6     ...  
7 }
```

All available endpoints of the *CostController* can be used together with the HTTP headers listed in Listing 5.4. This is the way how the corresponding time period and aggregation interval (e.g., requests per second) is set by the consuming client.

Introducing a Front-End

In order to increase ease of use (requirement R4), a feedback handler web client has been implemented. It allows the registration and administration of FDD target applications, which specifi-

cally means:

- Registering a new target microservice in order to get an access token, which is needed for both, the plugin and the monitoring component.
- Managing existing target microservices in order to set the number of instances, maximal number of requests per instance per second and the price per instance per hour.

The implementation is based on state-of-the-art web technologies such as the JavaScript framework *AngularJS*¹³ and the major angular module *ngResource*¹⁴, which simplifies the interaction with RESTful services. The visualization of the client is based on *Angular Material*¹⁵, an implementation of the widespread *Material Design Guidelines*¹⁶. A screenshot is presented in Section 5.4.5, when it is directly integrated into the CostHat plugin to further ease the setup of new target microservices.

5.4 Eclipse Plugin

The key point of bringing cost-relevant metrics to the developer is the integration into their workflow. This prototype achieves this with a plugin-in for the Eclipse IDE¹⁷. Eclipse plugins are implemented based on the Eclipse Java development tools (JDT)¹⁸, which provide necessary utensils to run the plugin, to access Eclipse resources and to build upon Eclipse user interface components. The scope of this prototype contains Feature 1 and Feature 2a of the features proposed in concept (see Section 4.3.4). To sum up, Feature 1 enriches methods that serve as entry point for incoming request with the Analytical CostHat (incoming request statistics, number of instances of this service, etc.), while Feature 2a is concerned with the addition of invocations of other services and predicting their cost impact.

The PerformanceHat plugin already provides a variety of FDD-specific extensions of the JDT. In order to prevent code duplicates and to increase code reusability, classes useful for other FDD systems were extracted from the PerformanceHat. The resulting *Base Plugin* is introduced in the subsequent section with the proposed idea that the CostHat and future FDD systems add context-independent code to the base plugin. This provides a clean separation of context-independent (e.g., FDD properties) and context-dependent code (e.g., cost predictions) to future FDD systems. It also increases consistency among FDD systems in regard to user interface components.

¹³<https://angularjs.org/>

¹⁴<https://docs.angularjs.org/api/ngResource>

¹⁵<https://material.angularjs.org/latest/>

¹⁶<http://www.google.ch/design/spec/material-design/introduction.html>

¹⁷<http://www.eclipse.org/>

¹⁸<http://www.eclipse.org/jdt/>

5.4.1 Introducing the FDD Base Plugin

The proposed FDD plugin architecture is visualized in Figure 5.3. FDD systems such as the CostHat should only have dependencies to the base plugin and not among each other. This makes sure that with the growth of the FDD plugin environment single plugins can be added and removed without having to resolve dependency issues and without having to reinvent the wheel again and again. In the following, the most important components of the base plugin are outlined.

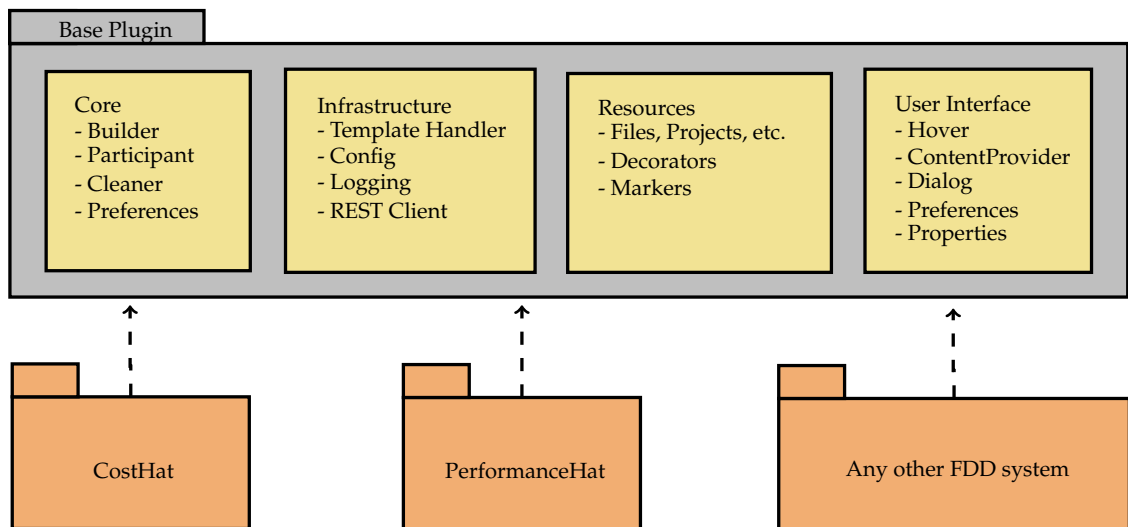


Figure 5.3: FDD Base Plugin as a dependency of the CostHat and the PerformanceHat

Builders. JDT introduces builders, which are capable of being invoked when a build is conducted, e.g., when a changed file is saved [Ble13]. The base plugin contains an abstract *FeedbackBuilder*, which distinguishes the type of build (full or incremental build) and delegates the actual handling to all of the registered *FeedbackBuilderParticipant*'s. Hence, when developing a new FDD system, the handling of different types of builds is abstracted and one can focus on the context-dependent implementations. Therefore every FDD system extends the *FeedbackBuilder* and registers its own participants, e.g., the *CostBuilder* registers a participant for every feature, namely a *MicroserviceClientInvocationParticipant* and a *MicroserviceMethodDeclarationParticipant*.

Natures. The builder is invoked as soon as it is added as a build command to the corresponding project(s) [Ble13]. Manually one would simply add it to the *.project* file but this violates requirement R4 (Ease of use). A running Eclipse is associated with a workspace, which contains zero or more projects, which again may be associated with different *Project Natures* that "represent a type of dimension that a project has" (e.g., Java project nature for Java projects) [Ble13]. This can

be used to tackle the problem of manually adding a builder to a project because builders can be automatically associated with certain natures.

Two approaches can be considered for the FDD base plugin: registering concrete builders of FDD systems to a shared FDD nature or alternatively, creating a new nature for every FDD system. We use the latter approach, because this allows to separately activate and deactivate different FDD systems.

This decision affects how resources are removed before the builder runs once again. The base plugin contains an abstract implementation of a *FeedbackCleaner*, which cleans all markers in case of a full build and only markers of affected resources in case of an incremental build. Therefore it has to be extended by every FDD system to clean its corresponding resources.

Resources. The base plugin contains different layers of resource extensions introduced by [Bos15] as a part of the PerformanceHat. Eclipse resources such as the workspace, projects, files and folders are decorated with FDD-specific functionality, e.g., a *FeedbackProject* with methods to access the application id.

User Interface. The Eclipse compiler uses error and warning markers to indicate problems in source files [Ble13]. FDD builders can also create such problem markers, whereof each has a specific type to be referable later, e.g., by the *FeedbackCleaner*. The base plugin introduces a *FeedbackMarker*, which acts as parent of subtypes like the *CostMarker*. This way it is possible to either refer to markers of any FDD system or to markers of a specific FDD system. Moreover, this *FeedbackMarker* is associated with an annotation, which marks the area in the Eclipse editor.

The need to customize a plugin is assumed to be common among multiple FDD systems, therefore the base plugin contains tools to create preferences (project-independent, workspace-specific) and properties (project-dependent). On the one hand, this contains UI elements, e.g. *PropertyPage-DateField*, which can then be used to create FDD system-specific properties. On the other hand, it contains the fully implemented FDD preferences on base plugin level used to capture authentication details of the target application.

5.4.2 Setting up the Plugin

Eclipse plugins are hierarchically structured into categories, which contain features and each plugin is associated with a feature¹⁹. The CostHat is part of the FDD feature introduced by the PerformanceHat. As soon as the FDD feature is installed in the corresponding Eclipse, a right click on a selected project opens up a menu, which contains the options shown in Figure 5.4. Hence, the developer is able to activate or deactivate the plugin, which basically means that the corresponding project nature is added or removed for the project.

The CostHat introduces three menu entries. While *Enable/Disable Microservice Cost Feedback* is the one mentioned above to toggle the plugin activation, the subsequent two entries are concerned

¹⁹<http://www.vogella.com/tutorials/EclipsePlugin/>

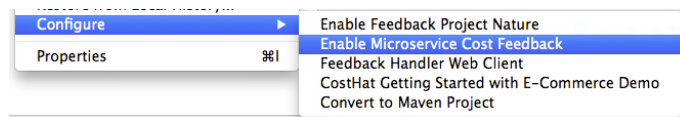


Figure 5.4: Activate/Deactivate CostHat for a certain project

with a feedback handler client and a guide, which is also available in Attachment B.1.

When adding a new target microservice, an application identifier is used to register the application at the feedback handler. In order to increase ease of use for developers, the feedback handler front-end has been brought into the IDE. Figure 5.5 shows a screenshot of the client ready to be used to add a new target microservice, which results in getting an access token from the feedback handler that can be added to the common FDD preferences.

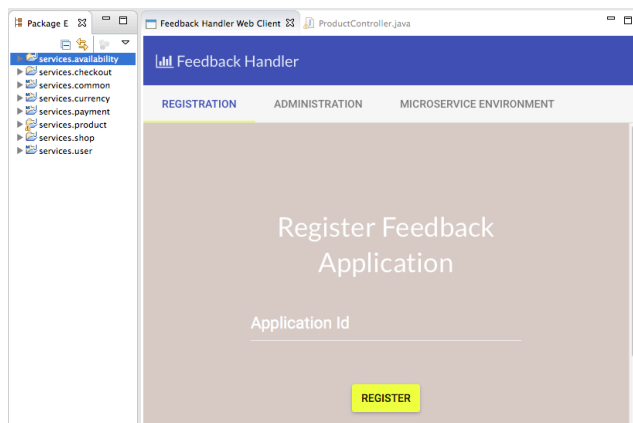


Figure 5.5: Feedback Handler Client within the CostHat Plugin that allows to register new target microservices

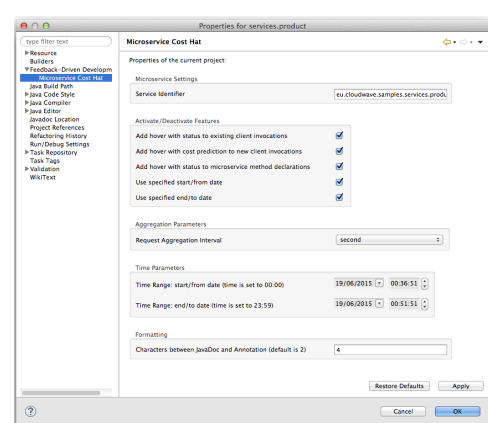


Figure 5.6: CostHat plugin properties for time range, request interval and feature toggling

Figure 5.6 shows the CostHat-related properties. Developers are able to set a unique identifier for their target microservice. This implies the assumption of having a separate project for each microservice. Also, different features of the CostHat can be activated or deactivated, e.g., whether the developer wants to prevent or allow the CostHat plugin to add hovers or a custom time/date range. The date/time can be selected from a calendar and the desired aggregation interval can be selected from a list (e.g., "second" for requests per second). Without having set a start and end date/time, all available requests are considered and thus, the assumed time period starts with the lowest existing timestamp and ends with the latest.

5.4.3 Detecting Code Changes

The *FeedbackBuilder* from the base plugin is extended by the *CostBuilder*, which registers two participants *MicroserviceMethodDeclarationParticipant* and *MicroserviceClientInvocationParticipant*. The former is concerned with the implementation of Feature 1 and the latter with Feature 2a. Each participant contains a *buildFile* method, which is triggered when Eclipse builds a file, e.g. after the developer saved his code changes. The build process itself is abstracted from the actual *Participant* as it extends the base plugin class *AbstractFeedbackBuilderParticipant*, which calls the *buildFile* method on every file that is built. Therefore, participants can be implemented without having to care about different types of builds (e.g., full build, incremental build, etc). The project, the file and the associated computation unit is passed to each participant with the *buildFile* method.

Listing 5.5: CostHat Participants harness the overloaded visit method of an ASTVisitor

```

1  public class MicroserviceMethodDeclarationParticipant extends ... implements ... {
2      protected void buildFile(FeedbackJavaProject project, FeedbackJavaFile javaFile,
        CompilationUnit astRoot){
3
4          astRoot.accept(new ASTVisitor() {
5              @Override
6              public boolean visit(MethodDeclaration node) {...}
7          })
8  }
```

```

1  public class MicroserviceClientInvocationParticipant extends ... implements ... {
2      protected void buildFile(FeedbackJavaProject project, FeedbackJavaFile javaFile,
        CompilationUnit astRoot){
3
4          astRoot.accept(new ASTVisitor() {
5              @Override
6              public boolean visit(MethodInvocation node) {...}
7          })
8  }}
```

Eclipse JDT maps the Java source code to an Abstract Syntax Tree (AST) [KT06]. The computation unit passed to the participant is a node in the AST. Hence, the node can be used to access the source code of the file, which is built. This is the spawn point for participants to map the cost-relevant dynamic information of the respective feature to the static source code.

Each AST node that is received can be queried for child nodes with the help of a visitor [KT06]. The *ASTVisitor* class overloads the visit method with different types of visitors that are passed to any node of the AST. Overriding a suitable visit method allows us to recursively step through the AST. Listing 5.5 shows how the CostHat participants make use of the visit method.

The *MicroserviceMethodDeclarationParticipant* matches all method declarations and checks if a de-

clared method is annotated as entry point for incoming requests. If so, dynamic feedback of Feature 1 can be mapped to the source code.

The *MicroserviceClientInvocationParticipant* matches all method invocations and checks if the invoked method is marked as a client method that performs a call to another microservice. If so, it checks if this service has already been called before from the current target service. A prediction of the cost impact is calculated if this service is newly invoked within the current target service.

5.4.4 Preparing and Predicting Dynamic Content

Beside analysing the static source code with the help of the AST, participants are also responsible to prepare dynamic content that is mapped to the source code. Both participants need to display a lot of information as apparent from the screenshots in Figure 5.9 and Figure 5.8. While the *ContextBuilder* introduced below helps to collect the content in a clear manner, the succeeding sections are concerned with the preparation of actual content.

Context Builder

The *ContextBuilder* simplifies the collection of content within participants. The code snippet in Listing 5.6 demonstrates an utilization of the *ContextBuilder*.

Listing 5.6: *ContextBuilder* that collects dynamic content before it is passed to the template engine

```

1  CostContextBuilder.init()
2    .setTimeParameters(timeRangeFrom, timeRangeTo, aggregationInterval)
3    .setApplication(getApplication())
4    .setRequestStats("incoming", getIncomingRequestsByMethod(serviceMethodId))
5    .setRequestStats("overall", getOverallRequests())
6    .add("serviceIdentifier", serviceIdentifier)
7    .addIfNotNull("serviceMethod", serviceMethodId, serviceMethodName)
8    .build();

```

The *ContextBuilder* provides a fluent API to prepare the dynamic content. Starting with the *init()* method, which creates a new context, key/value pairs can be added before the *build()* method builds the context in order to pass it to the *TemplateHandler*. The *addIfNotNull()* method is another method represented in the code snippet. It adds an object only to context if it is not null. There is also an overloaded version of this method, which accepts an alternative if the first object is null. Beside that, the *ContextBuilder* is extended by the *CostContextBuilder*, which provides CostHat-specific methods such as a special treatment of time periods.

