Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit Nr. 110

Performance Anomaly Detection in Microservice Architectures Under Continuous Change

Thomas F. Düllmann

Course of Study: Softwaretechnik

Examiner: Dr.-Ing. André van Hoorn (Prof.-Vertr.)

Supervisor: Dr.-Ing. André van Hoorn (Prof.-Vertr.)

Commenced: June 14, 2016

Completed: January 3, 2017

CR-Classification: C.2.4, H.3.4

Abstract

The idea of DevOps and agile approaches like Continuous Integration (CI) and microservice architectures are bocoming more and more popular as the demand for flexible and scalable solutions is increasing. By raising the degree of automation and distribution new challenges in terms of application performance monitoring arise because microservices are possibly short-lived and may be replaced within seconds. The fact that microservices are added and removed on a regular basis brings new requirements in the way anomaly detection is conducted as these changes could also be the cause for anomalies.

This work proposes to take information about such events into account to improve the anomaly detection quality. Additionally, a meta model for microservice environments and supplemental tooling was developed that can be used to generate actual microservice environments from an instance of such a meta model. The generation tool also comprises the means to instrument the generated microservices with Kieker to collect monitoring data and generate supplemental files to be able to create Docker images and Kubernetes configuration files which allow to run the microservices on a Kubernetes cluster.

In the evaluation such a generated microservice environment is run in a lab experiment with delay injections. The obtained data were then used to evaluate a customized version of the RanCorr approach which was modified to fit in the microservice context and a newly developed basic approach named Event-aware Anomaly Revisor (EAR) that takes event information into account when conducting anomaly detection. The evaluation results showed, that the customized RanCorr approach could not satisfy the expectations in terms of improvement of the anomaly detection results while the EAR approach could slightly improve the anomaly detection quality.

Kurzfassung

Die Idee von DevOps und agilen Ansätzen wie Continuous Integration (CI) und Microservice Architekturen bekommen immer populärer, da die Nachfrage nach flexiblen und skalierbaren Lösungen steigt. Durch den steigenden Grad der Automatisierung und Verteilung entstehen neue Herausforderungen im Bereich der Überwachung von Anwendungsperformanz, da Microservices sehr kurzlebig sein können und es vorkommen kann, dass sie binnen weniger Sekunden ersetzt werden. Die Tatsache, dass Microservices regelmäßig hinzugefügt und entfernt werden führt zu neuen Anforderungen in der Art wie Anomalieerkennung durchgeführt wird, da die genannten Änderungen ebenfalls den Grund für Anomalien darstellen können.

Diese Arbeit schlägt vor Informationen über solche Ereignisse in die Anomalieerkennung einzubeziehen. Zusätzlich wurde ein Metamodell für Microservice-Umgebungen mit zusätzlichen Werkzeugen zur Erzeugung von echten Microservice-Umgebungen anhand einer Instanz des Metamodells. Das Generierungswerkzeug bietet außerdem die Mittel, um die generierten Microservices zur Erfassung von Messdaten mit Kieker zu instrumentieren und ist in der Lage zusätzliche Dateien erzeugen, die es ermöglichen aus den Microservices Docker Images zu erzeugen und Kubernetes Konfigurationsdateien, die es ermöglichen die Microservices auf einem Kubernetescluster auszuführen.

In der Evaluation wurde eine solche Microservice-Umgebung generiert, die in einem Laborexperiment mit injizierten Verzögerungsinjektionen ausgeführt wurden. Die dadurch ermittelten Daten wurden daraufhin genutzt, um zum Einen die für den Microservicekontext angepasste Version des RanCorr-Ansatzes und die Basisversion eines neu entwickelten Ansatzes namens Event-aware Anomaly Revisor (EAR), der Informationen über Ereignisse bei der Erkennung von Anomalien berücksichtigt. Die Evaluationsergebnisse haben gezeigt, dass der angepasste RanCorr-Ansatz die Erwartungen im Sinne der Verbesserung der Anomalieerkennungsergebnisse nicht erfüllen konnte während der EAR-Ansatz die Erkennungsqualität zumindest leicht verbessern konnte.

Contents

1. Introduction 1
   1.1. Motivation 1
   1.2. Thesis Structure 4

2. Foundations and Related Work 7
   2.1. Microservices 7
   2.2. DevOps 8
   2.3. Continuous Deployment 8
   2.4. Time Series 9
   2.5. Anomaly Detection 10
   2.6. Root-cause Detection 11
   2.7. Reducing Monitoring False Alarms with Context Knowledge 11
   2.8. Workload Behavior Analysis and Prediction 12
   2.9. EGADS by Yahoo 12
   2.10. Anomaly Detection in Commercial APM Tools 13
   2.11. Tooling 14

3. Modeling, Generation, Extraction of Microservice Architectures 17
   3.1. Requirements 17
   3.2. Metamodel Structure 18
   3.3. Microservice Environment Generation 25
   3.4. Model Extraction 29

4. Change-aware Anomaly Detection 31
   4.1. Overview 31
   4.2. Types of Anomalies 32
   4.3. Assumptions Regarding Changes 34
   4.4. Multi-tier Anomaly Evaluation 35
   4.5. Event-aware Anomaly Revisor (EAR): Incorporating the Acquired Data to Reduce False Positives 37

4.6. Implementation 40

5. Evaluation 45
5.1. Evaluation Goals 45
5.2. Evaluation Methodology 46
5.3. Evaluation Setup 46
5.4. Experiment Settings 49
5.5. Description of Results 52
5.6. Discussion of Results 55

6. Conclusion 59
6.1. Summary 59
6.2. Discussion 59
6.3. Future Work 60

A. Ecore Model Diagrams 63
A.1. Metamodel Structure 63
A.2. Deployment Model 64
A.3. Microservice Model 65
A.4. Dependency Model 66
A.5. Monitoring Model 67

Bibliography 69

List of Figures

1.1. Visualization of an initialization anomaly 3
1.2. Visualization of a continuous anomaly 3
1.3. Visualization of initialization and continuous anomaly 4

2.1. Extraction of a time series from measurement data 9

3.1. Intended process for modeling, generation and extraction 18
3.2. Structure of the metamodel 18
3.3. Structure of the deployment model 20
3.4. Example instance of the deployment model 20
3.5. Structure of the microservice meta model (MicroserviceRepository) . . . 21
3.6. Structure of the microservice meta model 22
3.7. Instance example the microservice meta model 23
3.8. Structure of the dependency model 23
3.9. Example for an instance of the dependency model 24
3.10. Structure of the monitoring model 25
3.11. Example for an instance of the monitoring model 25

4.1. Visualization of an initialization anomaly 33
4.2. Visualization of a anomaly caused by code changes 34
4.3. Visualization of a real anomaly 34
4.4. Microservice anomaly detection results 35
4.5. Microservice anomaly detection results after including architectural information 36
4.6. Microservice anomaly detection results after including architectural and event information 37
4.7. Basic approach using a factor applied to the anomaly score 38
4.8. Advanced approach using a gradually reducing anomaly score after an event was detected 39

5.1. System Under Test (SUT) 47

5.2. SUT structure generated from the monitoring data 47
5.3. Evaluation Setup for the generated SUT 48
5.4. Results of the prediction performance of the ΘPAD approach visualized by a ROC curve 53
5.5. Results of the prediction performance of the customized RanCorr approach visualized by a ROC curve 54
5.6. Results of the Event-aware Anomaly Revisor (EAR) approach in a ROC curve 54
5.7. Example for EAR in effect 56

A.1. Ecore model visualization of the metamodel structure 63
A.2. Ecore model visualization of the deployment model 64
A.3. Ecore model visualization of the microservice model 65
A.4. Ecore model visualization of the dependency model 66
A.5. Ecore model visualization of the monitoring model 67

List of Tables

5.1. Injections in the experiment 51
5.2. Statistics for the different approaches for an anomaly threshold of 0.1 . . 52
5.3. Comparison of the different approaches for an anomaly threshold of 0.1 (1: ΘPAD, 2: Customized RanCorr, 3: EAR) 52

List of Listings

3.1. Example of a Dockerfile for a generated microservice 27
3.2. Kubernetes service file example 27
3.3. Kubernetes controller file example 28

List of Algorithms

4.1. Customized RanCorr Algorithm 42
4.2. EAAR Basic Algorithm 43

Chapter 1

Introduction

1.1. Motivation

In order to keep applications scalable and reliable, developers tend to move from monolithic application architectures to microservice architectures [BHJ16]. Microservices are small services that have a small feature set but are interconnected with many other microservices to provide complex services like web applications. Companies like Netflix [Net16], Zalando [Zal16] and Otto [Ott16] are actively pushing this topic having most of their infrastructure migrated to microservices. These provide better scalability and enable the possibility to distribute the application over multiple physical and virtual systems which also infers more reliability. Furthermore, it is easier to adopt agile DevOps-oriented methodologies as it is less error-prone to upgrade and test only certain components without having to replace the application as a whole.