Microservice Status

The feedback handler is queried to obtain the status of a microservices. More precisely, the number of running instances, the maximum number of requests per instance per second and the price

per instance of the corresponding microservice. Each response is mapped to a *ApplicationDto*, represented in the class diagram in Figure 5.7. It is used for both features, e.g., Feature 1 in Figure 5.9 displays this information as *Cost Factors* of the *Product Service*. Feature 2a uses this information as input to predict the impact but also to provide the developer with the status of the microservice, which is called.

Aggregated Incoming and Outgoing Requests

When querying aggregated requests, the time period and aggregation interval set in the properties of the corresponding project is considered. The response is mapped to concrete implementations of the *AggregatedRequestDto* interface, according to the type of aggregated requests (incoming and outgoing). The interface is represented in the class diagram in Figure 5.7. All concrete Data Transfer Objects (DTO) of aggregated requests have in common that they provide methods to get the basic statistics (minimum, average, maximum) and the corresponding aggregation interval. All statements about a certain amount of requests per seconds visualized in Figure 5.9 and Figure 5.8 make use of a concrete implementation of this interface.

Predictions

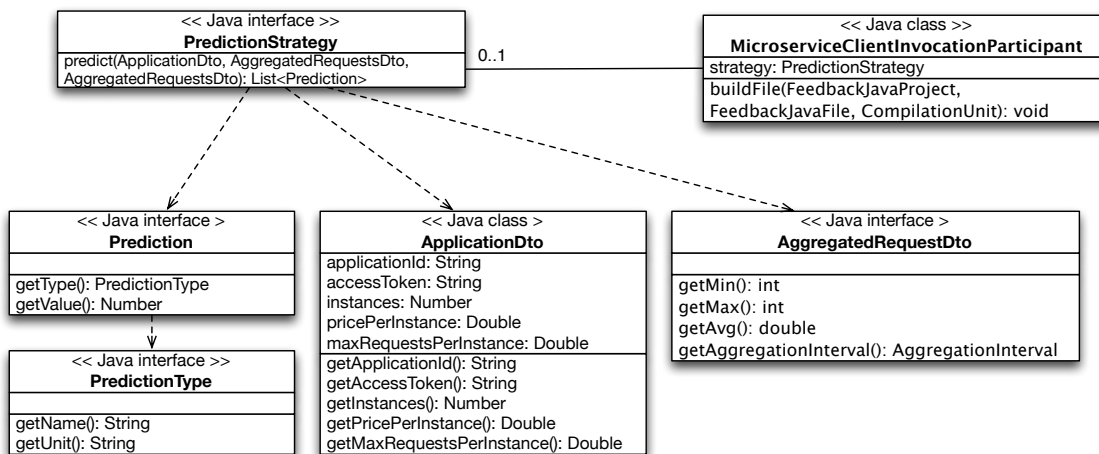


Figure 5.7: Class Diagram of classes and interfaces relevant for cost predictions

The *MicroserviceClientInvocationParticipant* is concerned with the predictive part of the CostHat. As represented in Figure 5.7, this participant has a concrete implementation of a *PredictionStrategy*. When the participant reaches an AST node, which requires a prediction it gathers the status of the invoked microservice. The number of running instances, the price per instance and the number

of requests one instance can handle to keep the response time constant is then passed to the strategy together with *AggregatedRequestsDto*'s of an estimated number of additional requests and the number of existing requests to the invoked microservice. The expected number of additional requests is currently assumed to be the average number of requests to the method from which it is called. Feature 2b could be implemented in the future by simply varying this expected number of additional requests based on other load patterns, e.g., when the developer selects a third service that he expects to have a similar load as the newly invoked service.

However, the domain model has been extended by a *Prediction* and a *PredictionType* interface with corresponding implementations. The *PredictionStrategy* calculates predictions and does always return a list of *Predictions*, also represented in the class diagram. This allows concrete implementations of the strategy to create their own predictions. Within the template, the list of given predictions is iterated and the type of prediction, the predicted value and the unit is displayed. This allows to easily exchange the strategy without having to change the visualization.

Based on the simplifying assumption that all requests are uniformly distributed compute-intensive, the concrete strategy used in the CostHat creates three predictions:

- number of additional instances:

$$requestTotal = avgExistingRequestsPerSecToInvokedService + expAdditionalRequestsPerSec$$

$$instancesTotal = requestTotal / maxRequestsPerInstancePerSec$$

$$newInstances = instancesTotal - existingInstances$$

- additional costs per hour:

$$additionalCosts = newInstances * pricePerInstancePerHour$$

- cost trend

The predictions are demonstrated in the screenshot in Figure 5.8 as *Expected Impact*. Under the expected impact, the developer is also provided with the status of the newly invoked service on which the prediction is based.

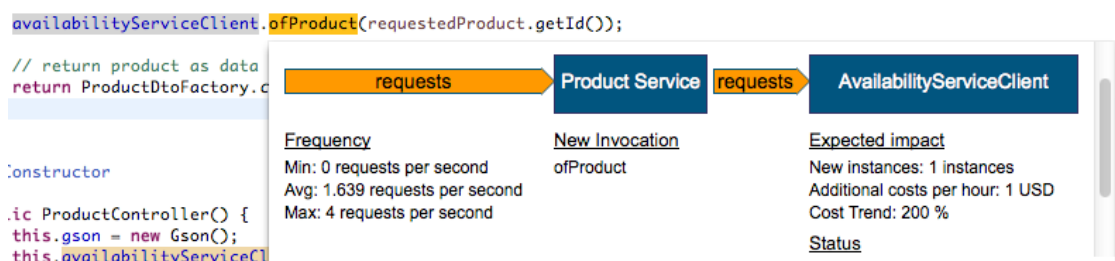


Figure 5.8: Screenshot of the CostHat plugin: prediction of the cost impact

5.4.5 Making the Content Visible

Once the feedback is prepared, it needs to be mapped to the source code. This is still accomplished within the corresponding participant. Similar to the PerformanceHat, the CostHat uses *Markers*, *Marker Annotations* and *Hovers* as common in the Eclipse IDE. The *CostMarker* extends the *FeedbackMarker* from the base plugin. Those markers are used for both features, but with a different severity level. In order to enrich method declarations, a marker with the severity level "info" is used to inform the developer. While the appearance is equal, information markers are labeled accordingly in the Eclipse problem view. To warn developers, when the number of instances needs to be increased, the severity level "warning" is used for those markers. Hence, those markers are also visible as problems in the Eclipse problems view.

Markers appear in the margin and in the problems view of the Eclipse IDE [Ble13]. *Marker Annotations* are used to add a marking to a specific position within the source code. As planned in the concept, Feature 1 displays orange *Marker Annotations* on top of the *@MicroserviceMethodDeclaration* annotation, which is used by the CostHat to indicate a method declaration (see Figure 5.9). Feature 2a puts the *Marker Annotation* at the place where the method is invoked and colours the name of the invoked method (Figure 5.8).

So far, the visualization is limited to *Markers* and orange *Marker Annotations* without providing a lot information content. Therefore, *Hovers* are used, which appear when hovering over the orange *Marker Annotations*. The goal is to implement the mockups presented with the concept in Section 4.3.4. Therefore, the hovers are filled with a Standard Widget Toolkit (SWT) Browser. Then, the template engine FreeMarker²⁰ is used to render the html based on templates. FreeMarker provides conditional blocks that are used, for example, to handle empty variables and iterations are used to list multiple callers of a certain microservice. Furthermore, string, arithmetic and date operations are, among other cases, used to display the time period in a proper way.

This allows to keep the visualization well-separated from the Java code. Every feature has its own template, while they share a common Cascading Style Sheet (CSS)²¹ file. This is accomplished based on the include operation of FreeMarker.

²⁰<http://freemarker.org/>

²¹<http://www.w3.org/Style/CSS/Overview.en.html>

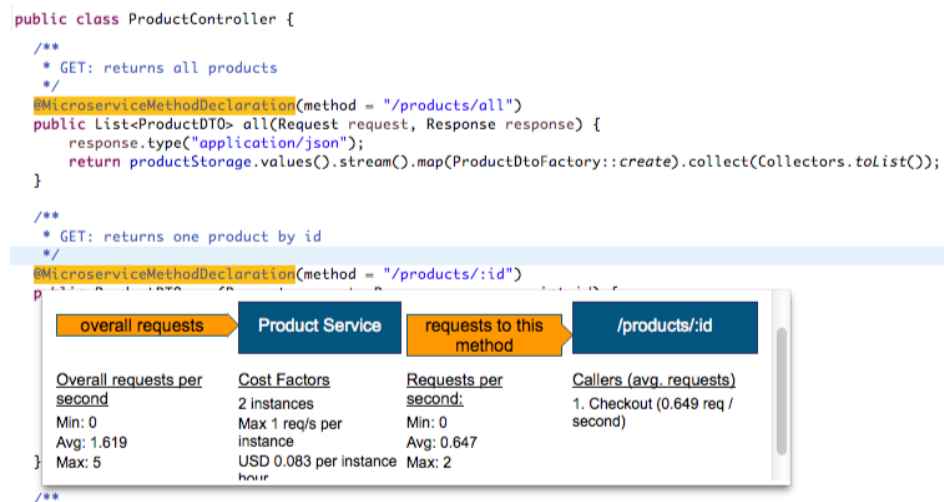


Figure 5.9: Screenshot of the CostHat plugin: method of a ProductController enriched by the Analytical CostHat

Evaluation

The evaluation of the proposed CostHat is divided into three pillars. In the first, a demo application is introduced on which the evaluation of the FDD system is based. An interview study investigates how the CostHat is perceived by software developers and a quantitative evaluation examines the build time overhead that results from the CostHat plugin.

6.1 Demo Application

Due to the fact that microservice architectures are quite novel, available Open Source projects are rare, incomplete because only part of the architecture is publicly available or consist only of one microservice, e.g., people who published their code after following a tutorial to build a microservice with a framework such as Spring Boot¹. This does not suffice the requirements to evaluate the CostHat. Therefore, a demo microservice architecture consisting of seven simple services has been implemented. The CostHat concept described in Chapter 4 implies multiple collaborating services, which are deployed in the cloud. In order to be able to demonstrate some load, each service has been deployed one or multiple times on t2.micro Amazon EC2 instances and the feedback handler backend has been deployed on a t2.large Amazon EC2 instance. While keeping implementational details as simple as possible, the focus was on creating a use case, which provides us with a meshed network of services in order to be able to use this demo application for the interview study. Developers should be able to see the idea and the benefit of the CostHat plugin in a modest way. Therefore we first outline the target use case of the demo application before we give an short overview about implementational details.

Use Case. The demo application covers a classical e-commerce case represented in Figure 6.1. There is a shop service with front-end, which depends on a product and a user service in order to display the shop. When a singed in user buys a product, the checkout service is called, which ensures to have the latest product, currency, user and availability details. If this is given, the checkout service proceeds by calling the payment service.

¹<http://www.infoq.com/articles/boot-microservices>

The underlying use case of the interview study in Section 6.2 simulates the scenario of a developer of the product service, which has the task to add availabilities to the products and only return products that are available. Based on that, the incoming requests to the availability service will be doubled and the CostHat will show a warning.

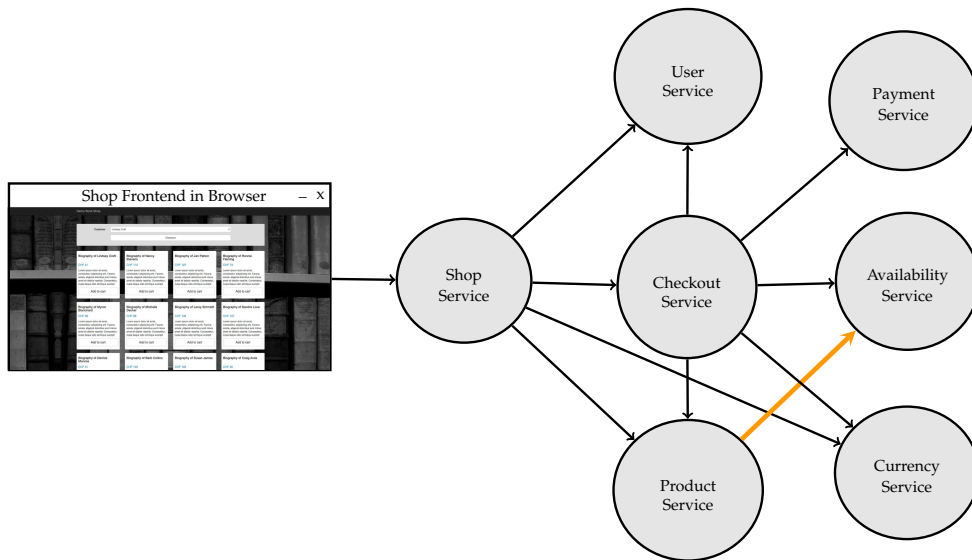


Figure 6.1: Demo Microservice Architecture (arrows represent request flows)

Implementation. All seven services have been implemented in Java based on the Web framework Spark². This particular framework was chosen because of its simplicity and Java 8 support. Due to the fact that the CostHat is not restrictive in regard to specific communication mechanisms, simple direct service-to-service communication based on HTTP requests were used. Neither a service registry/discovery nor any kind of middleware was used for this showcase. For each service, a client class was implemented in order to harness the exposed API of the corresponding service. All clients use simple HTTP requests, which are implemented on the basis of the plain Java *URLConnection* class. As common in state-of-the-art web services, JSON was used as serialization format to exchange objects between services. And in order to deserialize JSON to Java data transfer objects (DTO), the widely-used library FasterXML Jackson³ was used. In order to provide the interviewees with reasonable load information between the given services, a load generator was required. Gatling.io⁴, a Scala open-source load testing framework, was used in order to generate load based on a shopping scenario. Each user navigates twice on the shop before he orders a product. In total, this scenario was executed for 12000 users distributed over 12 hours.

²<http://sparkjava.com/>

³<https://github.com/FasterXML/jackson-databind/>

⁴<http://gatling.io/>

6.2 Interview Study

In this section, the proposed FDD system is evaluated in a interview study with software developers. While the first section covers the study methodology, the subsequent sections report and discuss the results. Afterwards, the interview study is closed with a roundup.

6.2.1 Study Methodology

The interview study was conducted in form of semi-structured interviews. The approach of bringing cost-related metrics to the developer is novel, no similar attempts were found and costs seem not to be a tangible factor for developers. Therefore, the objective of this study is to investigate personal feelings, perceptions and opinions of software engineers when being confronted with the CostHat tooling. Due to the fact that no comparable approaches exist, the goal is to find tendencies that help to assess if this form of bringing cost-related metrics to the developer might be the right way to go.

Protocol. All interviews have been conducted in German and lasted for 26 minutes in average. Two interviews were conducted via Skype and four face-to-face. In the first part, the focus was on gathering context information about the relationship of the participating software engineer to costs of operation and the responsibilities in their companies to keep track of these costs. In the second part, the CostHat tooling was shown to the interviewee. Interviewer and interviewee went together through the use case scenario introduced in Section 6.1. The task accomplished with the use case is consciously a simple one in order to encourage the interviewees to tell right away what catches their eyes and to share their personal feelings, perceptions and opinions by their own accord. Subsequent open questions aim at investigating the impact of the tooling on cost awareness.

Participants. The interviews were conducted with six participants, all of them are male software engineers with more than 5 years of professional experience and an average age of 28. Two participants work for large companies (> 100 employees) based in the United States, four in Switzerland. Three of the four participants work for smaller companies (1 - 100 employees) and one participant is employed in a large Swiss Software and Security Engineering company.

Limitations. There are certain factors that limit the validity of this evaluation. First, the interviews were conducted with only 6 participants, which is a very small sample size. Second, the simple task aimed at encouraging the interviewee to comment on everything they notice and to talk about personal feelings, perceptions and opinions just in the moment of confrontation. While this does not allow us to report results with quantitative measures and statistical significance, it is possible to show tendencies and to assess if this early form of bringing cost-related metrics to the developer is the right path to follow.

Postprocessing. Upon conduction of each interview, the audio recording was transcribed. The raw interview data was then examined using open coding. 12 tendencies were identified and summarized as observations. They were further grouped into 5 categories, which build the subsequent sections *status quo*, *features of the CostHat tooling*, *visual representation of the CostHat tooling*, *integration of the CostHat tooling*, *overhead*. Citations used in this report have been translated to English by the author. The original statements in German are attached in Appendix B.3.

6.2.2 Status Quo

In the beginning, the relationship to costs of operation and relating responsibilities were targeted. Ensuing observations are consolidated as *Status Quo*. It helps to assess if the proposed direction of the CostHat corresponds with the current workflow of the participating developers.

Results

Observation A 1

As far and as soon as operation costs attach great importance, costs also lie within the responsibilities of developers

Participant P4 states that operation costs are less important for his team because the custom client hardware in their business field is much more expensive compared to the costs to operate their service back-ends on which he is working. All other participants indicated that responsibilities of keeping track of operation costs are allocated to lead and senior software developers (P1), DevOps personell (P2), technical project leaders (P5) or even to the interviewed software developer himself (P3 and P6). P5 explains that they usually have a strict separation between business project leaders and technical project leaders but as soon as operation costs in a project are an issue, the technical project leader is affected. In such a case, number of instance limitations and/or throughput requirements are specified and need to be considered during development.

Also, some interviewees indicate that developers are interested in costs as soon as it is part of their responsibilities, e.g., P1 states that "it matters for me because I'm responsible for several deployments".

Observation A 2

Operation costs are treated in a reactive rather than a proactive manner.

The majority of developers treat operation costs like traditional monitoring, in retrospect. Some of the interviewees report that they only act in case of problems:

"[...] costs are mostly considered if there is a problem or something seems strange to us. [...] We

perform changes, release them to alpha and beta channels and then we're going to look at the their impact manually."

– Translated from Interview P1

"[...] when you realize that something is no longer going well."

– Translated from Interview P3

Others in order to make sure that requirements are fulfilled:

"[...] we check if we fulfil the requirements or if further optimizations are necessary."

– Translated from Interview P5

Observation A 3

Existing tools to assess costs of operations are scarce and not geared to developers

While P2 simply states that "this is tough" to assess costs of operations, P3 often relies on his experience. Only one participant was able to mention tools they use, they have even created complicated Microsoft Excel spreadsheets in order to calculate the quality they can offer to clients because tools offered by cloud providers do not provide enough analysis features. He further states that existing billing tools of Amazon and Microsoft are geared to sales people rather than to software developers. Tools that focus on software developers are almost non-existent.

P3 remarks that he ignores possible negative impacts on other services, which he himself is not developing, due to the lack of assistance. This goes along with the statement of P4 that a lot of other teams were not able to correctly assess the impact of their invocations and overused his internal services.

Observation A 4

Some developers express their need to assess operation costs during development in an indirect form such as the complexity of algorithms they can implement to achieve certain performance or the quality they can offer to customers

We found that developers are indeed interested to assess costs of operation in advance. Two of the participants working for smaller companies (P3,P6) indicate their need for a timely assessment of direct costs, e.g. P6 states that it would be more pleasant to anticipate costs in order to detect cost explosions before one ruin oneself financially. P3 even goes further and argues that finding negative cost impacts before the implementation is finished would allow further evaluations of other solutions to a particular problem. In contrast, once the implementation is finished and deployed, incremental optimizations are applied in order to save costs. But maybe if one was already aware of the cost impact during development, one would have made other design decisions or one would have even declined the change request. Even if developers are not responsible

for costs of operations they might be able to anticipate cost impacts and warn their project leaders, which would not be possible otherwise.

On the other side, P1 is mainly interested to know what quality they can offer to their customers referring to the quality of the generated thumbnails, which will influence the required resources such as network traffic, computing power and storage.

"It is not about [direct] costs, it is about performance. Those are indirect costs."

"Higher complexity probably leads to the need of more resources"

– Translated from Interview P2

Discussion

Cito et al. [CLFG14] report that some of their interviewed developers would be interested to have more information about operation costs but it is not part of their responsibilities. While our findings coincide with that, they also show that as soon as costs are an issue, they seem to fall into the responsibility of developers but maybe in an indirect form like a throughput requirement and a number of instance limitation.

Observation A2 indicates that costs are treated in retrospect. One reason of this reactive way of dealing with operation costs could be the lack of available tools (Observation A3). This lack of available tools also explains why [CLFG14] find that operation costs are not a tangible factor for most of their interviewees.

In cloud environments, depending on the application, increasing resources mostly leads to increased performance but also increased costs and as a consequence to the trade-off between costs and performance. One way to look at Observation A4 is, that software engineers necessarily have to make implementation decisions, which will have great influence on costs of operations, especially when developing for the cloud. While every developer has to come to such decisions, only a part of the developers really associate this decisions with costs.

6.2.3 Features of the CostHat FDD Tooling

Observations regarding personal feelings, perceptions and opinions about CostHat features are consolidated as *Features of the CostHat*. It helps to assess the usefulness of the proposed CostHat features and reveals how they are perceived.

Results

Observation B 1

The majority of developers is interested in the load of service methods and in their callers

P3, P5 and P6 state that they really like to see, from which other services a particular service is used. Not only to assess the cost impact but also to understand the system and the network as a whole. P6 points out that to his knowledge no other tool provides such an overview.

P3 expects this feature to be very useful when it comes to debugging and maintenance. He mentions that he often has to integrate other services without tools to assess the responsiveness of those services. He would like to be able to assess how fast those services respond and how they behave depending on the circumstances.

P5 further draws the connection to the writing of testing scenarios. An overview about how the service is used from other services might also be inspiring when it comes to think of possible test cases for this service.

P4 agrees on having the need to know, which services call which of the methods he is working on. But he amends that there are dozens of services, which use his endpoints. It might be impossible to know what they are doing.

Two developers (P3, P6) express the wish to have a complete overview:

"Since I saw the hover, indicating which services call this method, I would prefer a whole overview about my projects, indicating which services call which endpoints and to what extent."

– Translated from Interview P3

Observation B 2

Some developers express their interest in the 'number of instances'-metric, while the majority states that metrics encourage them to optimize

Based on expressions of four out of six interviewees, software developers seem to be driven to optimize by metrics. P2 compares the metrics shown in the plugin with Codacy⁵, a tool for automated code review. P5 perceives it as interesting to be able to directly see if a code change will lead to a massive load, he further states:

"The number of instances that need to be started is a cool metric"

"If I would receive a score, I would certainly be motivated to optimize because no-one likes getting low ratings regardless of whether it is automated or personally."

– Translated from Interview P2

P1 suggests to additionally consider positive impacts of code changes, e.g. if instances can be stopped. He even goes beyond and mentions possibilities to add gamification and the effect of positive reinforcement, which could motivate developers to save costs.

⁵<https://www.codacy.com/>

"Most people optimize for something what they can measure, this means as soon as one has metrics available, one tries to optimize them."

– Translated from Interview P4

P3 perfectly illustrates the problem tackled by the plugin: from a certain system size onwards, everything besides autoscaling is ineffectual because over-provisioning is too expensive and under-provisioning is too slow. In this case there is certainly a need for the developer to be able to assess if new instances will start up due to code changes.

While P1 was the only participant that has already used a tool to assess costs of operations before, he describes that it increased his cost awareness but it had no impact on how he implemented software. After showing the participants the CostHat plugin, two developers (P1, P4) state that the awareness that one performs cost intense external calls has already increased solely due to the fact that one can see an orange hover.

Observation B 3

One developer expresses his need for a feature that tackles the reverse order

P5 again mentions that they often realize that costs of operation are too high after deployment. He raises the need for assistance in order to estimate if his code changes have already reduced the costs sufficiently.

"Often one has the task to make a temporary workaround more efficient in order to reduce costs. In that case one need assistance the other way around: what happens if I remove the following lines of code?"

– Translated from Interview P5

Furthermore, P4 and P5 suggest to extend the plugin with the possibility to perform what-if analysis as captured in the conception of this work (see Chapter 4).

Discussion

Observation B3 could be explained with Observation A2: because the current workflow of assessing costs of operations is reactive, there seems to be a need for assistance when having to improve given issues. It could also be attributed to the difficulties to understand and debug dynamic and loosely coupled systems as identified by [CZG11], [FWWH13] or [MC⁺01].

It seems that both implemented features are well received but it has been indicated that developers do not know all services, which call their services, by name. One could argue that this highlights the limited benefit of a list of callers when the developer has no use for exact names. In turn, the list is sorted by the average number of requests and thus it can be assumed that it helps the developer to assess in which context his service is mostly used. Beside this advantage, there is also a downside of that the plugin has to query the back-end in order to get this information,

which increases build time. A future evaluation where concrete tasks are assigned to developers could be used to further evaluate this.

However, the height of the hover always limits the size of the list of callers unless scrolling is considered. Based on this fact and the input of two developers, the visualization of the complete microservice architecture as a separate feature, which creates an overview, seems fruitful.

6.2.4 Visual Representation of the CostHat FDD Tooling

Observations regarding personal feelings, perceptions and opinions about the visualization of the CostHat plugin are consolidated as *Visual Representation of the CostHat*. It helps to improve the adequate visual representation of essential information.

Results

Observation C 1

Some developers suggest to use different colours to allow a fast estimation of the severity

While one developer (P1) emphasizes that the orange colour of the hover makes it noticeable, two other developers (P5 and P6) suggest to use different colours in order to highlight the severity or magnitude (e.g., how many instances have to be started). It is argued that this enables the developer to get a fast overview without having to move the cursor.

On the other side, P4 puts the stress on how the orange improves the recognition of external service calls. Even without knowing the performance of an external service, it seems interesting to him to see that a call goes outside of the application.

Observation C 2

Half of the developers prefer a more concise visualization that requires fewer interactions

One interviewee (P1) argues that he does not like huge information blocks, he would prefer to see the impact of his code changes in a short and concise form. He suggests to simply use a red +7% label for negative or a green -7% label for positive implications. P2 agrees on that with the additional suggestion to use a simple information bar that shows the percentage of requests going to a particular method, together with the expected duration of the call in milliseconds. He even seems to be confused about the arrows, which represent the incoming flow of requests from the controller to a specific method. He mentions Scala Worksheet, which offers to visualize metrics within the same line of code without having to move the cursor to a hover. He argues that he does not have the patience to move the cursor to a certain hover in order to wait for the content to show up. P3 puts the stress on the cost impact view, which hides, according to him, the most important information that new instances have to be started. Additionally, he confirms the opinion of P2

that having to move to cursor to a hover that hides the information is too invisible.

Observation C 3

The majority of interviewees are not interested to see costs in form of precise and absolute values

P2 clarifies that accuracies to three decimal places are not necessary. P3 and P4 go one step beyond and describe their perception of costs per hours as uninteresting. P2 agrees on that and adds an example that explains why absolute prices are less interesting than relative prices:

"costs per hours... whatever... let's assume you write some code, which is still in use in two years from today; then it's just not meaningful. When I see that an instance hour costs one Swiss frank today, I still don't know how much it is in 6 months."

– Translated from Interview P2

Discussion

The suggestion of a more concise visualization method than hovers might be explained by other upcoming IDEs like IntelliJ⁶, which offer features like Inline Debugging. While the need for concise visualizations and minimal user interactions must be taken seriously, it can also be argued that each IDE has its standards to visualize information and therefore it would be exaggerated to assume that hovers are per se suboptimal. Also, because some developers expressed that they really like hovers and perceive them as adequate. Unfortunately, in the presented interview study we did not ask the participants which IDE they usually use but this might be worth to do in future interviews. But when developing this plugin for other IDEs, it might be worth to reconsider the visualization methods based on what the users are used to in the corresponding IDE.

However, the suggestion to use different colours to mark the availability of a hover (Observation C1) seems fruitful and could encourage fewer user interactions (Observation C2). While our plugin allows to enable and disable certain features (e.g., enable to show predictions that new instances have to be started, disable information about the load of a controller method), the developer always has to move his cursor to the hover. Different colours might even encourage developers to keep more features enabled because they get additional information without having to derogate from writing code.

The observation that developers do not like precise and absolute values might be explained with Observation A4: developers have to implement certain tasks in regard to certain throughput requirements and resource limitations. Therefore it seems more important to see if the impact of their code changes is within the scope of their expectations and limitations. One could even think of integrating a messenger, which allows to directly inform collaborators, e.g., a supervisor, about unexpected cost trends and demand of resources. This could even increase the FDD feedback

⁶<https://www.jetbrains.com/idea/>

loop and would allow integrate non-developers at a early stage. As one participant suggested, it could be reconsidered if it is really worth to implement a certain change request.

6.2.5 Integration of the CostHat FDD Tooling

Observations regarding personal feelings, perceptions and opinions about the integration of the CostHat plugin in the workflow of the developers are consolidated as *Integration of the CostHat*. It helps to assess if the integration of the CostHat into the IDE of developers fits their needs.

Results

Observation D 1

Two out of six developers state that an integration into the Continuous Integration Pipeline would best fit their current workflow

In the beginning, P5 and P6 state that the most appropriate place to integrate notes about significant cost impacts of code changes would be the continuous integration tooling or nightly builds. But P6 also adds that it would be even better if this information could be used within the development process.

Two developers (P1, P5) express that a lean integration without pushed information is important. P1 does not like to have he does not want too much in the IDE. Although P5 likes hover, he mentions that it really depends on how often such hovers occur in the code. When every second line is covered by a hover, it will be difficult to move the cursor without seeing a hover.