DevOps approaches in combination with Continuous Integration (CI) and continuous deployment foster a high degree of automation leading to short release cycles and fast and frequent deployments. For example, at Amazon.com the mean time between deployments on weekdays is about 11.6 seconds as stated in [Jen11].

Though microservice architectures have many advantages over the monolithic style, they also increase the complexity in terms of management and monitoring for operations teams. By distributing the application over multiple (virtualized) machines the monitoring aspect becomes a tough task.

In the area of application performance monitoring it is common to monitor response times of method calls to detect problems before they pose a real threat to the productivity of the system. Due to the agility of distributed microservice environments where services are upgraded multiple times a day and where several versions of an application can be deployed at the same time, this creates several new challenges. As every update may

change the response time behavior of methods within applications, this could result in many false alarms of the application monitoring systems which use historical data and possibly forecasting approaches to detect anomalies. One example for an approach that uses historical data is the baselining approach by Mayle et al. [MRCW01]. It takes measurements from the past to calculate the baseline of metrics to deduct a threshold from it. When a measurement surpasses the threshold it is treated as an anomaly.

According to Bass, Weber, and Zhu [BWZ15] monitoring under continuous change is a challenge to be addressed due to the new practices in modern computing environments (e.g., microservices, continuous deployment, and DevOps). To tackle this problem it is necessary to be able to know of such version upgrades and to change the alerting behavior accordingly.

Due to the fact that microservices are started and stopped on a frequent basis, the following events were identified that may have an impact on the false positive anomaly detection rate.

Update-induced ramp-up anomalies
When a service is updated in a microservice environment, this usually means that it is replaced by a new version of the service. As the microservice should be available without interruptions, usually a new microservice instance is started before the old version is shut down. Due to the startup of a new service instance, one can observe increased load on the system as constructors are executed and variables are initialized. Such tasks affect the response time of a service which could lead to anomaly alerts if a classical anomaly detection approach would be used for monitoring.

Figure 1.1 shows the response time behavior of a microservice that is replaced with a newer one (update). Due to the initialization phase of the new microservice instance the response times are increased and go back to normal after some time. Due to the anomaly threshold, the increased response times caused by the initialization phase of the microservice would be treated as anomalies.

Update-induced behavior changes
Another challenge is the effective change that an update introduces to the applications behavior. For example, if an additional command is added to the source code of an application, the execution time of that command adds up to the total response time. This change may lead to an anomaly alert as the changed behavior exceeds the defined threshold.

Figure 1.2 shows the response time behavior change after the implementation of a microservice has been changed. As this visualization does not include the initialization

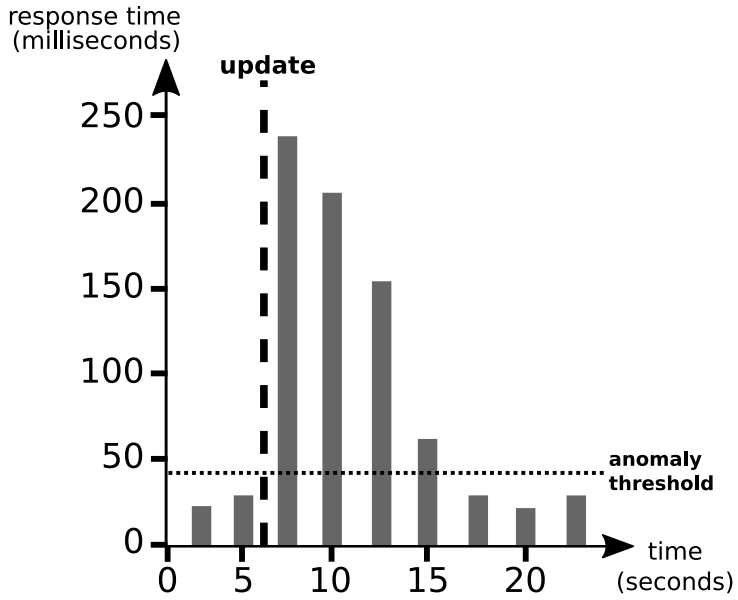

Figure 1.1.: Visualization of an initialization anomaly

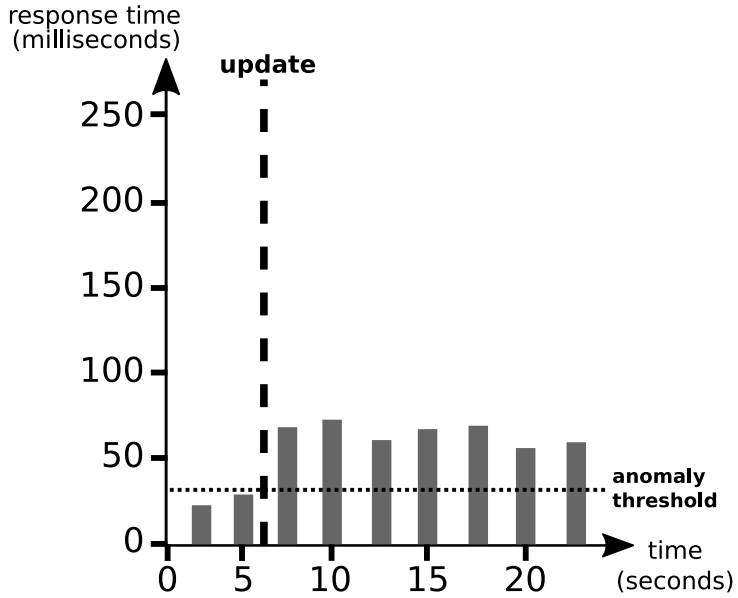

Figure 1.2.: Visualization of a continuous anomaly

anomaly it is clear to see. When starting a new microservice with code changes impacting the response times, it would rather look like depicted in Figure 1.3 where the impact of the behavior changes is masked by the initialization anomaly and is only visible after the initialization is finished. Due to the anomaly threshold, the increased response times caused by changes of the implementation of the microservice would be treated as anomalies.

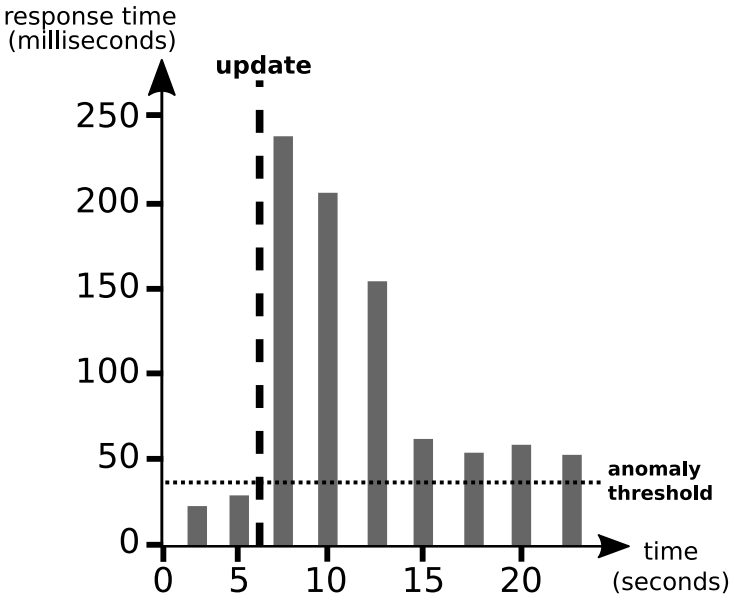

Figure 1.3.: Visualization of initialization and continuous anomaly

Goals
The overall goal of this thesis is to reduce the false positive anomaly detection rate in comparison to the rates of the combination of the ΘPAD [Bie12; Fro13] and RanCorr [MRHH09] approaches. This improvement is intended to be achieved by taking event information into account when determining whether changes in the behavior are actual anomalies. In case such update events occur, the anomaly score should be reconfigured for some time to allow the initialization phases to be executed without raising an anomaly alert.

1.2. Thesis Structure

The thesis is structured as follows.

Chapter 2 (Foundations and Related Work) provides an overview over the topics that are important for the understanding of the thesis and presents the work that is related to the topic of this thesis.

Chapter 3 (Modeling, Generation, Extraction of Microservice Architectures) presents the meta model for microservice environments that was created as an underlying building block of the following steps throughout this thesis. Furthermore, it describes the aspects of the generation of microservice environments from a model instance of the said model.

Chapter 4 (Change-aware Anomaly Detection) describes the customizations to the RanCorr approach in order to use it in our context and the proposed approach to improve the anomaly detection quality in microservice environments by taking event information into account.

Chapter 5 (Evaluation) specifies the evaluation of the proposed approach and its results which eventually are discussed.

Chapter 6 (Conclusion) concludes the thesis by summarizing the experiences and results that were gained throughout the thesis. Finally, some aspects that would be relevant for future work are described.