Discussion

While an integration of static code analysis into continuous integration tools is absolutely reasonable, it can be argued that the purpose of FDD goes beyond and aims at assisting the developer during development, in a early stage and before the implementation is already finished and ready for delivery.

The desire to keep the IDE clean and without information, which gets pushed, does amplify the relevance to specifically enable and disable features of the plugin. This reveals that offering properties for that is a good approach and can be even extended. Kersten and Murphy [KM06] introduced Mylar, which is now a popular Eclipse Open Source Project called Mylyn⁷, that comes up with task contexts of a developer. Their idea is to filter the amount of information that is presented to a developer by predicting, which information is relevant to a task. This seems beneficial to consider in order to keep the IDE clean unless the developer might need the FDD tooling.

⁷<http://www.eclipse.org/mylyn/>

6.2.6 Overhead of the CostHat FDD Tooling

Observations regarding personal feelings, perceptions and opinions about the overhead are consolidated as *Overhead of the CostHat*. It helps to assess the severity of the overhead that results from both, the CostHat plugin and the monitoring component.

Results

Observation E 1

To multiple developers, the monitoring overhead is more important than the build time overhead

It seems that the build time overhead is perceived as less important as long as the developer can continue working without a blocked user interface:

"I will not wait"

– Translated from Interview P2

P3 mentions that build times in the maintenance phase are less important compared to the initial development where full builds are more frequent. But three out of six interviewees questioned the overhead of the monitoring component:

"I don't care, the overhead of monitoring the productive system is way more important."

– Translated from Interview P6

Some developers expressed that monitoring the production environment adds value, e.g., P5 indicates that one can always draw the most interesting conclusions from production data because there will always be situations where it is difficult to replicate problems that occur.

Discussion

A non-blocking user interface is given because Eclipse makes sure that the build process is executed separately. Moreover, the Java and Maven Builder is executed first and afterwards the CostBuilder starts to run. Therefore, one can argue that developers see the result of their build as fast as before but get additional feedback afterwards. While it seems that reducing the monitoring overhead is more important than the build overhead, it can be argued that those inferences are only limited expressive because developer did not really have to solve a task in the presented interview study. This is certainly true and leaves room for further evaluation. Nevertheless, one has to take into consideration that in microservice architectures the size of a single service is very small and thus, build times are lower compared to large software systems.

6.2.7 Roundup

The interview study finds that although costs are not a tangible factor for developers, costs do lie within their responsibilities when relevant for a project. If relevant, costs are often manifested indirectly, e.g., as quality offered to customers, and they are treated in a reactive rather than a proactive manner. Consequently, the fulfilment of cost-related limitations such as the number of instances or requirements like a desired throughput are only checked from time to time. Tools are rare and not geared to developers and therefore the CostHat seems to fill a gap.

The study finds that both CostHat features meet with a positive response from the participating software developers. While the Analytical CostHat indeed seems to increase the overall understanding of the system, the Predictive CostHat seems to drive developers to optimize. A next step is to consider both, negative and positive cost impacts in order to encourage developers to optimize. Also, an overview of all services together with invocations in-between and the costs arising from each service would reveal the big picture and help developers to assess costs implications. Findings in regard to the visual representation of the CostHat indicate that the visual representation of the plugin should be more concise and allow for faster estimation of the severity without the need for user interactions. A next step into this direction would be to color the marker annotations depending on the severity, e.g., red in case the number instances increases significantly. Also, the information displayed regarding the number of incoming requests could be limited to an average value. It is further suggested to add a percentual value of requests that are passed to a certain method of a microservice.

While the integration of the CostHat in form of an Eclipse IDE plugin seems to be appropriate as long as the IDE remains clean and not overloaded, the Continuous Integration Pipeline was suggested as an alternative. However, we suggest to continue with IDE plugins but a next step would be to consider the current context of developers to limit the feedback to situations where it is needed.

The study indicates that the monitoring overhead might be more relevant than the build time overhead. Based on this, it is suggested to decrease the monitoring overhead in a next step.

6.3 Quantitative Evaluation

The quantitative part of the CostHat evaluation consists of measuring the build time performance of the Eclipse plugin. Thus, the overhead resulting from activating the plugin is quantified.

6.3.1 Methodology

Beside the interview study that investigates how the CostHat plugin is perceived by software developers, this quantitative evaluation measures how the plugin actually performs.

Three builders are involved when building with the CostHat Eclipse IDE plugin:

- Java Builder
- Maven Builder
- CostBuilder (see Section 5.4.3)

Hence, this quantitative evaluation conducts a comparison of the corresponding CostBuilder to the Java and Maven Builder. This comparison is conducted with a Luna Service Release 2 (4.4.2), which is started from the terminal with an activated debug mode as demonstrated in Listing 6.1.

Listing 6.1: Unix shell commands to debug Eclipse builders

```
1 cd Eclipse.app/Contents/MacOS
2 echo "org.eclipse.core.resources/debug=true" > .options
3 echo "org.eclipse.core.resources/build/invoking=true" >> .options
4 ./eclipse -debug
```

The evaluation is based on the demo application introduced in Section 6.1. Thereof, 3 different services were selected (Payment Service, Shop Service and Checkout Service). In a first step, we will execute a full build of all three services to compare the Java and Maven Builder against the CostHat Builder. In a second and third step we will vary the number of both CostHat annotations (*@MicroserviceMethodDeclaration* and *@MicroserviceClientMethodDeclaration* annotations).

All build time measurements are based on 20 executions and the server-side backend is deployed on a t2.large Amazon EC2 instance.

6.3.2 Results

While a complete overview of all measurement values is attached in Appendix B.4, Table 6.1 shows the results of the first part, which measures the general overhead of the CostHat Builder in comparison to the Java and Maven Builder in regard to the three selected demo services (Payment Service, Shop Service and Checkout Service).

	Java and Maven Builder	CostBuilder
Full Build Payment Service with 2 Sources (1 method decl. annotation, 0 invocation annotations)	0.017s	1.980s
Full Build Shop Service with 6 Sources (2 method decl. annotations, 1 invocation annotation)	0.035s	3.896s
Full Build Checkout Service with 9 Sources (3 method decl. annotations, 5 invocation annotations)	0.068s	9.62s

Table 6.1: Average Build Times of Java/Maven Builder versus CostHat Builder

Table 6.2 and Table 6.3 belong to the second part and keeps its focus on one project (Shop Service), while the in the former table the number of client invocation annotations varied and in the latter table the number of method declaration annotations varied.

Shop Service with 6 Sources	CostBuilder	Shop Service with 6 Sources	CostBuilder
0 invocation annot.	3,514s	1 method decl. annot.	3,592s
1 invocation annot.	3,896s	2 method decl. annot.	3,896s
3 distinct invocation annot.	7.470s	3 method decl. annot.	4.331s

Figure 6.2: Avg. Build Times of CostHat Builder with varying number of client invocation annotations

Figure 6.3: Avg. Build Times of CostHat Builder with varying number of method declaration annotations

6.3.3 Discussion

The results show a serious overhead resulting from the CostBuilder. The results are based on a comparison of builds with the CostBuilder and builds limited to the use of the Java and Maven Builder. A possible explanation is the need to fetch data from the server-side backend. While the data needed to display the *Analytical CostHat* on top of *@MicroserviceMethodDeclaration* annotations only needs to be queried once, the data needed to display the hover on top of *@Microservice-ClientMethodDeclaration* annotations is fetched at every annotation. This explains the results from Table 6.3 that shows that the CostHat does not scale with an increased number of service invocation annotations to distinct services. The next step is to query data about all known microservices invoked by the target microservice in the beginning of the build using one single request. In this way, only data about microservices that are added in code changes would need to be fetched during build time.

Trade-Off. There is a trade-off in between *currentness* and *build performance*. Information about microservices such as the number of running instances and the maximum number of requests per instance they can handle in order to keep a certain response time could be cached to reduce the overhead. Also, caching the average number of requests between microservices in the microser-

vice environment would allow to only update this data from time to time and independently from the build process. While this seems fruitful on the one hand, it decreases the currentness.

Mitigating Factors. There are several factors that mitigate the consequences of the overhead. Findings from the interview study indicate that the overhead of the monitoring component is more important than the actual build time overhead as long as developers are able to continue working on their task and already see the results from the actual Java and Maven build. However, Eclipse enables developers to continue working by default due to the fact that the build process is executed in a background thread⁸. While this mitigates the actual build time overhead, developers will still notice the delay of the CostBuilder because the markers and hovers are added later. It is subject to further research to investigate what actual overhead of FDD builders is still perceived as acceptable and hence, finding a reasonable compromise between currentness and performance.

Next Steps. Nevertheless, there is also room for improvement without having to reduce currentness. While expensive network latency is always part of FDD, a next step would be to minimize the overhead by nesting different requests to one single aggregated request from the plugin to the backend. Moreover, the current task context of a developer could be considered. Certain task contexts could be used to predict if a developer needs runtime feedback or not (this is also subject to discussion in Section 6.2.5).

Limitations of this Evaluation. For the sake of completeness, using the CostHat FDD system leads to two types of overhead: First, the monitoring overhead and second, the Eclipse IDE build time overhead. Therefore, this evaluation is limited to one part of the resulting overhead. As outlined in detail in Section 5.2, the implementation of the monitoring component extends the one of [Bos15] who has already evaluated the monitoring component. In his findings, he reports a significant overhead to the overall performance of the monitored application. However, this kind of evaluation is not repeated mainly because of two reasons, (1) it can be assumed that the overhead is at least as significant as it was before, which means it is subject to improvement anyway, and (2) a single target microservice is relatively small, which alleviates the problematic nature of this kind of overhead.

⁸<https://eclipse.org/articles/Article-Builders/builders.html>

Closing Remarks

7.1 Conclusion

We presented a approach that predicts costs of code changes in microservice architectures based on runtime feedback consisting of requests between those microservices. We pointed out that bringing cost-related metrics to the developer is more complicated than just representing a predicted value in the IDE. Hence, Research Question 1 answered by this thesis reads as follows:

How can cost-related metrics be integrated into software development environments in order to enable software developers to assess the financial impact of their code changes?

Understanding the system as a whole and assessing the costs of operation in highly scalable, automatized and distributed systems creates a need for assistance (see Chapter 2 and 3). Therefore the proposed FDD concept (see Chapter 4) introduced an *Analytical CostHat* that maps dynamic information about a microservice to the corresponding service method. This includes metrics such as the number of incoming requests, the number of running instances with the price and the maximum number of requests a service can handle to keep the response time constant. While this illustrates our solution to enable developers to better assess costs of operation, the second *Predictive CostHat* provides the developers with warnings when additional instances need to be allocated.

A prototype was implemented (see Chapter 5) and evaluated based on a semi-structured interview study (see Chapter 6) in order to answer Research Question 2:

How do software developers perceive the availability of cost-relevant metrics within a software development environment?

The *Analytical CostHat* as well as the *Predictive CostHat* were much appreciated by the interviewed software developers. The study indicates that the *Analytical CostHat* may increase the overall understanding of the system and the *Predictive CostHat* seems to drive developers to optimize. Except for the build time and monitoring overhead, it can be concluded that this first approach of bringing cost-related metrics to the developer seems promising.

7.2 Future Work

We report on the first approach that brings cost-related metrics to the developer. Hence, there are many directions for future work.

Visualization Improvements. As the findings of the interview study suggest, future visual representations of the CostHat should be more concise and should reduce the user interactions needed to get feedback. A first call for action includes the coloring of marker annotations based on the corresponding severity level of the warning. A next step regarding the Analytical CostHat is to limit the representation of requests to the average value and in return, add a percentual value of requests that are passed to a certain method of a microservice.

Prediction of Positive Impact. We found that a cost-related metric such as the number of additional instances drive developers to optimize. Therefore, providing the developer not only with negative, but also with positive feedback seems fruitful and could encourage developers to further optimization. Going beyond could even include financial incentives or gamification in order to motivate developers to save costs.

Keeping Track of Requirements. The interview study pointed out that software developers currently treat costs of operation in a reactive manner, while the CostHat provides a tooling that allows a proactive assessment. It seems that developers are often confronted with the situation of having to improve an existing solution in order to fulfill certain limitations or requirements associated with the cloud resource consumption. Future work could elaborate ways to keep track of such requirements and limitations during development in order to prevent developers from violating those requirements.

Weighting of Methods. The presented approach assumes that a microservice offer an API with one or more methods, which are equally compute-intensive. Future work should assess a weighting of methods because in reality there might be a huge gap between simple methods and methods, which cause a great load.

Time Series Analysis. The predictions in the proposed work are based on an average number of requests. Also, the average costs per instance are assumed to be given. Statistical Time Series Analysis of both, requests and costs, could be subject to future work in order to improve accuracy of the cost predictions.

Cost Chain Reactions. So far, cost predictions are limited to the negative cost impact on a microservice B , when increasing the number of incoming requests from a microservice A . In an autoscaling microservice environment, the increased number of incoming requests to microservice B may not only lead to more instances of B but also of microservice C and D and so on and

so forth. A next step would be to keep the simplifying assumptions of this work but to consider such chain reactions.

Visualizing the Big Picture Our findings from the interview study suggest to extend the Analytical CostHat with an overview about the whole microservice architecture. It seems fruitful to visualize all services together with invocations in between, e.g., with the help of a circular view (see Related Work, Figure 3.2). The costs arising from each service could be visualized by the use of different colors. This would provide developers with the big picture of their microservice environment and would allow them to assess where costs arise.

New Combined Metrics. The Analytical CostHat provides the average number of incoming requests to a microservice API method. The PerformanceHat provides an average execution time. Further research may focus on combining these and other metrics in order to spot new combined metrics.

Dynamic Documentation. The majority of interviewed software developers liked to see how frequently a method is called by other microservices. Ordinary static documentation like Javadoc lacks at providing the developer with all information needed during development. It seems fruitful to elaborate ways to integrate a dynamic documentation of microservices. On the one hand, this could include the documentation of the API and the average response time. On the other hand, it could include cost-relevant metrics such as the amount by which changes in requests tend to affect the costs of operations of this service (helps to assess indirect costs) and the price of this service (direct costs when internal pricing is used).

Implementation Hints. In order to better design and implement software, [Zel07] suggests that future IDEs should provide developers with information how similar problems were solved in the past. Also, similar implementations that caused problems in the past should be detected. Applying this to distributed microservice architectures leads to the idea to provide a developer of a service A that calls service B with details of how service C or D successfully implement communication to service B. This could be solved by integrating code snippets from the repository, which seem to offer a better solution (e.g., because they already implemented proper caching).

Cloud-based IDE. A major advantage of both, cloud computing and microservice architectures, is scalability. Hence, cloud-based IDEs like CodeBox¹ provide the same advantages. Furthermore, Wang et al. [WWDR14] refer to the ability to use a flexible number of cloud resources to perform build tasks. One way to tackle the increased build times of the CostHat Plugin would be to rebuild it for a cloud-based IDE. Also, a cloud-based IDE would also facilitate A/B testing.

¹<https://www.codebox.io/>

Considering the Context of Developer. From our findings of the interview study, we extract the suggestion to consider the context of developers in order to predict if developers need runtime feedback to accomplish their current task. Future Work could make use of the popular Eclipse Open Source Project Mylyn [KM06] in order to keep the IDE clean unless the developer needs the FDD tooling.

Overhead Reduction. The presented implementation is a prototype with a significant monitoring and plugin build time overhead. While this fulfilled its purpose to use it for an interview study, the overhead needs to be reduced in order to use it in practice.

Evaluations. The validity of the conducted evaluation is limited by its small sample size. A qualitative evaluation seems fruitful for the relatively new field of bringing cost-related metrics to the developer. We suggest to conduct a task-oriented study with a larger sample size that can be divided into two equally sized groups, whereof one acts as control group. The study could include tasks that lead to code changes with a large negative cost impact.

Appendix A

Acronyms

AOP	Aspect-Oriented Programming
API	Application Programming Interface
APM	Application Performance Management
AST	Abstract Syntax Tree
CD	Continuous Delivery
CI	Continuous Integration
DAG	Directed Acyclic Graph
DTO	Data Transfer Object
FDD	Feedback-Driven Development
IDE	Integrated Development Environment
JDK	Java Development Kit
JDT	Eclipse Java development tools
OOP	Object-Oriented Programming
QoS	Quality of Service
VCS	Version Control System

Attachments

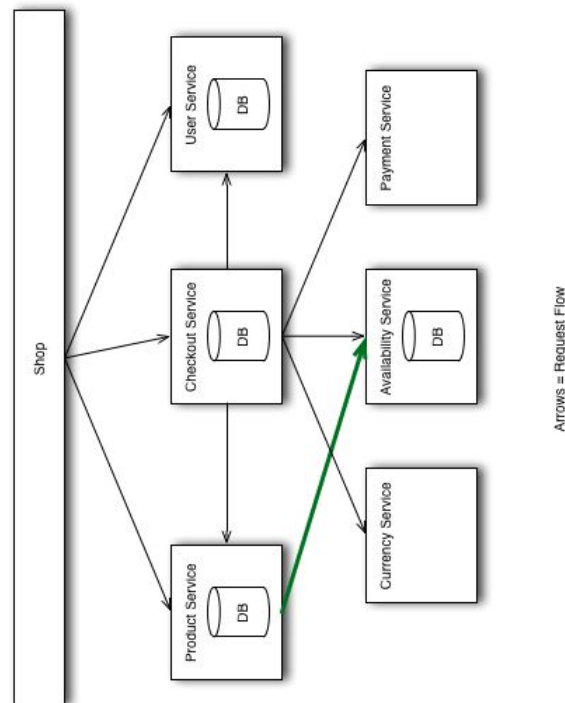
B.1 Demo Application Guide

FDD CostHat

E-commerce Demo Case Study



Introduction

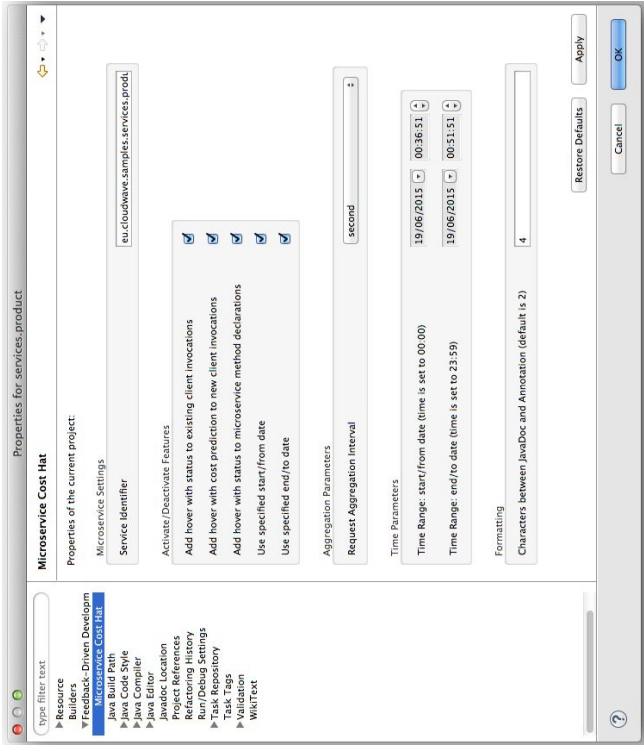
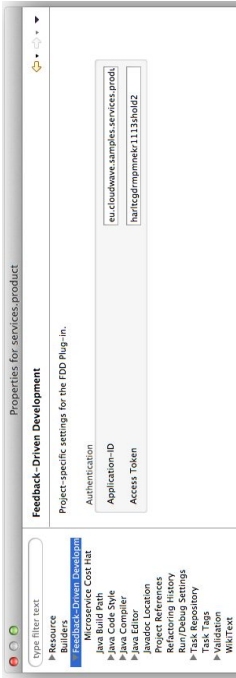
- 8 small demo services in the e-commerce environment
- consciously meshed services in order to demonstrate the plugin
- each service is deployed to a AWS t2.micro instance

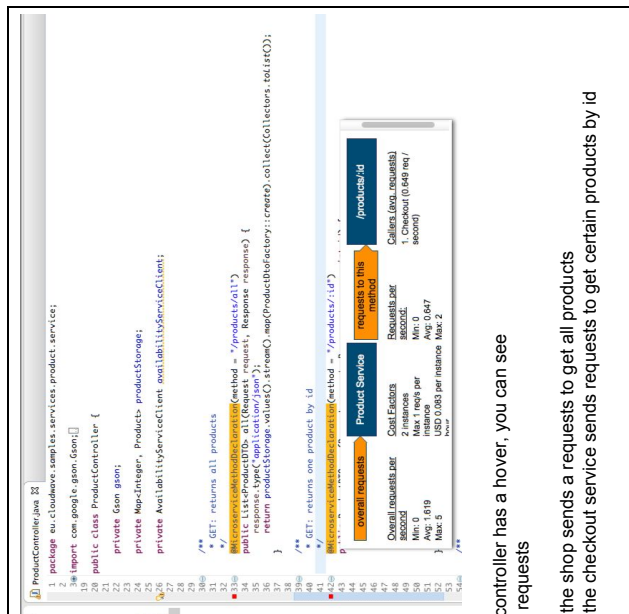
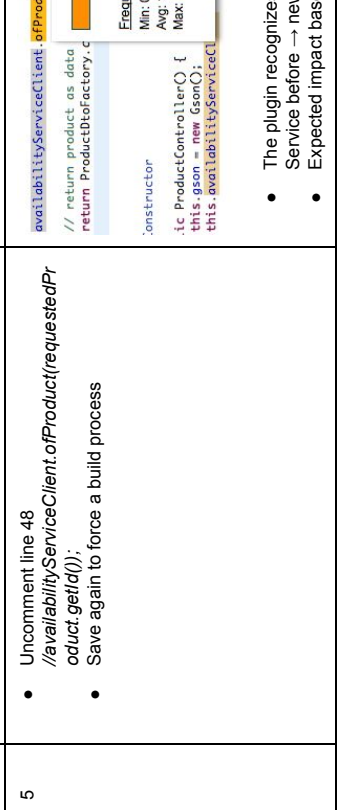


Step-by-Step Guide

Goal: Adding a request from the product service to the availability service → green arrow in the image above

Step	Todo	Desired Result
1	<ul style="list-style-type: none">• Unzip• Open "demoEclipse"	<p>Eclipse with 8 projects is open</p> 
2	<ul style="list-style-type: none">• Right click on service.availability• Configure > Feedback Handler Web Client	<ul style="list-style-type: none">• You can see a web client of our backend• This is the place where you normally would register a new application. But all 8 projects are already registered.• The administration aims at giving you an overview about other services in your microservice environment 

<div>3</div> <div><ul style="list-style-type: none">• Right click on service.product<ul style="list-style-type: none">◦ > Configure<ul style="list-style-type: none">◦ > Enable Microservice Cost Feedback• Right click on service.product (again)<ul style="list-style-type: none">◦ > Properties◦ Compare properties with the screenshots on the right (should be equal).</div>	<div></div>
--	--

4	<ul style="list-style-type: none"> • Open the file ProductController.java (src/main/java/eu.cloudwave.samples.services.product.service) • Remove any empty space and save (ctrl+s) to force a build process 	 <ul style="list-style-type: none"> • Each method of this controller has a hover, you can see <ul style="list-style-type: none"> ◦ frequency of requests ◦ callers <ul style="list-style-type: none"> ▪ e.g. the shop sends a requests to get all products ▪ e.g. the checkout service sends requests to get certain products by id
5	<ul style="list-style-type: none"> • Uncomment line 48 <code>//availabilityServiceClient.ofProduct(requestedProduct.getId());</code> • Save again to force a build process 	 <ul style="list-style-type: none"> • The plugin recognizes that the deployed service in the cloud has never called the Availability Service before → new invocation • Expected impact based on a simple linear model

B.2 Interview Study Questionnaire

CostHat Interview Study - Questionnaire

Emanuel Stöckli / 08-727-836

Part 1

Gathering context information about the relationship of the participating software engineer to costs of operation, the responsibilities in their companies to keep track of these costs and their expectations

Interview Questions

- In what extent is your company interested to estimate the costs of the deployment in the cloud?
- Describe the job of people in your company that are interested in cost-relevant metrics?
Software Engineers? Managers?
- How do you estimate costs? Describe your approach.
- At which point in time do you make those estimations? Before or after deployment?
- How do you keep track of the deployment costs of each project?

Tools

- What tools do you use to estimate deployment costs?
- Describe the limitations of those tools?
- How well are those tools integrated in the software development environment? Describe the workflow.
- Imagine we live in a perfect world:
How would such a tool look like and how would it be integrated in the software development environment?

Think of a project, for example one of your current projects, that you would consider to fit into a cloud and microservice / service-oriented environment.

- Would you use load balancers that automatically scale up and down depending on the number of incoming requests?
 - If yes, how do you prevent excessive costs?
 - If no, what keeps you from using elastic load balancers?
- How would you try to keep track of the communication flow between those services?

Think of a recent project, can you describe a situation where...

- it would have been beneficial to look at some data, but you decided against it because the effort of getting the data seemed to high? What kind of data is it?

Part 2

The CostHat tooling is shown to the interviewee

How Software Developers are affected

- How did cost awareness change after starting to use your tool to estimate costs?
- You're a software developer, what are incentives for you to care about costs?
- How can the availability of cost metrics change the way you implement certain tasks?
 - Does it influence the quality you offer the end customer?
 - or the complexity of the algorithms you use?

Play around with the given CostHat prototype

- What attracts your attention?
- How do you think would it change the way you implement software?
- Do you feel the build time delay?

B.3 Interview Study Results

Interview Study Analysis

Participant	Company Size	Company Country	Age	Gender
P1	small (1 – 100 employees)	CH	26	male
P2	large (> 100 employees)	US	27	male
P3	small (1 – 100 employees)	CH	26	male
P4	large (> 100 employees)	US	26	male
P5	large (> 100 employees)	CH	37	male
P6	small (1 – 100 employees)	CH	28	male

1 Who

Who is affected?	P1: Der Manager lässt uns eigentlich mehrheitlich unsere eigenen Dinge tun. Den Kostenteil übernimmt bei uns der Lead Engineer oder der Senior Developer. Er ist super begeistert von diesen Dingen und hat es geschafft die Kosten um einen Drittel zu reduzieren.
	P1: Wir sind jedoch alle ein wenig "aware" und versuchen dies ein bisschen zu optimieren. Gewisse queries wurden gerade in jüngster Vergangenheit optimiert. Gewisse Daten haben wir z.B. immer mitgeladen, obwohl sie keiner braucht. Nun laden wir sie erst wenn sie auch tatsächlich gebraucht werden. Der Kunde muss zwar dann länger warten aber das sind kleine Optimierungen, die aber bei tausenden von Nutzern eine grosse Auswirkung haben.
	P1: Ich selber finde die Kostenthematik interessant aber ich denke das interessiert viele Entwickler nicht. Interessant wäre zu sehen wie lange dauert ein request vor- und nach meinen Änderungen. Vielleicht kann man das wie TestCases aufbauen.
	P1: Ich könnte mir vorstellen, auch im Team, dass es einige überhaupt nicht interessiert weil sie als Developer am Schluss nicht dafür bezahlen müssen.
	P1: Vielleicht interessiert die Entwickler mehr die Performance an sich, weil sie daran auch vom Kunden gemessen werden. Ich könnte mir vorstellen, dass Lead Entwickler gewisse Kosten einhalten müssen und die wären dann auch mehr daran interessiert.
	P2: Kosten werden schon "considered" aber grundsätzlich haben wir ein spezielles Team dafür, also die DevOps, welche den ganzen Code deployen und monitoren.
	P2: Ich sage nicht, dass mich kosten nicht interessieren, ich sage nur für die Firma ist es nicht relevant. Für mich persönlich, wenn ich optimieren würde auf grosser scale, dann geht es um viel Geld.
	P3: Da ich durchaus für einige Deployments zuständig bin spielt das für mich schon eine Rolle. Ich deploye aber nur selten in einem geclusterten Umfeld, mehr einzelne spezifische Instanzen in verschiedenen Clouds.
	P3: Das sollten hauptsächlich Projektleiter sein und die steuernden Organe im Projekt. Es kommt aber auch darauf an wer das Deployment überwacht, wenn dies der Entwickler ist, dann spielt das durchaus eine Rolle. Spezifische Zahlen werden vermutlich erst eine Hierarchiestufe

	<p>weiter oben wichtig sein aber eine Abschätzung wie viel Aufwand es für den Entwickler wird ist immer wichtig.</p>
	<p>P4: In dem Team wo ich arbeite müssen wir uns nicht gross mit Kosten auseinandersetzen hauptsächlich weil die meisten Kosten die wir haben sind Hardware-Module und Lizenzen. Wir haben viele hardware security modules und die kosten viel mehr als eine Maschine wo unsere services deployed sind.</p>
	<p>P5: Wir haben üblicherweise eine strikte Trennung zwischen Business Projektleiter, welche die Kosten interessieren und technischen welche sich für die Umsetzung interessieren.</p>
	<p>P5: Wenn Kosten ein Thema sind dann sind sie es sehr bewusst und dadurch ein Kriterium und man bekommt Einschränkungen und Requirements wie viele Instanzen von welchen Services es geben soll oder wieviel Durchsatz es braucht. Dann ist es jedoch kein minimieren der Kosten sondern man schaut, ob man die Anforderungen erreicht und muss ggf. optimieren. In so einem Fall interessiert das auch den technischen Projektleiter und diese Rolle habe ich auch schon eingenommen.</p>
	<p>P6: Kosten interessieren mich definitiv, vor allem in einem kleineren Unternehmen wo man nicht viel Geld hat spielt das auch als Entwickler eine Rolle.</p>

2 When

Costs are handled retrospectively	P1: Der Manager findet solange alles in einem gewissen Rahmen ist, ist es ok
	P1: Den Kostenanteil schauen wir mehrheitlich an wenn es Probleme gibt oder wenn wir denken es ist etwas komisch. So einmal pro Woche schauen wir in die Tools, ansonsten gehen wir nicht nach jedem deploy im Monitoring die Auswirkungen anschauen. Ausser wir möchten gezielt Dinge optimieren, dann interessiert uns ob die Requests kleiner werden. Können wir dadurch Zeit und Geld sparen?