The supplemental material for this thesis is publicly availabe at [Dül17].

Chapter 2

Foundations and Related Work

This chapter provides the foundations that are relevant in the context of this thesis and presents approaches that are related to the topic.

2.1. Microservices

Microservices are part of a rising complex that comprises several new processes and technologies in the field of software development. According to Newman [New15], microservices can be described as *"small, autonomous services that work together"*. They enable several other approaches in this field by projecting the Unix philosophy, coined by Doug McIlroy, to software development processes.

> *"This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together."*
> – Doug McIlroy, [Sal94]

James Lewis and Martin Fowler defined microservices as a suite of small services, each running in their own processes and being independently deployable having a bare minimum of centralized management in [Fow15]. According to them, each microservice does only have to adhere to the Application Programming Interface (API) requirements without having to be written in a specific programming language or using specific data storage technology which leads to flexibility regarding the choice of technologies.

Microservices today mainly use Representational State Transfer (REST) as a general way to communicate with other microservices. The term REST was coined by Fielding et al. [FT02] and is a technique broadly used in the web service and microservice context. It uses the combination of Hypertext Transfer Protocol (HTTP) verbs and the Uniform

Resource Locator (URL) to specify the actions and the resources the action should be executed with.

Cockcroft [Coc15] developed a tool named Spigo[1] which can simulate microservice environments and their behavior as well as visualize them. In contrast to the aim of this thesis, these simulated microservices can not be deployed to actual environments suited for microservices (e.g., cluster infrastructures).

2.2. DevOps

DevOps stands for the approach of bringing two groups within the software life cycle together that were separated before: the development and the operations team [BWZ15; KBS14; PWC15]. The classic way to develop and operate software was to have the development team develop an application and have it deployed by the operations team without too much communication before handing over the application. Afterwards, the operations team had to keep the application running without having had any contact with the application beforehand or having had any saying during the planning and development process.

While DevOps in general does not imply actual activities or processes, it encourages the teams to work together or melt into one combined team with development and operations tasks. The idea of cooperation of DevOps is intended to start right at the beginning of a software development process when defining the requirements (e.g., it is advised that the operations team is an equal stakeholder for such a projects) [BWZ15; EAD14].

2.3. Continuous Deployment

Continuous deployment stands for the idea of deploying all commits to a Version Control System (VCS) that pass all tests in the CI pipeline directly to the production environment. Due to this automation it is possible that defective versions are deployed. In order to prevent the whole system from failing when defective versions are deployed, several mechanisms exist to keep the impact low. Examples for such mechanism are rolling updates and canary releasing [HF10]. By using the advantages of continuous deployment and microservices it is possible to deploy new versions of microservices in rapid succession while having the possibility to control the impact of the changes. An

[1] https://github.com/adrianco/spigo

additional way to get more detailed insight is to monitor the behavior of the services (e.g., in terms of performance). This employs the ability to detect anomalies.

2.4. Time Series

Observations that are obtained by measurements are not necessarily occurring in an equidistant manner (e.g., measurements of requests to a online shop). As it is desirable to have single values per time frame for statistical analysis, it is possible to aggregate the measurements of a time frame into one.

According to Shumway and Stoffer [SS10] a time series can be defined as a collection of random variables indexed according to the order they are obtained in time. Projecting this on response time measurements, a time series of response times would be a set of tuples containing the time of the measurement and the actual measurement value. Time series can be obtained by polling data on a regular basis (e.g., executing a request every minute to measure the response time) or instrumenting all requests and calculate the average of all measurements within a specified time frame to get equidistant data points. Figure 2.1 depicts the extraction of a time series from measurement data. The left part of the figure represents the actual measurements that are monitored and the dashed lines mark the time windows that are used to split the data into equidistant data points. The right part shows the resulting time series after having accumulated the measurements within each of the areas using the arithmetic mean. The resulting dataset is a time series with equidistant data points.

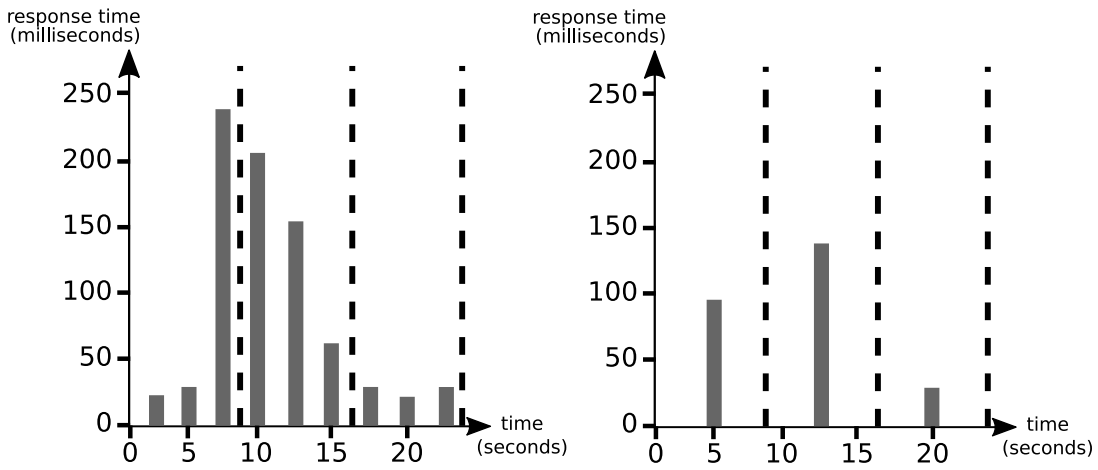

Figure 2.1.: Extraction of a time series from measurement data

2.5. Anomaly Detection

According to Chandola, Banerjee, and Kumar [CBK09] anomalies are patterns in data that do not conform to the normal behavior and can be indicators for problems in the corresponding domain of measurement (e.g., break-down of a system).

In general baselining means that the existing data (e.g., response times) is used to learn the normal behavior of an application. In order to decide whether the current situation is an anomaly or not, thresholds can be set. This can happen either manually by setting the threshold to a fixed value or can be determined automatically using the normal behavior as an indicator.

There are also approaches that use forecasting models to predict the next measurements based on the historical data. Applying such an approach could be to mark actual measurements as anomalies if the grade of deviation from the forecast is too high.

2.5.1. Metrics for the Evaluation of Anomaly Detection

In order to be able to measure whether an approach is an improvement, the following metrics will be used in the evaluation.

True Positives (TP)
True positives are all detections that were correctly labeled as anomalies.

False Positives (FP)
False positives are all detections that were falsely labeled as anomalies.

False Negatives (FN)
False negatives are the measurements that were wrongly labeled as being no anomalies even though they should have been labeled as anomalies.

Accuracy (ACC)
The accuracy property calculates the quality of the classification related to the total of all measurements.

$$ACC = \frac{TP + FP}{\text{all measurements}} \tag{2.1}$$

Positive Predictive Value (PPV)
The PPV measures the amount of correctly predicted values in comparison to the total of positively predicted values.

$$PPV = \frac{TP}{TP + FP} \tag{2.2}$$

True Positive Rate (TPR)
The TPR measures the amount of correctly predicted values in contrast to all values that are actually correct.

$$TPR = \frac{TP}{TP + FN} \tag{2.3}$$

F-measure (F_1)
The F-measure (or F-score) provides the harmonic mean of the positive predictive value and the true positive rate.

$$F_1 = 2 \cdot \frac{PPV \cdot TPR}{PPV + TPR} \tag{2.4}$$

2.6. Root-cause Detection

By monitoring applications it is comparably easy to find the part of the application that shows high response times, but it is not necessarily the source of the problem. Often this part of the code itself also calls some other methods that might be the cause for the issue instead. To find the actual source of the problem, root-cause analysis is used. The approach of Marwede et al. [MRHH09] analyzes the monitored execution traces and response times of an application to determine the source of the problem. Although the original implementation by Marwede et al. does not support data analysis at runtime, this approach was adapted to analyze data at runtime in a development project at the University of Stuttgart [DEE+14].

2.7. Reducing Monitoring False Alarms with Context Knowledge

Xu et al.[XZF+15] deduct information about possible false alarms directly from system metrics (e.g., CPU utilization) and training machine learning models with such metrics.

Both approaches can be combined to silence anomaly alerts that are raised by classical monitoring tools.

In the work of Farshchi et al.[FSWG15] statistical methods are used to deduct a regression-based model that resembles possible correlations between logged operation events and cloud resource metrics. The said model is then used to detect anomalies in the system.

In contrast to the approach proposed in this thesis, Xu et al. and Farshchi et al. mainly focus on system metrics and not application-specific measurements. Additionally they do not take architectural information (e.g., dependencies between applications or services) into account. The approach described in this thesis does not use machine learning approaches at all.