	P1: Es ist niemand aktiv am monitoren. Wir reagieren eher reaktiv. Und wenn wir gezielt etwas verbessern wollen dann werden die tools vielleicht 1-2 Tage lang intensiver angeschaut. Wenn alles gut ist dann ist es abgehakt.
	P1: Wir müssen die Änderungen vornehmen, dann releasen wir auf den Alpha- und Beta-Release Channels und erst danach gehen wir wieder von Hand nachschauen was sich jetzt in den Statistiken verändert hat.
	P3: Es ist mehr in der Retrospektive, wenn man merkt, dass etwas nicht mehr gut läuft.
	P3: Es kommt auch immer darauf an wie das Testing abläuft, weil spätestens dann muss man eine Umgebung schaffen, die der Realität möglichst nahe ist. In der Konzeptionierungsphase des Projektes spielt es nur bedingt eine Rolle.
	P5: Wenn Kosten ein Thema sind dann sind sie es sehr bewusst und dadurch ein Kriterium und man bekommt Einschränkungen und Requirements wie viele Instanzen von welchen Services es geben soll oder wieviel Durchsatz es braucht. Dann ist es jedoch kein minimieren der Kosten sondern man schaut, ob man die Anforderungen erreicht und muss ggf. optimieren.
	P5: Bei uns geschieht das ziemlich reaktiv: wenn man plötzlich komplizierte Datenbankabfragen macht und Auswirkungen sieht, dann wird es zu einem Thema. Dann müssen wir die Ursachen suchen und fixen.
	P6: Es wäre natürlich angenehmer, wenn man antizipieren könnte und man Kostenexplosionen bemerkt bevor man ruiniert ist.

3 What

Cost Considerations / Motivation	<p>P1: Danach haben wir versucht herauszufinden welcher Request wie lange dauert und welcher Request wieviel kostet. Welcher Request findet wo statt? Wir haben alle Request daten unserer REST services ins Google Analytics eingespeist. Für jeden request haben wir gewisse geschätzte Preise definiert. Das ist jedoch in erster Linie spielerisches Interesse.</p> <p>Etwas konkretes was wir viel anschauen ist, in welchen Ländern finden welche request am meisten statt. So haben wir gesehen, dass in Südamerika der Dienst sehr intensiv gebraucht wird und z.B. in Indien die App einfach mal runtergeladen wird aber wenig gebraucht wird. So können wir unseren Service auf diese Länder optimieren. Einerseits auf Sprachen, andererseits können wir zum Beispiel für Südamerika eigene Server hochfahren um die Performance zu verbessern.</p>
	P1: Eine Weiterverrechnung an den Kunden wird es bei den gebrandeten Version geben.
	<p>P1: Wir berechnen wie viel Qualität können wir unseren Kunden anbieten, damit es uns nicht kilt? Weil bei Bildern insbesondere der Traffic zwischen Client und Server ziemlich teuer ist und sich die Frage stellt, welche Auflösung wir anbieten und ob wir zuerst einmal Thumbnails generieren. Diese Thumbnails müssen jedoch auch zuerst wieder generiert werden, was auch wieder processing power und storage kostet. Mit unserem Google Analytics tool haben wir aber herausgefunden, dass letztgenanntes jedoch sehr günstig ist.</p>
	P2: Das einzige was wir optimieren sind die Anzahl Personen, welche involviert sind. Andere Ressourcen interessiert gar niemanden. Die Zeit vom Deployment ist das wichtigste.
	P2: Wenn ich halt höhere Komplexitäten habe dann brauche ich vermutlich mehr Ressourcen. In dem Sinne kostet es schon etwas. Wenn ich Dinge im Memory mache und nicht auf der Disk dann brauche ich auch grössere Maschinen.
	P2: Es geht wirklich nicht um Kosten, es geht nur um Geschwindigkeit. Das sind auch Kosten aber indirekt.
	P4: Unser Team maintained viele Services, welche von vielen verschiedenen anderen Teams in der Firma verwendet werden. Wir haben das Problem, dass die unsere Services zu fest in Anspruch nehmen, ohne dass sie auf das achtgeben. Daher haben wir viele rate limits eingeführt, per service limits und Monitoring. Wir können so limitieren wie viele Ressourcen welcher client in Anspruch nehmen kann. Von meiner Seite her wär es natürlich gut wenn die anderen Teams ein bisschen mehr achtgeben würden wie viel Kapazität wir haben.
	P6: Sagen wir mal wir hätten einen Machine Learning Task, welchen wir danach deployen. Dann wäre es vor allem spannend zu sehen wie die Kosten wachsen mit mehr Daten.

Why costs are not an issue	P2: Bei uns gilt eigentlich das credo "Machines sind günstig, Developers sind teuer". Die Zeit welche wir mit Optimierung verbringen ist teurer als die zusätzliche Instanz auf Amazon.
	P4: Ich arbeite meistens an Services, welche von anderen Leuten verwendet werden. Ich habe immer das umgekehrte Problem.

4 How

Status Quo: Alternative Tooling	P1: Neben dem Optimieren von Kosten haben wir auch komplizierte Excel Spreadsheets gemacht, womit wir berechnen wieviel Qualität können wir unseren Kunden anbieten, damit es uns nicht kilt?
	P1: Des Weiteren verwenden wir das Abrechnungstool auf Azure. Im Allgemeinen sind jedoch diese Abrechnungstools, auch auf AWS, nicht wirklich gut. Sie bieten wenig an, um Analysen durchzuführen. Da ist relativ viel Verbesserungspotential vorhanden.
	P1: Bei anderen Dingen, aber das hat weniger mit Development zu tun, bauen wir ein eigenes Abrechnungstool. Wir haben eine neue grosse Microsoft Azure Instanz, welche 50k-60k pro Monat kostet. Auf der laufen sehr viele Kunden und die Abrechnungstool von Amazon und Microsoft können zu wenig. Interessant ist, dass Microsoft interessiert daran ist und scheinbar selber noch nicht genau weiss wie man so ein Tool universell gestalten soll. Da gibt es sogar Unterstützung von ihnen, weil sie gerne lernen
	P2: Das ist schwer
	P2: hast du schon einmal einen Java Profiler angeschaut? z.B. Visual VM? Dort gibt es eine Annotation für self-time. Man sieht seine self-time und die Werte, wo er selbst nichts dafür kann.
	P3: Normalerweise basierend auf meiner Erfahrung.

Status Quo: Limitations	P1: von unseren AWS und Azure Leuten in der Firma habe ich gehört, dass diese Tools zu wenig bieten. Alle waren recht überrascht, dass so wenig Analysemöglichkeiten angeboten wird. In gewissen Bereichen sind diese zwar ganz gut, aber die Zielgruppe dafür sind mehr die Sales Personen. Auf Seiten der Entwickler gibt es sehr wenig.
	P1: Überhaupt keine Integration in die IDE
	P3: Auswirkung auf andere services an denen man nicht selber mit-entwickelt hat "werden momentan ignoriert".
	P3: Öfters gibt es die Situation wo man andere services anbinden muss und die responsiveness unklar ist. Ob ein service schnell oder langsam ist und in welchen Umständen er sich wie verhältet ist eigentlich nie dokumentiert.

Status Quo: Needs	P1: Ich möchte nicht zu viel in der IDE haben, vielleicht höchstens ein kleiner feiner Hinweis, falls etwas ist.
	P1: D.h. bzgl. Kosten würde ich eine Information erhalten und wenn ich diese anklicke sehe ich, was mich dieser Request kosten wird. Ich weiss nicht ob eine absolute Zahl Sinn machen würde.

	<p>P1: Vielleicht reicht auch einfach zu sehen wie lange dieser Request dauert und was die Auswirkungen für den Endkunden sind. Vielleicht auch welche Services dies braucht. Wenn der Service, welcher neu aufgerufen wird nur etwas einfaches aus dem Storage liest dann wird es kein Problem sein aber wenn er noch einige JOIN Operationen durchführt wird es schon komplexer.</p>
	<p>P1: Ich kann mir vorstellen, dass dies alles sehr individuell auf den Entwickler zugeschnitten sein muss. Ich bin zum Beispiel sehr daran interessiert zu schauen, dass meine API für mobile Geräte genug Performance bietet. Bei Desktopapplikationen wird das wieder etwas anderes sein. Wenn man unterwegs ist muss die Response einfach optimiert sein.</p>
	<p>P2: Mir hätte ein Balken gereicht, welcher sagt 30% aller requests gehen zu dieser Methode</p>
	<p>P2: ich brauche auch keine Zahlen auf 3 Kommastellen genau</p>
	<p>P2: Es ist cool, aber es würde reichen wenn ich mir das gleich anschauen kann und auf einen Blick sehen würde dass x Prozent der Requests zu einer Methode gehen die y Millisekunden der call dauert.</p>
	<p>P3: Ab einer gewissen Grösse bringt alles andere nicht mehr viel, weil wenn man overprovisioned ist, wird es zu teuer und wenn man underprovisioned dann wird es zu langsam. Bei Applikationen mit load balancer und automatischer Skalierung kann es den Entwickler durchaus interessieren ob durch seine Änderungen Instanzen hochgefahren werden müssen.</p>
	<p>P4: Wenn ich an einem Service arbeite, weiss ich nicht wirklich wer das verwendet und für was. Es wäre schon gut zu wissen, aber es sind dutzende von client services, ich kann nicht bei allen wissen was die machen.</p>
	<p>P5: Bzgl. der Integration würde ich bevorzugen, dass nichts gepushed wird.</p>

Plugin Feedback Visualization	<p>P1: Ich möchte keinen riesigen Informationsblock haben, sondern ich möchte kurz und prägnant lesen was die Auswirkungen meiner changes ist. Das einfachste wäre, wenn man ein +7% in rot oder -5% in grün sehen würde. Die Frage ist immer mit was man es vergleicht und normalisiert.</p>
	<p>P1: Sieht man es nur wenn man auf den orangen hover klickt?</p>
	<p>P1: Ich wäre sicher aware wenn etwas ist, weil es orange wird. Dies ist auch wichtig, dass es so ein klares Zeichen hat. Wenn ich selber nachschauen müsste würde ich es vermutlich nicht machen. So sieht man es von weitem und wenn ich mich entscheide die Änderung nicht rückgängig zu machen, dann will ich den hover nicht mehr sehen.</p>
	<p>P2: Kann man das Window ein bisschen grösser machen? (Anmerkung: es sind die hovers gemeint)</p>
	<p>P2: Ich finde es ein bisschen schwierig von der Darstellung her. Gerade mit den Pfeilen und so, das geht gar nicht. Das sieht aus wie eine pipeline, welche das Zeug durchstreamed. Mir hätte ein Balken gereicht, welcher sagt 30% aller requests gehen zu dieser Methode und ich brauche auch keine Zahlen auf 3 Kommastellen genau.</p>
	<p>P2: Ich weiss nicht ob man das in Eclipse kann aber bei Scala Worksheets hat man das Resultat schon auf gleicher Linie mit drin, wie ein Kommentar.</p>

	P2: Ich hätte nicht gerne Hovers sondern einen Kommentar gleich daneben, weil hovers sind mega... Ach, ich habe doch keine Geduld mit der Maus zuerst da drauf zu klicken und dann zu warten bis content angezeigt wird.
	P3: Bei neuen Aufrufen und deren Kostenauswirkungen finde ich die wichtigen Informationen ein bisschen zu sehr versteckt. Es ist zwar klar markiert, dass es Informationen von den Aufrufen gibt, aber es ist doch versteckt hinter einem hover und man muss es genauer inspizieren. Spannender wäre es direkt in der Zeile zu sehen, oder als Notification.
	P4: Ich finde es alleine schon interessant, dass ich schon farblich erkennen kann, dass dies ein externer service ist der aufgerufen wird. Ohne dass ich weiss dass es etwas kostet finde ich das gut, weil man manchmal eine Funktion verwendet, ohne dass es klar ist, dass es mit einem externen Service kommuniziert.
	P5: Ich finde die hovers eigentlich gut, eine sehr angenehme Variante. Es ist immer die Frage wie oft kommt das vor im code? Wenn jede zweite Zeile einen hover hat, dann stellt sich die Frage wo ich meinen Mauszeiger hinlege ohne dass etwas aufklappt. Ich finde das aber besser als eine Tastenkombination. Ich könnte mir eine farbliche Hervorhebung als schneller Überblick und zur Einschätzung von Grössenordnungen vorstellen. Im Sinne von heatmaps, rot wenn etwas.
	P6: Auch grafisch ästhetisch gemacht. Eine Möglichkeit, welche mir einfällt wäre, dass man die Predictions farblich unterscheidet vom Status Quo mit den Angaben wie oft eine Methode aufgerufen wird.
	P6: das hovering finde ich perfekt

Plugin Feedback Needs	P2: die Kosten pro Stunde.. whatever... ganz ehrlich, sagen wir du schreibst code und der lebt 2 Jahre lang, dann sagt das gar nichts aus. Wenn eine Instanz heute einen Franken pro Stunde kostet weiss ich nicht was es in 6 Monaten kostet. Die wirklichen Preise finde ich weniger spannend als das relative.
	P3: Wenn es in einem komplexen System bereits läuft und der Entwickler change requests bekommt, welche er umsetzen muss, dann ist es sehr interessant. Vor allem auch um dem Projektleiter zu melden, was dieser change überhaupt für einen Einfluss hat. Dieser kann dies meistens gar nicht einschätzen. Wenn der Entwickler dies aber meldet kann man bereits den change request anpassen oder ablehnen falls der impact zu gross ist. Vielleicht kann man alternative Lösungen finden, welche gar nicht evaluiert worden wären, sondern man hätte einfach am Schluss mehr bezahlt, weil reaktiv hat man meistens keine Zeit mehr grosse Anpassungen zu machen.
	P3: Die Annotation der Methoden, welche zeigt, welche anderen services diese überhaupt verwenden und was der load darauf ist enorm spannend - nicht nur um Kosten abzuschätzen, einfach als Verständnis vom ganzen System, vom ganzen Netzwerk. Das kann sowohl beim Debugging als auch bei Erweiterungen der Software sehr helfen.
	P3: Man sieht einen Kostentrend in Prozent und Kosten pro Stunde, aber das ist meistens nicht sonderlich spannend. Was heisst schon einen Dollar pro Stunde?
	P4: Die genauen Kosten sind für mich weniger interessant. Mehr im Sinne von was sind die Kosten in Anzahl von requests pro Sekunde.
	P4: Wenn wir neue Clients haben, welche unsere services aufrufen wollen dann ist das erste was wir fragen: wie viele requests pro Sekunde wirst du schicken von welchen Operationen?

	Wir wissen wie viele requests pro Sekunde unsere services bearbeiten können. Ich kann mir vorstellen, dass es hilfreich ist, wenn ich eine Analyse fahren kann, welche mir sagt, wenn mein service 10x/second aufgerufen wird, dann wird der andere 20x/second aufgerufen, weil es 2 RPC calls drin hat.