2.8. Workload Behavior Analysis and Prediction

Rohr [Roh15] applied anomaly detection on response times measurements that are normalized by removing the impact of workload on those measurements. While this approach is architecture-aware it is not designed to be used at runtime of the monitored application. The approach of Rathfelder et al. [RBKR12] mainly focuses on workload-changes and workload prediction which is not the scope of this thesis.

2.8.1. Hora

The Hora approach proposed by Pitakrat et al. [POHG16] employs architectural models and monitoring data from a system to calculate failure probabilities using Bayesian networks. This architecture-aware approach can be used to predict the failure of a distributed system at runtime and gives the opportunity to prepare for the failure to keep the downtimes low. In comparison to the approach of this thesis, Hora uses architectural knowledge but does not take change events into account.

2.9. EGADS by Yahoo

EGADS by Laptev, Amizadeh, and Flint [LAF15] is a Java library for anomaly detection for large datasets. It supports several time series and forecasting models and is available

on GitHub[2]. In contrast to the approach of this thesis it does not take event information into account.

2.10. Anomaly Detection in Commercial APM Tools

There are several vendors of Application Performance Monitoring (APM) solutions that are advertised to be able to monitor microservice environments. As the source code or the algorithms for dependency tracking and anomaly detection is not publicly available it is not possible to inspect their approaches. The following list is an alphabetically sorted list of well-known providers for commercial APM solutions that claim to support microservice or similar environments. Of course, this list is not exhaustive.

- AppDynamics[3]
- Dynatrace[4]
- CA APM[5]
- Instana[6]

2.10.1. ΘPAD(x)

The ΘPAD [Bie12] approach and its extension, the ΘPADx [Fro13] approach have been implemented as filter sets for the Kieker monitoring framework. In general, these approaches use historical data to forecast the next measurement value. Depending on the difference between the forecasted and the actual value, it is determined whether an anomaly is present.

These approaches are designed for systems that do not (or seldom) change their behavior. They neither take architectural information into account, nor event information (e.g., update events) is considered when deciding about whether an anomaly is present or not.

As parts of the ΘPAD approaches are used in this thesis, important aspects will be detailed here.

[2] https://github.com/yahoo/egads
[3] https://www.appdynamics.com/product/application-performance-management/
[4] https://www.dynatrace.com/
[5] http://ca.com/apm
[6] https://www.instana.com/

Aggregation
The first step in the process of the application of the ΘPAD approach is to create a time series from the received measurement data. To do so, the data within defined time windows is aggregated. The result of this step is a time series containing one triple which contains an identifier for the monitored resource, a timestamp and the response time per time window per identifier.

Forecasting
Based on the time series data obtained from the aggregation step, the next value is forecasted using one of the available forecasting methods that can be configured in the corresponding Kieker filter. The output of this component is the forecast and the actually measured value.

Anomaly Calculation
Using the output of the forecasting component, the anomaly calculation filter calculates an anomaly score ranging from zero to one based on the difference between the forecasted value and the actual measurement.

Anomaly Detection
The anomaly detection component uses an initially defined threshold and compares it with the calculated anomaly score to decide whether a measurements is to be treated as an anomaly.

2.11. Tooling

Apart from general approaches, the following tools were used in this thesis.

2.11.1. Kieker

Kieker is an open source performance monitoring framework [HWH12]. It provides components for instrumenting applications, logging application performance data and running analyses on the collected data. As Kieker being an open source project and employing a pipe and filter structure it is possible to extend it by adding custom filters (e.g., to do custom analyses on the data acquired).

2.11.2. Docker

Docker[7] is a tool to create virtual-machine like environments named “containers” that are logically separated from the host operating system. Due to the fact that the processes of the containers are run directly on the host operating system, Docker provides a lightweight way to run applications in a separate environment. Such containers can provide immutable environments for building applications or can be used to run microservice applications that can be deployed on multiple physical hosts.

2.11.3. Eclipse Ecore

Eclipse Ecore is a modeling framework which allows creating meta models that can be created and instantiated in a graphical editor while making sure that the model constraints are not violated [SBPM09]. Furthermore, it allows to use such instances in a programmatic context.

2.11.4. Kubernetes

Kubernetes[8] by Google is a cloud platform that allows the deployment of Docker containers in a distributed and scalable manner. It can be run on multiple physical hosts which are interconnected to represent a cluster. The platform itself takes care of the actual distribution over the different nodes and the routing. As this is a state of the art cloud platform that supports Docker images, this environment is used to run our evaluation on. The Kubernetes team also provides a tool named minikube[9] which allows running a small sized Kubernetes node in a virtual machine.

[7] https://www.docker.com
[8] http://kubernetes.io/
[9] https://github.com/kubernetes/minikube

Chapter 3

Modeling, Generation, Extraction of Microservice Architectures

In order to formalize our approach, we propose metamodels for different aspects of microservice environments.

3.1. Requirements

Meta models of microservice are intended to provide a way to define the structure of a microservice environment from different points of view (e.g., deployment, microservice types, dependencies). Apart from being able to represent the state of a microservice environment it should be usable in a programmatic context. By providing this, it is possible to extend applications to use this model or its instances and stay compatible with the original model. Based on said model real microservices should be generated that can be compiled and actually deployed as a real microservice environment. To be able to monitor these microservices, they should be instrumented upon creation. This allows to extract the data from the services and could be a mean to extract either the monitoring data or even the whole microservice environment to again create a model of it. This could be useful if changes were made to the deployed microservices and wants to use the model of it for further analysis. Based on the extracted monitoring data, it is possible to conduct anomaly detection or other post-processing steps.

Figure 3.1 depicts the steps that are intended to be possible with the outcome of this thesis.

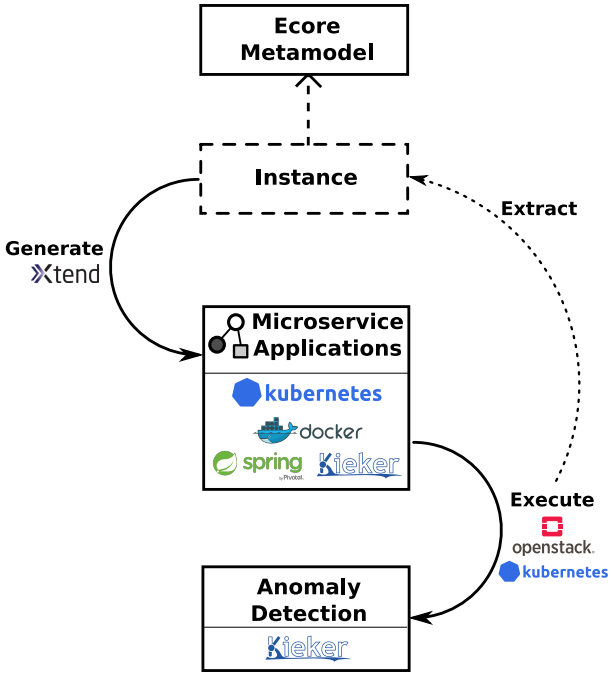

Figure 3.1.: Intended process for modeling, generation and extraction

3.2. Metamodel Structure

In this section the different meta models are illustrated and described. The images of the actual Ecore diagrams can be found in Appendix A.

To put the different aspects of the model into context, there is one model to interconnect the different views on the model. This structure is depicted in Figure 3.2.

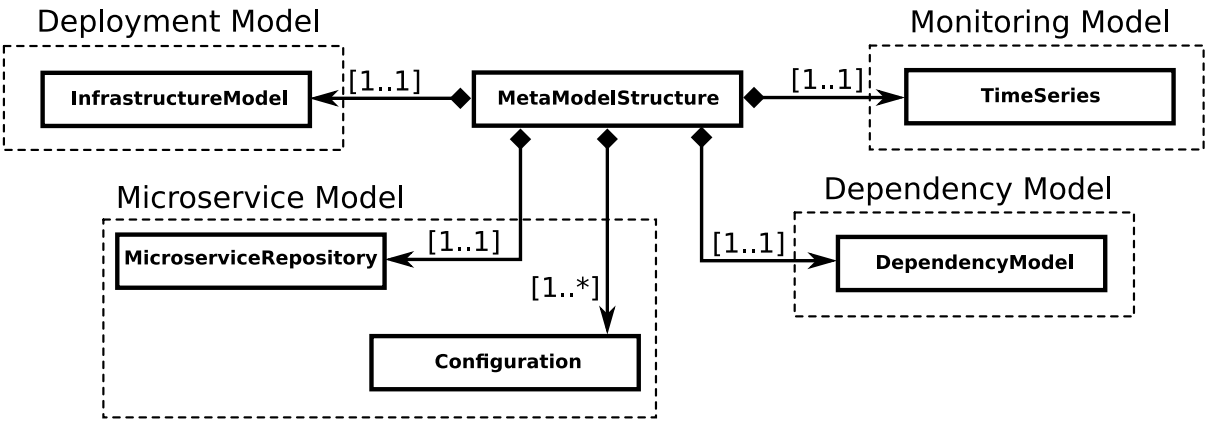

Figure 3.2.: Structure of the metamodel

The deployment model describes the environment where the microservices are deployed to. It is described in detail in Section 3.2.1. The model contains exactly one deployment model.

The microservice model (Section 3.2.2) represents the set of available microservices (MicroserviceRepository) and the state of an actual microservice setup (Configuration).

The microservice repository contains all existing microservice types and their versions. There is always exactly one MicroserviceRepository. As the state of a microservice environment can change over time, there can be multiple Configurations.

The dependency model (Section 3.2.3) describes the dependencies between microservices. Dependencies are based on the microservice type and REST operations and define which REST operations of which microservice types call which other microservices.

The monitoring model (Section 3.2.4) represents the monitoring data that has been extracted from running the model. This model is not intended to be prepopulated but added after the environment has been executed. By supplementing the model with this data, the model can then be evaluated as a whole.

3.2.1. Deployment Model

The deployment model represents the architectural context the microservices are deployed in. Throughout this thesis the assumption is made that the underlying system architecture is not changed. Therefor, the model has exactly one deployment model. It is depicted in Figure 3.3.

The main building blocks of the deployment model are the physical hosts because every layer of abstraction is somehow based on a physical machine. Therefor, we assume that there is always at least one physical host present in the infrastructure model which is the parent element connected to the overall meta model.

Microservices can be run on an abstract execution environment which can either be a host or a container. Any kind of host can contain any number of containers (e.g., Docker container) while every container is run on exactly one abstract host. Again, hosts can be either virtual hosts or physical hosts. While physical hosts are the actual physical machines (e.g., servers in a data center), the virtual hosts are virtual machines that run on physical hosts. Therefor, a physical can contain any number of virtual hosts while a virtual host always runs on exactly one physical host. In conclusion, there are several environments where the microservice instances can be executed on, while they always depend on physical hosts that either contain the instances themselves or have them nested inside of virtual machines and/or containers.

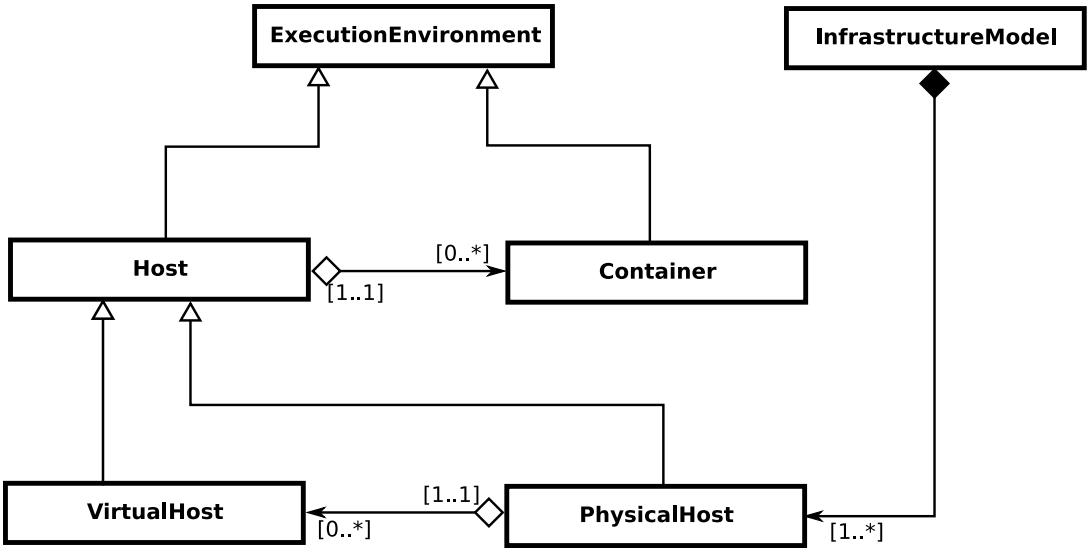

Figure 3.3.: Structure of the deployment model

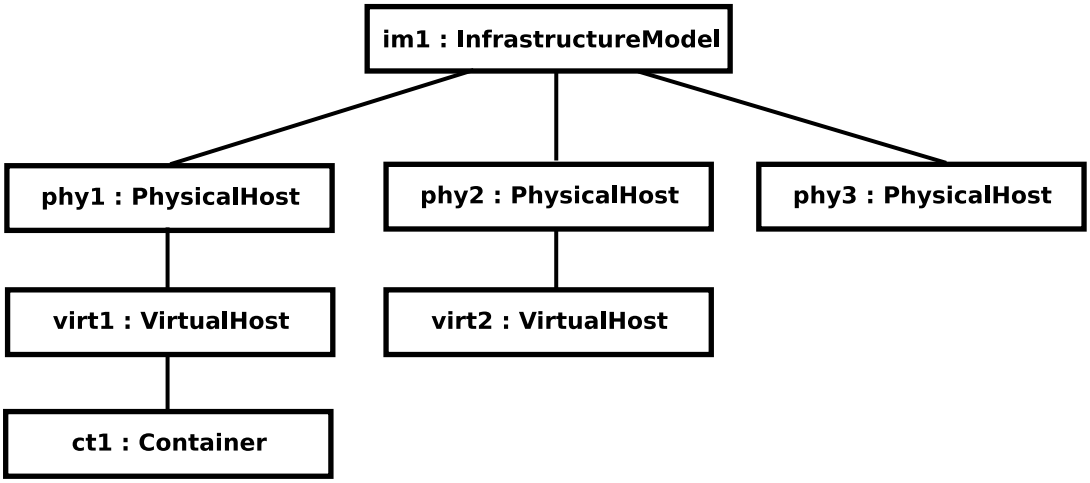

Figure 3.4.: Example instance of the deployment model

Figure 3.4 shows an example instance of the deployment model. One can see that the infrastructure model instance `im1` contains three physical host instances which contain different numbers of nested elements. While the physical host `phy1` contains a virtual host which again contains a container, the physical host `phy2` hosts a virtual host only. According to the model each level (physical host, virtual host and container) can run a microservice instance as they all can act as an execution environment.

3.2.2. Microservice Model

The microservice model shown in Figure 3.5 represents two aspects of a microservice environment. Apart from the microservice and versions that are available for deployment contained in a microservice repository, also the state of microservice instantiation at a certain point in time can be held in the configuration.

A microservice repository, contains of at least one microservice type which is further defined by a set of REST operations which define what methods can be called and how. There may be multiple versions of one microservice type. While the implementation of the microservice type may vary, we assume that the set of REST operations is fixed.

For every microservice type there is at least one This part of the microservice model is independent from actual instances of microservices that are deployed.

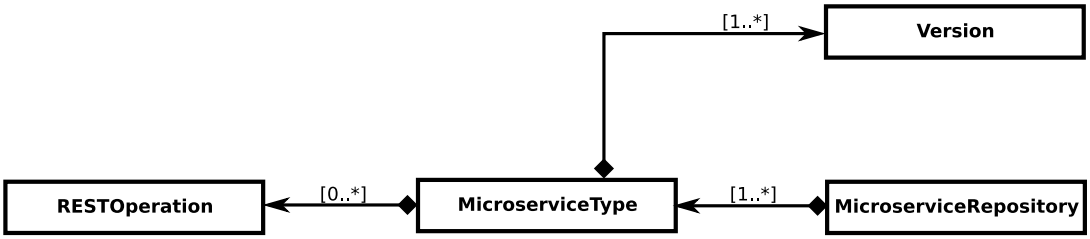

Figure 3.5.: Structure of the microservice meta model (MicroserviceRepository)

The part contained by configuration objects represents the state of actual instances. In Figure 3.6 One configuration can contain multiple microservices that are of exactly one microservice type with exactly one version. While the operations that are available are defined by the microservice type, the address where the microservice can be reached depends on the execution environment. Therefor the endpoint is a dynamic property that may be different between multiple instances of one microservice type. One microservice instance always runs in exactly one execution environment.