	P5: Ich sehe das auch recht nützlich in einer Situation wo man diverse Test Cases erstellen muss.
	P5: Man kriegt ein gutes Gefühl dafür wie alles zusammen hängt. Die andere Richtung ist vermutlich auch interessant: man sieht, dass eine Methode unwahrscheinlich stark beansprucht wird, "da kommen sehr viele calls hinein". Wo kommen denn die her? Die Information "was rufe ich auf?" ist interessant, aber die umgekehrte Frage ist eine die sich vermutlich häufiger ergibt.
	P5: Mir kommen 3 Situationen in den Sinn in welchen mich solche Daten interessieren: während der Entwicklung, um zu sehen welche Änderungen was bewirken. Die zweite Situation: man macht Lasttests, manuell oder automatisiert, und da kann man gezielt auf die Suche nach Veränderung gehen. Die kann ich mir sehr gut integriert in nightly builds vorstellen. Die dritte Situation ist, wenn sich das produktive System nicht so verhält wie wir das erwartet haben. Oft hat man dann notfallmässig die Fehler behoben aber die Kostenfrage wird gestellt und man steht vor der Aufgabe die Notfalloption effizienter zu machen. Dann braucht man aber genau die umgekehrte Information "was passiert wenn ich diesen Teil entferne?".
	P6: Ich finde also beides super cool, also zum einen, dass man eine Übersicht hat über den Status quo. Sozusagen, was wie fest weh tut. Das ist mega krass, das hat man ja sonst nie. Klar kann man das versuchen zu simulieren aber wenn man das wirklich genau mit runtime Daten sieht ist das natürlich super. Der Versuch, dass man den expected impact sieht ist auch richtig cool.

Plugin Feedback Assumptions	P4: Wenn wir neue Clients haben, welche unsere services aufrufen wollen dann ist das erste was wir fragen: wie viele requests pro Sekunde wirst du schicken von welchen Operationen? Wir wissen wie viele requests pro Sekunde unsere services bearbeiten können.

Plugin Feedback Integration	P5: So wie wir arbeiten wäre es am besten wenn man Continuous Integration oder Nightly Builds hätte und dann merkt, wenn es massive Kostenänderungen geben könnte mit einem Hinweis auf welche changes diese zurückzuführen sind. Bei signifikanten Änderungen könnte man dann nachhaken.
	P5: ... man macht Lasttests, manuell oder automatisiert, und da kann man gezielt auf die Suche nach Veränderung gehen. Die kann ich mir sehr gut integriert in nightly builds vorstellen ...
	P6: Bzgl. der Integration würde ich bevorzugen, dass nichts gepushed wird. Wir arbeiten immer mit drei Branches, development, acceptance und production. Wenn man zum Beispiel etwas auf den acceptance server deployed könnte man beim Continuous Integration neben den Quality Checks auch Einschätzungen abgeben wieviele Instanzen es dann in der Produktion brauchen wird. Somit als Zusatz zu allen statischen Codeanalysen. Es wäre aber natürlich cool, wenn

	man das direkt bei der Entwicklung sehen würde aber ich weiss halt nicht wie man das machen könnte.
	P6: Die Integration ins Eclipse finde ich sehr cool.
	P6: Ich finde es krass, dass es so gut integriert ist, das hovering finde ich perfekt. Weil sonst, wenn man diese Information als Web Service zur Verfügung hätte, würde man es am Schluss trotzdem nie verwenden.

Awareness (before)	P1: Die Awareness ist schon höher geworden. Ich habe mir auch immer wieder mal Statistiken angeschaut. Bei mir hatte es aber weniger Auswirkungen auf die Implementierung, mehr auf die Unterschiede von verschiedenen Ländern.

Awareness (after)	P1: Die Awareness selber ist schon alleine wegen dem orangen hover da.
	P4: Ich finde es alleine schon interessant, dass ich schon farblich erkennen kann, dass dies ein externer service ist der aufgerufen wird. Ohne dass ich weiss dass es etwas kostet finde ich das gut, weil man manchmal eine Funktion verwendet, ohne dass es klar ist, dass es mit einem externen Service kommuniziert.
	P5: Wenn man natürlich am entwickeln ist, dann ist es eine interessante Situation, um zu sehen "hier erzeuge ich einen riesigen load" und zwar finde ich das nicht erst einen Tag später raus oder in der Produktion. Man kann es sofort sehen aber es müssten bereits ein paar requests auf diesen service abgesetzt worden sein, d.h. man müsste alles es sehr schnell deployen.

Feature Suggestions	P1: Kann man pro Methode sagen "mich interessiert nur dieser request"?
	P1: Weiss man wie lange jeder Aufruf geht? Ich kann mir vorstellen, dass 1000 kleine Aufrufe weniger schlimm sind als 10 grosse.
	P1: Cool wäre, wenn der Entwickler nicht nur auf negative Aspekte hingewiesen wird, sondern auch, wenn dank einer Verbesserung Instanzen heruntergefahren werden können.
	P1: Es wäre schon gut, wenn man die Annotations zum Beispiel mit den Spring Annotations kombiniert, um so das Ganze stärker zu automatisieren. Je mehr Automatisierung desto besser, vor allem wenn es um Kosten geht, wo die Entwickler selber nicht betroffen sind.
	P3: Seit ich den Hover gesehen habe, der anzeigt welcher service welche Methode aufruft, hätte ich am liebsten eine ganze view von meinem Projekt, welche anzeigt welche services

	welche endpoints aufrufen und in was für einem Mass. Schon nur das zu sehen fände ich sehr spannend. Damit man auch problemlos sieht, ob ein Endpoint missbraucht wird.
	P5: Wie sehr sieht man "Was-wäre-wenn Simulationen"? Zum Beispiel "was passiert wenn ich diese Zeile auskommentiere, welche im produktiven Umfeld noch drin ist?". Wie sehr findet man die Auswirkung davon heraus? Bei einem laufenden System sieht man schnell "ah da kommen sehr viele Requests her". Dann schaut man sich den code an und sieht, dass es einfach ineffizient gelöst ist.
	P5: Ich könnte mir eine farbliche Hervorhebung als schneller Überblick und zur Einschätzung von Grössenordnungen vorstellen. Im Sinne von heatmaps, rot wenn etwas.
	P6: Eine overview-page würde mir auch interessieren. Da könnte man sicher krasse Statistiken generieren.

Willingness to install and configure the plugin	P1: Wenn es wichtig wäre Kosten und Performance einzuhalten, dann denke ich schon, dass ich bereit wäre Annotations zu setzen. Ich frage mich trotzdem ob es nicht einen Weg geben würde dies zu automatisieren.
	P1: bei komplizierten Applikationen, wo Kosten und Performance eine Rolle spielen. Vor allem weil ich dies von Hand viel weniger nachvollziehen kann.
	P1: Wenn ich jedoch von Hand immer wieder Annotations setzen muss, dann vergesse ich das und dann werde ich dies vermutlich erst nachholen, wenn ich merke dass ich irgendwo Kosten oder Performance sparen muss.
	P2: So als Debugger finde ich das alles sehr cool um insights zu bekommen. Aber ich glaube nicht, dass ich das ständig anlassen würde. Aber wenn ich nachschauen möchte wieso es nicht skaliert, oder bei einem Code Review finde ich das super.

Motivation to consider costs	P1: Cool wäre, wenn der Entwickler nicht nur auf negative Aspekte hingewiesen wird, sondern auch, wenn dank einer Verbesserung Instanzen heruntergefahren werden können. Im Stile von "you have saved 3 instances". So kann man die Entwickler ggf. auch motivieren um Kosten zu sparen (Positive Reinforcement). Man kann das sogar noch weiterdenken und Gamification reinbringen aber da bin ich meistens eher skeptisch. Das wäre vermutlich eher kurzfristig.
	P2: Gerade bei Codacy, wenn man sieht, dass man eine hohe zyklomatische Komplexität hat. Das macht dich hellhörig, gerade bei ressourcen-intensiven Applikation. Wenn ich nun einen score erhalten würde, würde mich das sicher motivieren, denn keiner bekommt gerne ein tiefes Rating, egal ob dies automatisiert ist oder persönlich.
	P2: Die Anzahl neuer Instanzen, welche hochgefahren werden müssen finde ich eine coole Metrik
	P2: Die wirklichen Preise finde ich weniger spannend als das relative. Dann wird es auch interessant für Leute die sich nicht für Kosten interessieren sondern nur für die Performance. Wo kann ich den throughput erhöhen? Wenn man sieht wo das bottleneck ist dann kann man optimieren, das ist cool. Und mit dem skaliert man auch.

	P3: Es treibt natürlich auch zur Optimierung. Es geht vielleicht auch ein bisschen gegen gewisse Konventionen, weil wenn man es konventionell implementiert kann es einen impact geben. Es könnte zum Beispiel auch die maintainability der Software beeinflussen
	P4: Ja natürlich, wenn ich natürlich schon im Voraus wüsste was es kosten wird und ich könnte versuchen zu optimieren damit wir niedrigere Kosten erreichen dann würde ich das schon machen. Die meisten Leute optimieren nur nach etwas was sie messen können, d.h. wenn man irgendwelche Metriken zur Verfügung hat dann wirst die optimieren und schaut, dass sie optimal sind. Und wenn man natürlich Kosten einer Instanz sehen würde könnte ich mir schon vorstellen, dass Leute dies anschauen würden.

Overhead	P2: Ich warte nicht.
	P2: Kann man das nachträglich machen? Im ersten compile step interessiert es mich, dass alles kompiliert, im nächsten Schritt vielleicht diese Metriken.
	P3: In der Entwicklungsphase sind build times enorm wichtig, in der Maintenance und Testing Phase dann weniger. Dann kommt es nur noch darauf an, dass es richtig läuft und nicht mehr dass es schnell gebuildet ist.
	P3: Bei dynamischen Instanzen hingegen spielt es wiederum eine Rolle. Wenn gewisse changes dynamisch propagiert werden müssen dann spielt es eine Rolle aber meistens sind es vorgefertigte binaries die gestartet werden dann spielt es nicht so eine grosse Rolle.
	P5: Sobald man etwas beobachtet gibt es einen overhead aber aus produktiven Daten kann man die meisten Rückschlüsse ziehen und es sind sicherlich auch die interessantesten Informationen. Selber etwas zusammenzustellen und zu simulieren kann man immer aber wenn es im produktiven Umfeld trotzdem abweicht ist es immer schwierig an Informationen zu kommen. Man findet vielleicht noch heraus, welcher service am meisten load generiert aber nicht wo und warum.
	P6: Was ich als Entwickler ein bisschen Angst hätte ist, dass dieses Monitoring ein grosser overhead bedeutet.
	P6: Das ist egal. Mir als Entwickler ist das egal, aber der overhead auf dem produktiven Server durch das Monitoring ist viel wichtiger. Monitoring overhead ist viel schlimmer.
	P6: Darf ich mal schauen wie die Performance ist? Ah easy! Und es blockiert einem nicht beim arbeiten? Das ist schnell, voll easy.

B.4 Quantitative Evaluation

Eclipse Build Time Measurements, Part 1

Build Type	Full Build Checkout Service 9 Sources		Full Build Shop Service 6 Sources		Full Build Payment Service 2 Sources	
	JavaMvn	CostAll	JavaMvn	CostAll	JavaMvn	CostAll
Builder	80	9932	40	4088	30	2025
	71	9580	52	3487	16	2084
	81	9485	32	4024	14	1885
	58	9785	33	4197	27	2100
	67	9564	34	3545	16	1957
	61	9770	42	3970	17	1971
	95	10334	39	3976	13	1919
	54	9842	34	3980	14	1935
	49	9798	35	3974	14	1998
	74	9911	39	3995	16	1989
	73	9681	41	3520	16	1851
	72	8688	32	4316	16	1927
	66	9346	33	3470	13	2112
	61	9679	29	3988	19	1839
	57	9368	32	3427	16	1868
	58	8984	33	4423	17	1944
	59	9544	31	3866	18	2129
	91	10190	33	3865	17	1857
	75	9451	29	3714	16	2175
	57	9474	32	4091	16	2051
Average in ms	67.95	9420.30	35.25	3895.80	17.05	1980.80
JavaMvn CostAll	Java and Maven Builder CostBuilder All Features					

Eclipse Build Time Measurements, Part 2. Varying the number of Client Invocation Annotations

Build Type Builder	Full Build Shop Service 6 Sources 2 method decl. annotations 0 invocation annotation		Full Build Shop Service 6 Sources 2 method decl. annotations 1 invocation annotation		Full Build Shop Service 6 Sources 2 method decl. annotations 3 invocation annotation	
	CostAll		JavaMvn	CostAll	CostAll	
	3227		40	4088	6750	
	3299		52	3487	6366	
	3586		32	4024	6339	
	3625		33	4197	7297	
	3672		34	3545	8974	
	3548		42	3970	6776	
	3503		39	3976	12078	
	3543		34	3980	8517	
	3381		35	3974	6363	
	3718		39	3995	7322	
	3406		41	3520	6397	
	3579		32	4316	8492	
	3417		33	3470	12104	
	3714		29	3988	6928	
	3585		32	3427	5876	
	3458		33	4423	7204	
	3722		31	3866	6955	
	3490		33	3865	7046	
	3322		29	3714	5902	
	3499		32	4091	7234	
Average	3514,70		35,25	3895,80	7546,00	
JavaMvn	Java and Maven Builder					
CostAll	CostBuilder All Features					

Eclipse Build Time Measurements, Part 3, varying number of Method Declaration Annotations

Build Type	Full Build Shop Service 6 Sources		Full Build Shop Service 6 Sources		Full Build Shop Service 6 Sources
	1 method decl. annotation 1 Invocation annotation	CostAll	2 method decl. annotations 1 Invocation annotation	CostAll	3 method decl. annotation 1 Invocation annotation
Builder		3309		4088	4424
		3707	40	3487	4263
		3839	52	4024	4168
		3395	32	4197	4377
		3676	33	3545	4461
		3442	34	3970	4043
		3727	42	3976	4356
		3626	39	3980	4404
		3665	34	3974	4236
		3365	35	3995	4740
		3440	39	3520	4149
		3564	41	4316	4822
		3456	32	3470	4138
		3372	33	3988	4336
		3938	29	3427	4123
		3950	32	4423	4807
		3965	33	3866	4353
		3474	31	3865	4068
		3531	33	3714	4051
		3413	29	4091	4316
Average	3592,70		35,25	3895,80	4331,75

JavaMvn
CostAll

Java and Maven Builder
CostBuilder All Features

B.5 CD Contents

- **master_thesis.pdf**
Master Thesis as PDF
- **abstract.txt**
Abstract in English
- **zusfsg.txt**
Abstract in German
- **src/**
The source code
- **evaluation/**
Data of interview study and quantitative evaluation

Bibliography

- [ABDDP13] Giuseppe Aceto, Alessio Botta, Walter De Donato, and Antonio Pescapè. Cloud monitoring: A survey. *Computer Networks*, 57(9):2093–2115, 2013.