To get a better understanding on how the elements are interconnected when instanciated, Figure 3.7 depicts one configuration instance containing two microservices and a microservice repository holding two microservice types. As the relations between the objects is rather complex, the objects that are contained (even transitively) by the configuration object `c1` are drawn with a dashed border. The objects contained by the microservice repository object `msr1` are visualized with a solid border. In this instance of the microservice repository contains two microservice types `mt1` and `mt2`. Each of these has at least one version object. As multiple microservice types can have similar version identifiers, one version object can be referenced by more than one microservice type (as seen for `v1`). Furthermore, every microservice type may contain REST operations. A

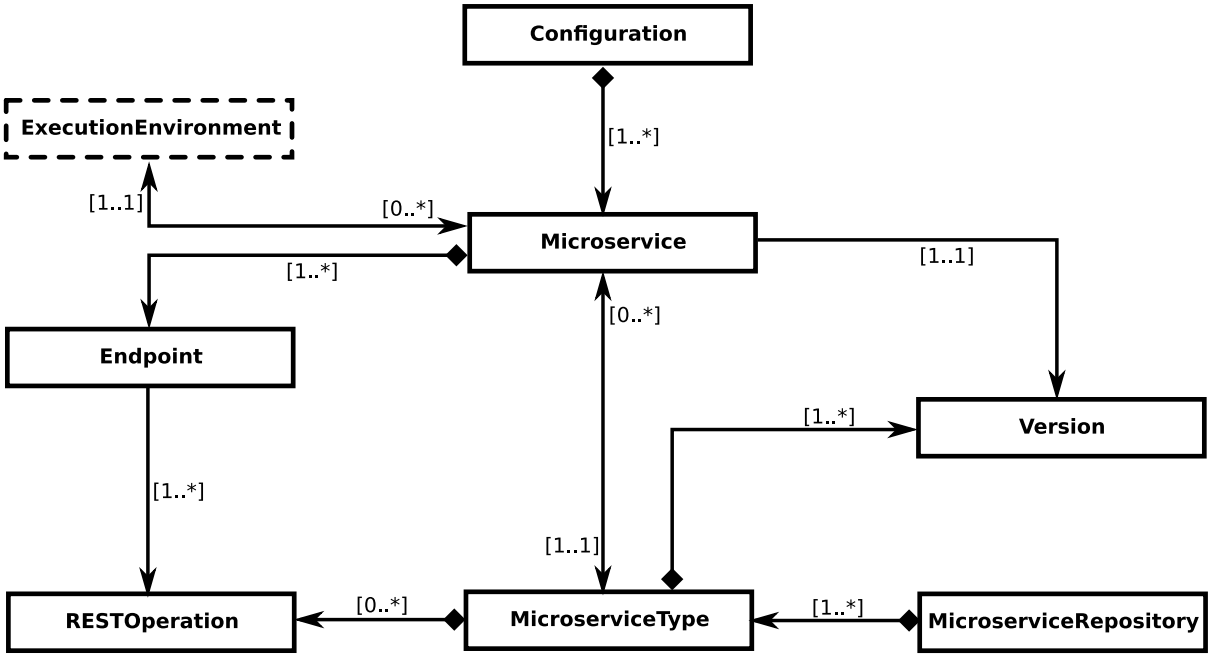

Figure 3.6.: Structure of the microservice meta model

configuration object, `c1` in this case, contains the actual microservice instances that are instances of specific microservice types with a fixed version. Object Constraint Language (OCL) constraints ensure that a microservice can only be associated with a version that the referenced microservice type contains. In this example it would not be possible for `ms2` to be associated with `mt2` and the version object `v2` as there is no version `v2` for the microservice type `mt2`. Additionally, every microservice has to be associated with at least one endpoint object that defines the connection parameters that are required to call the microservice instance using the REST operation API that is referenced by the associated endpoint.

3.2.3. Dependency Model

The dependency model defines the dependencies between microservices and the operations they provide. Figure 3.8 depicts the structure of the model. The dependency model item holds all dependencies that are in place. It can contain any number of OperationToOperationCallingDependencies that represent the actual dependencies between the operations of microservice types. Every OperationToOperationCallingDependency identifies the calling item by the type of the microservice, the corresponding version and the operation that calls the operation of another service. The version is required in the model as the dependencies throughout version changes may vary. The called service is

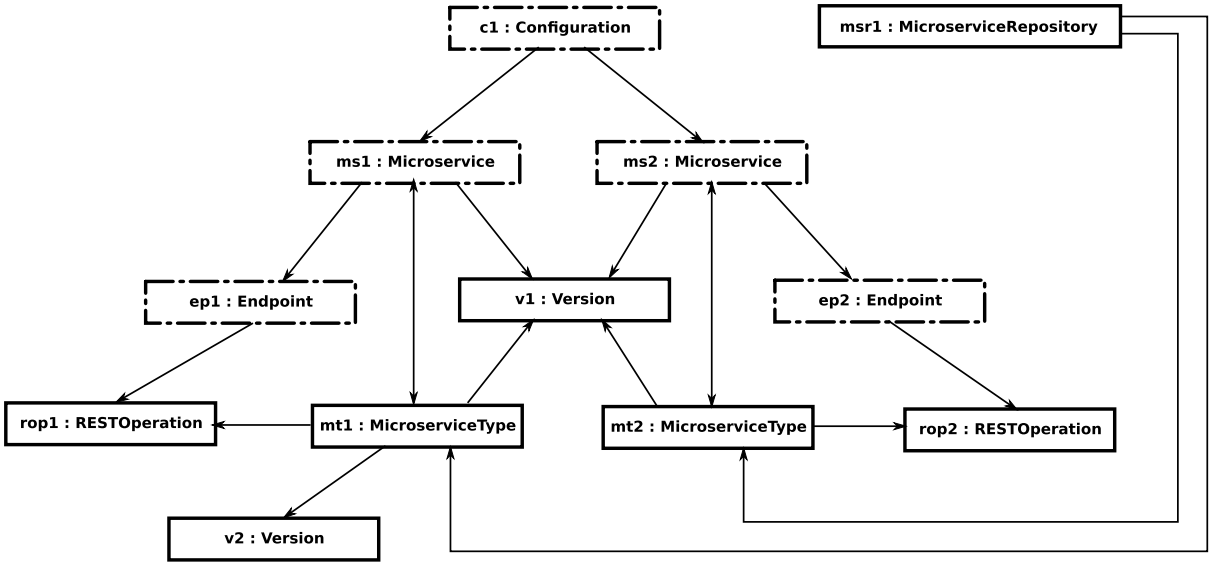

Figure 3.7.: Instance example the microservice meta model

defined by the microservice type and the operation. This way it is known which operation of which microservice type with which version calls which operation of another (or even the same) microservice. Every OperationToOperationCallingDependency defines the relation between exactly two microservices and operations. To realize multiple dependencies, it is necessary to define further OperationToOperationDependencies. Note that the version of the called microservice is not part of the dependency, as microservices are only identified by their type, but not their version. That way it cannot select the version of the called microservice.

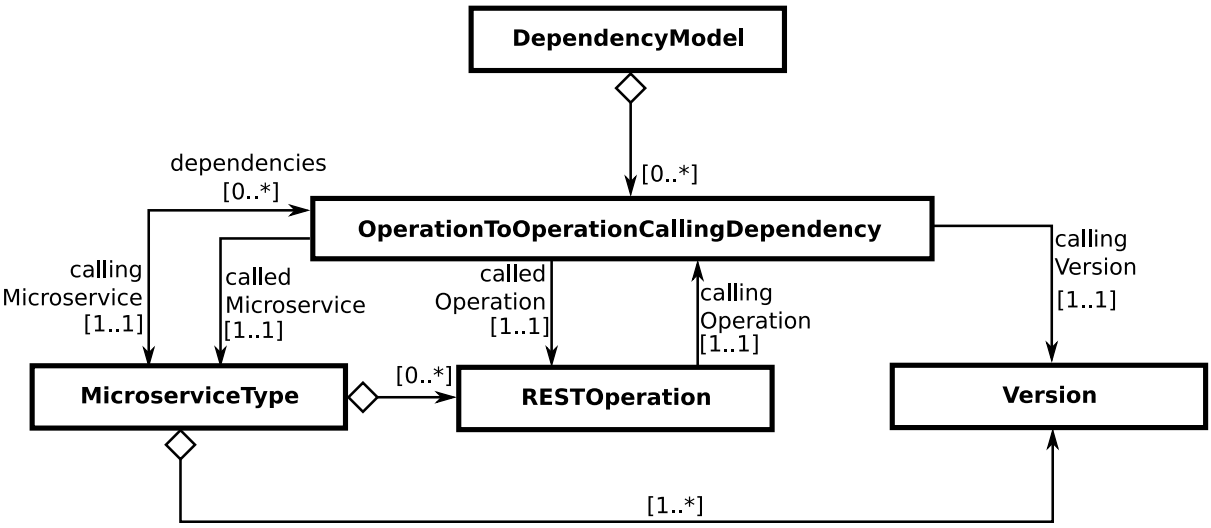

Figure 3.8.: Structure of the dependency model

To have a better understanding of the dependency model, Figure 3.9 gives an instanciated example. In the figure the operation `ro1` of microservice `mt1` in version `v1` calls the operation `ro2` of the microservice `mt2` .

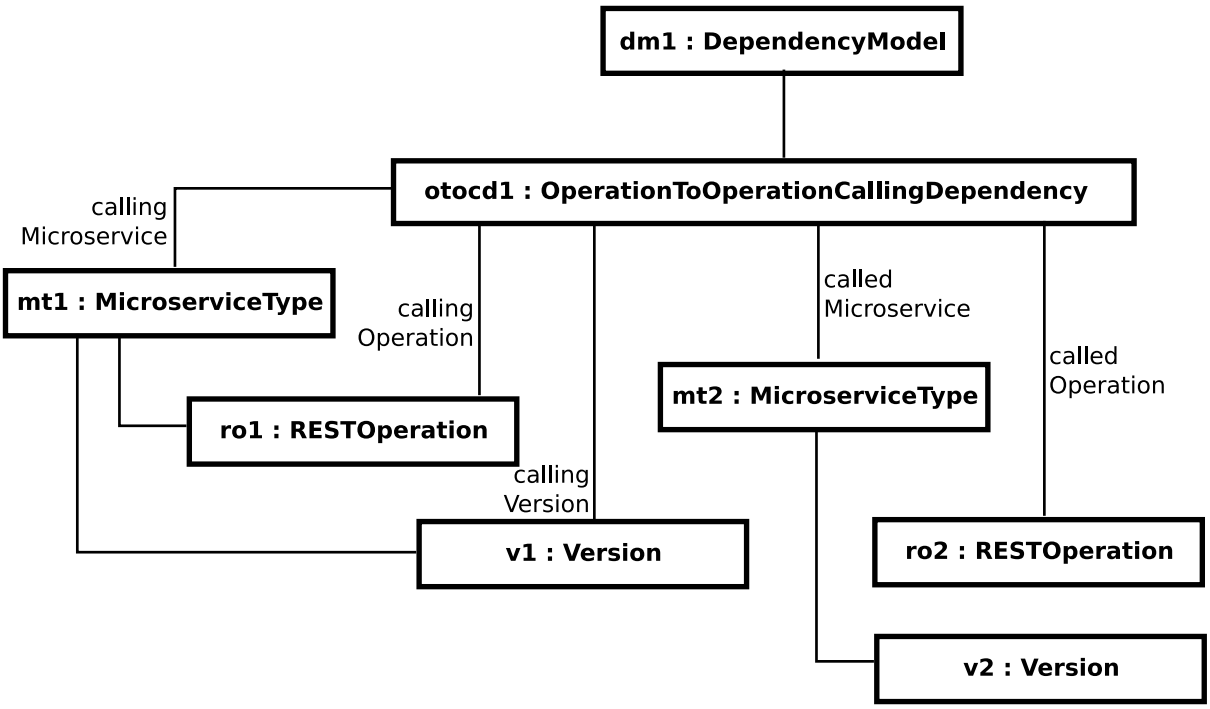

Figure 3.9.: Example for an instance of the dependency model

3.2.4. Monitoring Model

The monitoring model depicted in Figure 3.10 represents the monitoring data that is related to the items of the microservice model. The time series contains any number of time series points which represent actual monitoring data. In the case of this model a specific MicroserviceOperationTimeSeriesPoint was defined which is related to one endpoint and one REST operation which uniquely identifies an operation of a specific microservice instance.

To illustrate the model, Figure 3.11 shows an example instance of the monitoring model. One time series instance contains three MicroserviceOperationTimeSeriesPoints. `motsp1` is associated with the REST operation that is realized by the endpoint `ep2` while `motsp2` is associated with another operation realized by the same endpoint. `motsp3` is the measurement of another pair of endpoint and REST operation.

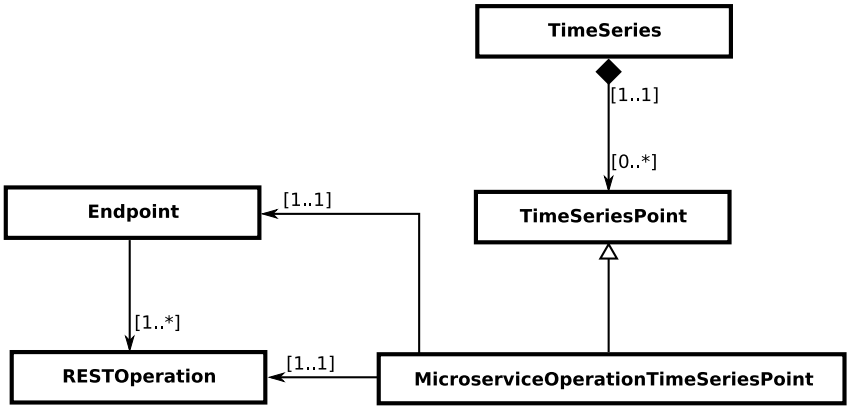

Figure 3.10.: Structure of the monitoring model

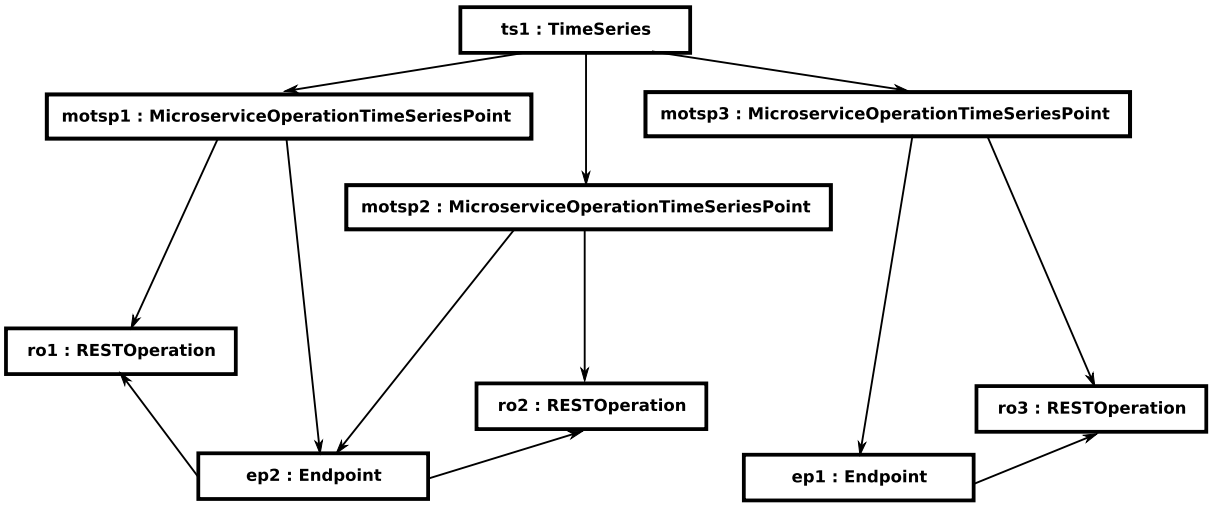

Figure 3.11.: Example for an instance of the monitoring model

3.3. Microservice Environment Generation

To generate a microservice environment, an instance of the Ecore meta model is created that represents the intended microservice environment. The result of this is an *xmi*-file. This file is used to generate the microservice environment using a Java-based programming language supporting template expressions named Xtend[1]. To create the evaluation environment for this thesis a template was created which generates a microservice environment based on the given instance of the model.

As the focus of this thesis is not the generation template itself but the environment resulting for it for evaluation purposes, the template does not use and generate all properties

[1] https://www.eclipse.org/xtend/

of the model. It is mainly limited to the aspects of the microservice types, instances and their dependencies. Currently it also does not use the REST verb in the generation process but uses the "GET" verb for all generated calls to other microservices.

To be able to execute and deploy the generated code, it has to be embedded in a web server and needs further configuration files to be deployed as a Docker image and also in a Kubernetes cluster. Therefor, in addition to the application code itself, the wrapping web server and the configuration files for Docker and Kubernetes are also generated which is explained in more detailed in the following sections.

3.3.1. Microservices

Based on the information extracted from the instance of the Ecore model, the microservice wrapper is generated by creating several artifacts that finally lead to a running microservice. The first artifact is for Apache Maven[2] which is a project dependency and build tool and allows to use a pom-file to pull in dependencies from a central repository that are needed to build the microservice. Through references in the generated code and by obtaining the needed dependencies it is possible to generate a working Spring Boot[3] application. Spring Boot is a framework which allows it to mainly use annotations to define REST APIs and bind them to actual methods in the code. Furthermore the Spring framework allows to add interceptors which we use to monitor incoming and outgoing requests. This aspect is covered in the following section.

3.3.2. Monitoring

To monitor the whole microservice environment, the generated microservices are instrumented with Kieker in multiple ways. The first way the microservices are instrumented is the application inside. This is achieved by using Kieker with AspectJ instrumentation. That allows us to track method calls.

In order to be able to monitor the communication between the microservices we adapted the Spring and Jersey interceptors in the Kieker repository[4] to our setup. They intercept incoming and outgoing requests and generate the corresponding records for us to be able to track the communication.