- [AF15] Martin L Abbott and Michael T Fisher. *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise*. Number 9780134031385. Pearson Education, 2015.
- [AFG⁺09] Michael Armbrust, O Fox, Rean Griffith, Anthony D Joseph, Y Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, et al. M.: Above the clouds: a berkeley view of cloud computing. 2009.
- [Ama15a] Amazon. Amazon EC2 Instance Purchasing Options. <http://aws.amazon.com/ec2/purchasing-options/>, 2015. Accessed on July 23, 2015.
- [Ama15b] Amazon. AWS Pricing. <http://aws.amazon.com/pricing/>, 2015. Accessed on June 30, 2015.
- [ARAEBA13] May Al-Roomi, Shaikha Al-Ebrahim, Sabika Buqrais, and Imtiaz Ahmad. Cloud computing pricing models: a survey. *International Journal of Grid & Distributed Computing*, 6(5):93–106, 2013.
- [BFKB⁺14] Dario Bruneo, Thomas Fritz, Sharon Keidar-Barner, Philipp Leitner, Federica Longo, Clarissa Marquezan, Andreas Metzger, Klaus Pohl, Antonio Puliafito, Andrei Roth, et al. Cloudwave: where adaptive cloud management meets devops. In *Computers and Communication (ISCC), 2014 IEEE Symposium on*, pages 1–6. IEEE, 2014.
- [BGG15] Martin Brandtner, Emanuel Giger, and Harald Gall. Sqa-mashup: A mashup framework for continuous integration. *Information and Software Technology*, 65:97–113, 2015.
- [Bin07] David Binkley. Source code analysis: A road map. In *2007 Future of Software Engineering*, pages 104–119. IEEE Computer Society, 2007.

- [Ble13] Alex Blewitt. *Eclipse 4 Plug-in Development by Example Beginner's Guide*. Packt Publishing Ltd, 2013.
- [Bos15] Christian Bosshard. Feedback Driven Development - Bringing Runtime Metrics to the Developer. Master's thesis, University of Zurich, 2015.
- [BPSZ10] Silvia Breu, Rahul Premraj, Jonathan Sillito, and Thomas Zimmermann. Information needs in bug reports: improving cooperation between developers and users. In *Proceedings of the 2010 ACM conference on Computer supported cooperative work*, pages 301–310. ACM, 2010.
- [BVD15] Kelly Blincoe, Giuseppe Valetto, and Daniela Damian. Facilitating coordination between software developers: A study and techniques for timely and efficient recommendations. *IEEE Transactions on Software Engineering*, 2015.
- [BWZ15] Len Bass, Ingo Weber, and Liming Zhu. *DevOps: A Software Architect's Perspective*. Addison-Wesley Professional, 2015.
- [CGS03] Abhishek Chandra, Weibo Gong, and Prashant Shenoy. Dynamic resource allocation for shared data centers using online measurements. In *Quality of Service—IWQoS 2003*, pages 381–398. Springer, 2003.
- [CHZ⁺07] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J Van Wijk, and Arie Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *Program Comprehension, 2007. ICPC'07. 15th IEEE International Conference on*, pages 49–58. IEEE, 2007.
- [CLFG14] Jürgen Cito, Philipp Leitner, Thomas Fritz, and Harald Gall. The Making of Cloud Applications – An Empirical Study on Software Development for the Cloud. *CoRR*, abs/1409.6, 2014.
- [CLG⁺15] Jürgen Cito, Philipp Leitner, Harald C Gall, Aryan Dadashi, Anne Keller, and Andreas Roth. Runtime metric meets developer-building better cloud applications using feedback. *PeerJ PrePrints*, 3:e1214, 2015.
- [CZG11] Cuiting Chen, Andy Zaidman, and Hans-Gerhard Gross. A framework-based runtime monitoring approach for service-oriented software systems. In *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, pages 17–20. ACM, 2011.
- [Dig15] Digitalocean. Simple Pricing. <https://www.digitalocean.com/pricing/>, 2015. Accessed on June 30, 2015.
- [DK⁺14] Rajdeep Dua, Dharmesh Kakadia, et al. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

- [Doc15] Docker. What is Docker? <https://www.docker.com/whatisdocker>, 2015. Accessed on June 30, 2015.
- [EFB01] Tzilla Elrad, Robert E Filman, and Atef Bader. Aspect-oriented programming: Introduction. *Communications of the ACM*, 44(10):29–32, 2001.
- [FAO10] Ikki Fujiwara, Kento Aida, and Isao Ono. Applying double-sided combinational auctions to resource allocation in cloud computing. In *Applications and the Internet (SAINT), 2010 10th IEEE/IPSJ International Symposium on*, pages 7–14. IEEE, 2010.
- [Fow04] Martin Fowler. *UML distilled: a brief guide to the standard object modeling language*. Addison-Wesley Professional, 2004.
- [Fow14] Martin Fowler. Microservices. <http://martinfowler.com/articles/microservices.html>, 2014. Accessed on May 01, 2015.
- [FWWH13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large software landscapes: The explorviz approach. In *Software Visualization (VISSOFT), 2013 First IEEE Working Conference on*, pages 1–4. IEEE, 2013.
- [GGW10] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *Network and Service Management (CNSM), 2010 International Conference on*, pages 9–16, Oct 2010.
- [Gu12] Zhongxian Gu. Capturing and exploiting fine-grained ide interactions. In *Proceedings of the 34th International Conference on Software Engineering*, pages 1630–1631. IEEE Press, 2012.
- [HF10] Jez Humble and David Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.
- [HK14] Peitsa Hynninen and Marjo Kauppinen. A/b testing: A promising tool for customer value evaluation. In *Requirements Engineering and Testing (RET), 2014 IEEE 1st International Workshop on*, pages 16–17. IEEE, 2014.
- [HKR13] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in cloud computing: What it is, and what it is not. In *ICAC*, pages 23–27, 2013.
- [IKLL12] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical prediction models for adaptive resource provisioning in the cloud. *Future Generation Computer Systems*, 28(1):155–162, 2012.
- [JHJ⁺10] Gueyoung Jung, M.A. Hiltunen, K.R. Joshi, R.D. Schlichting, and C. Pu. Mistral: Dynamically managing power, performance, and adaptation cost in cloud infrastructures. In *Distributed Computing Systems (ICDCS), 2010 IEEE 30th International Conference on*, pages 62–73, June 2010.

- [JS14] Brendan Jennings and Rolf Stadler. Resource management in clouds: Survey and research challenges. *Journal of Network and Systems Management*, pages 1–53, 2014.
- [JTB11] Bahman Javadi, Ruppia K Thulasiram, and Rajkumar Buyya. Statistical modeling of spot instance prices in public cloud environments. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 219–228. IEEE, 2011.
- [KDV07] Andrew J Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th international conference on Software Engineering*, pages 344–353. IEEE Computer Society, 2007.
- [KHH⁺01] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G Griswold. An overview of aspectj. In *ECOOP 2001—Object-Oriented Programming*, pages 327–354. Springer, 2001.
- [Kla14] Viktor Klang. Microservices. <http://klangism.tumblr.com/post/80087171446/microservices>, 2014. Accessed on June 25, 2015.
- [KM06] Mik Kersten and Gail C Murphy. Using task context to improve programmer productivity. In *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 1–11. ACM, 2006.
- [Kra14] Lucas Krause. *Microservices: Patterns and Applications*. 2014.
- [KT06] Thomas Kuhn and Olivier Thomann. Abstract Syntax Tree. http://www.eclipse.org/articles/article.php?file=Article-JavaCodeManipulation_AST/index.html, 2006. Accessed on August 03, 2015.
- [LD03] Michele Lanza and Stéphane Ducasse. Polymetric views-a lightweight visual approach to reverse engineering. *Software Engineering, IEEE Transactions on*, 29(9):782–795, 2003.
- [LGW⁺08] Xuezheng Liu, Zhenyu Guo, Xi Wang, Feibo Chen, Xiaochen Lian, Jian Tang, Ming Wu, M Frans Kaashoek, and Zheng Zhang. D3s: Debugging deployed distributed systems. In *NSDI*, volume 8, pages 423–437, 2008.
- [LL13] Remo Lemma and Michele Lanza. Co-evolution as the key for live programming. In *Proceedings of the 1st International Workshop on Live Programming*, pages 9–10. IEEE Press, 2013.
- [Mar03] Robert Cecil Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall PTR, 2003.
- [MBV⁺10] Philippe Moret, Walter Binder, Alex Villazón, Danilo Ansaloni, and Abbas Heydarnoori. Visualizing and exploring profiles with calling context ring charts. *Software: Practice and Experience*, 40(9):825–847, 2010.

- [MC⁺01] J Moc, David Carr, et al. Understanding distributed systems via execution trace data. In *Program Comprehension, 2001. IWPC 2001. Proceedings. 9th International Workshop on*, pages 60–67. IEEE, 2001.
- [MCC04] Matthew L Massie, Brent N Chun, and David E Culler. The ganglia distributed monitoring system: design, implementation, and experience. *Parallel Computing*, 30(7):817–840, 2004.
- [MG11a] Mario Macías and Jordi Guitart. A genetic model for pricing in cloud computing markets. In *Proceedings of the 2011 ACM Symposium on Applied Computing*, pages 113–118. ACM, 2011.
- [MG11b] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.
- [Mic15] Microsoft. Azure pricing. <http://azure.microsoft.com/en-us/pricing/>, 2015. Accessed on June 30, 2015.
- [MKA⁺13] Mika V Mantyla, Foutse Khomh, Bram Adams, Emelie Engstrom, and Kim Petersen. On rapid releases and software testing. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 20–29. IEEE, 2013.
- [MLW11] Shicong Meng, Ling Liu, and Ting Wang. State monitoring in cloud datacenters. *Knowledge and Data Engineering, IEEE Transactions on*, 23(9):1328–1344, 2011.
- [MMC⁺02] Jonathan Maletic, Andrian Marcus, Michael L Collard, et al. A task oriented view of software visualization. In *Visualizing Software for Understanding and Analysis, 2002. Proceedings. First International Workshop on*, pages 32–40. IEEE, 2002.
- [MMVP13] Rizwan Mian, Patrick Martin, and Jose Luis Vazquez-Poletti. Provisioning data analytic workloads in a cloud. *Future Generation Computer Systems*, 29(6):1452–1458, 2013.
- [MRLD09] Anton Michlmayr, Florian Rosenberg, Philipp Leitner, and Schahram 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*, pages 1–6. ACM, 2009.
- [New15] Sam Newman. *Building Microservices*. "O'Reilly Media, Inc.", 2015.
- [NLG⁺03] Harvey B Newman, Iosif C Legrand, Philippe Galvez, Ramiro Voicu, and Catalin Cirstoiu. Monalisa: A distributed monitoring service architecture. *arXiv preprint cs/0306096*, 2003.
- [NSS14] Dmitry Namiot and Manfred Sneps-Sneppé. On micro-services architecture. *International Journal of Open Information Technologies*, 2(9):24–27, 2014.

- [PDCB15] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. Efficient virtual machine sizing for hosting containers as a service. 2015.
- [Rac15] Rackspace. Pay-as-you-go Pricing. <http://www.rackspace.com/cloud/public-pricing>, 2015. Accessed on June 30, 2015.
- [RCL09] Bhaskar Prasad Rimal, Eunmi Choi, and Ian Lumb. A taxonomy and survey of cloud computing systems. In *INC, IMS and IDC, 2009. NCM'09. Fifth International Joint Conference on*, pages 44–51. Ieee, 2009.
- [Rei03] Steven P Reiss. Visualizing java in action. In *Proceedings of the 2003 ACM symposium on Software visualization*, pages 57–ff. ACM, 2003.
- [RGO06] Arnon Rotem-Gal-Oz. Fallacies of distributed computing explained. URL <http://www.rgoarchitects.com/Files/fallacies.pdf>, page 20, 2006.
- [RHB⁺12] David Röthlisberger, Marcel Härry, Walter Binder, Philippe Moret, Danilo Ansaloni, Alex Villazon, and Oscar Nierstrasz. Exploiting dynamic information in ides improves speed and correctness of software maintenance tasks. *Software Engineering, IEEE Transactions on*, 38(3):579–591, 2012.
- [RHV⁺09] David Röthlisberger, Marcel Härry, Alex Villazón, Danilo Ansaloni, Walter Binder, Oscar Nierstrasz, and Philippe Moret. Augmenting static source views in ides with dynamic metrics. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 253–262. IEEE, 2009.
- [SDGW08] Rolf Stadler, Mads Dam, Alberto Gonzalez, and Fetahi Wuhib. Decentralized real-time monitoring of network-wide aggregates. In *Proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*, page 7. ACM, 2008.
- [SP11] Parnia Samimi and Ahmed Patel. Review of pricing models for grid & cloud computing. In *Computers & Informatics (ISCI), 2011 IEEE Symposium on*, pages 634–639. IEEE, 2011.
- [SSN⁺11] Rahul Singh, Prashant Shenoy, Maitreya Natu, Vaishali Sadaphal, and Harrick Vin. Predico: A system for what-if analysis in complex data center applications. In *Proceedings of the 12th International Middleware Conference, Middleware '11*, pages 120–139, Laxenburg, Austria, Austria, 2011. International Federation for Information Processing.
- [STT⁺12] Bhanu Sharma, Ruppa K Thulasiram, Parimala Thulasiraman, Saurabh K Garg, and Rajkumar Buyya. Pricing cloud compute commodities: a novel financial economic model. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*, pages 451–457. IEEE Computer Society, 2012.

- [Swa14] Paul Swartout. *Continuous Delivery and DevOps—A Quickstart Guide*. Packt Publishing Ltd, 2014.
- [TKS⁺11] Konstantinos Tsakalozos, Herald Kllapi, Eva Sitaridi, Mema Roussopoulos, Dimitris Paparas, and Alex Delis. Flexible use of cloud resources through profit maximization and price discrimination. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 75–86. IEEE, 2011.
- [VHWH12] André Van Hoorn, Jan Waller, and Wilhelm Hasselbring. Kieker: A framework for application performance monitoring and dynamic software analysis. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, pages 247–248. ACM, 2012.
- [VRMB11] Luis M Vaquero, Luis Roderio-Merino, and Rajkumar Buyya. Dynamically scaling applications in the cloud. *ACM SIGCOMM Computer Communication Review*, 41(1):45–52, 2011.
- [WL08] Richard Wettel and Michele Lanza. Codecity: 3d visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, pages 921–922. ACM, 2008.
- [WWDR14] Yi Wang, Patrick Wagstrom, Evelyn Duesterwald, and David Redmiles. New opportunities for extracting insights from cloud based ides. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 408–411. ACM, 2014.
- [XL12] Hong Xu and Baochun Li. Maximizing revenue with dynamic cloud pricing: The infinite horizon case. In *Communications (ICC), 2012 IEEE International Conference on*, pages 2929–2933. IEEE, 2012.
- [YBDS08] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *Grid Computing Environments Workshop, 2008. GCE’08*, pages 1–10. IEEE, 2008.
- [ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of internet services and applications*, 1(1):7–18, 2010.
- [Zel07] Andreas Zeller. The future of programming environments: Integration, synergy, and assistance. In *2007 Future of Software Engineering*, pages 316–325. IEEE Computer Society, 2007.