[2] https://maven.apache.org/
[3] http://projects.spring.io/spring-boot/
[4] https://github.com/kieker-monitoring/kieker

3.3.3. Dockerfiles

Even though a microservice can be run and instrumented, we need to put them into containers to efficiently use them in a Kubernetes cluster. Therefor, the generated Docker files assume the microservices to be packed in Java Archive (JAR) files, which can be easily imported into a Docker container and be executed when deployed.

```
FROM java:openjdk-8
EXPOSE 8080
COPY target/portal-1.0.0.jar /
CMD java -jar portal-1.0.0.jar
```

Example of a Dockerfile for a generated microservice (`Dockerfile`)

The example file contains the definition of the base image for the Docker container defined by the keyword **FROM**. As the generated microservices are Java applications, we use the `java:openjdk-8` base image. To be reachable from the outside, it is necessary to define the port that should be exposed, which is defined by **EXPOSE** and set to port 8080 which is the default for the underlying web server. In the next step, the JAR file which contains the microservice application is copied to the root directory of the container with the keyword **COPY**. Eventually the command (keyword **CMD**)that should be executed when the container is started is defined, which in this case is just executing the copied JAR file.

3.3.4. Kubernetes

To run microservices in a Kubernetes cluster, two configuration files in the YAML serialization language are required. One defining the replication controller settings. It contains information about the name, the underlying Docker image and the initial number of replicas that should be started of this image. The second file contains the settings for the service. It mainly defines which ports of this container should be available to the outside.

```
apiVersion: v1
kind: Service
metadata:
  name: portal
  labels:
    name: portal
spec:
  type: NodePort
  ports:
  - port: 8080
```

```
    name: http
  selector:
    name: portal
```

Kubernetes service file example (`service.yaml`)

The service file contains basic information about a service. Apart from the definition of the API version, the kind of configuration the file contains and metadata, it specifies the ports that should be available to the outside. Every port is defined by the port number and a name.

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: portal
  labels:
    name: portal
spec:
  replicas: 1
  selector:
    name: portal
  template:
    metadata:
      labels:
        name: portal
    spec:
      containers:
      - name: portal
        image: my/portal
        imagePullPolicy: IfNotPresent
        ports:
        - containerPort: 8080
```

Kubernetes controller file example (`controller.yaml`)

Similar to the service file a controller file contains some basic information but specifically defines the number of replicas that should be initially started of this service and which Docker image should be used. Furthermore, the policy when the Docker image should be pulled from the repository is defined. In this case, the Docker image is only pulled from the repository if it is not locally available.

3.3.5. Scripts

Throughout the work on this thesis, several scripts have been written that make it easier to generate Docker images from the generated files and deploy them on a local virtual

machine running minikube or on a remote Kubernetes cluster. These scripts will also be provided in the supplemental material of this thesis.

3.4. Model Extraction

The model itself could likely be extracted from extracted microservice environments if the required data can be deducted. Due to the time limitations of this thesis and the fact, that this aspect is not a necessity for the approach in general, it was not implemented yet.

Chapter 4

Change-aware Anomaly Detection

This chapter elaborates on the approach that was developed and evaluated.

4.1. Overview

After having motivated the topic and challenges, this section states the goals that are intended to be achieved throughout this thesis.

4.1.1. Design of a Metamodel for Microservice Environments

In order to be able to systematically run experiments, a model that represents microservice environments including structural information like physical distribution and dependencies was created. As it is helpful to also be able to use such a model in the context of applications and algorithms, this was done using Eclipse Ecore.

4.1.2. Implementation of a Generator for Microservice Environments Based on the Specified Metamodel

The said model will be used to generate actual microservices that resemble the modeled structures in terms of dependencies between multiple microservices. This generator will also be capable of generating the needed wrappers to be able to deploy the services as Docker containers or to a Kubernetes cluster. As these service will be used for evaluation, they will be instrumented with Kieker.

4.1.3. Simulation Environment for Evaluation Purposes

In addition to the microservices derived from the model some utility microservices that complete the simulation environment were built. These provide centralized logging of measurements and events and manipulation of the generated services behavior in terms of response times.

4.1.4. Identification and Implementation of Approaches to Reduce Anomaly Alerts due to Version Changes

Based on the available data, approaches to reduce anomaly alerts due to version changes will be identified and implemented.

4.1.5. Evaluation

In the evaluation, the implemented algorithms will be run on the data from the simulation environment and will be compared with other existing algorithms.

4.2. Types of Anomalies

In the context of continuously changing services the following kinds of anomalies were identified. Some of them are to be expected when service instances are deleted and replaced with newer version. However, some of the anomalies represent unexpected behavior and should trigger alerts.

4.2.1. Anomalies Caused by Initialization

As services are not changed in place, but are replaced, the application initialization phase can behave different than the normal behavior. Response time behavior as depicted in Figure 4.1 could, for example, be caused by class loading or instantiation of objects. Due to such behavior changes, a classical anomaly detection tool could classify this phase as an anomaly.

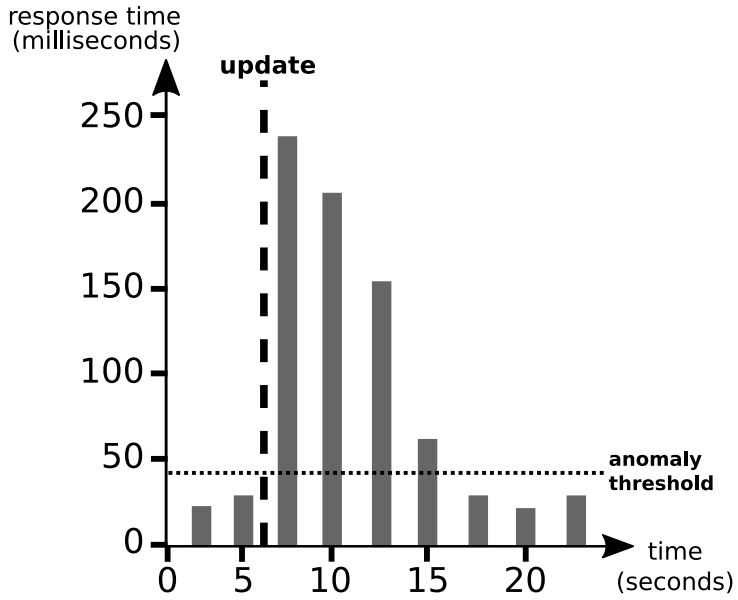

Figure 4.1.: Visualization of an initialization anomaly

4.2.2. Anomalies Caused by Code Changes

When services are changed, also their behavior may change to some extent. This leads to changes in the performance measurements that manifest themselves in a change of behavior in comparison to the previous version, but will be steady in the current version. Figure 4.2 shows the response time of a service instance with version 1 that will be replaced by a newer version of this service. Before the service with version 1 is shut down, an instance with the newer version is started. When the version 2 instance is running, the older service instance is stopped. Due to changes in the code of the service, the response time increases. If this would be observed by a classical anomaly detection tool, it would likely be classified as anomalous behavior.

4.2.3. Real Anomalies

This kind of anomalies are still to be detected and alarms should be raised in case they occur. Figure 4.3 depicts a anomaly that is not related to changes in a microservice environment. Even though the kinds of anomalies that were mentioned in the previous sections should not be treated as anomalies right from the beginning, there should be mechanisms in place that ensure that misbehaving initializations or code changes that lead to bad performance are treated as real anomalies.

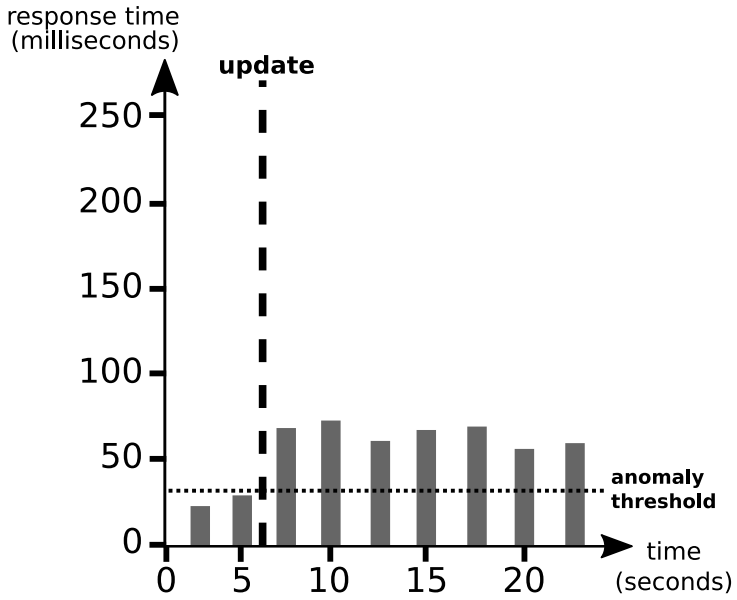

Figure 4.2.: Visualization of a anomaly caused by code changes

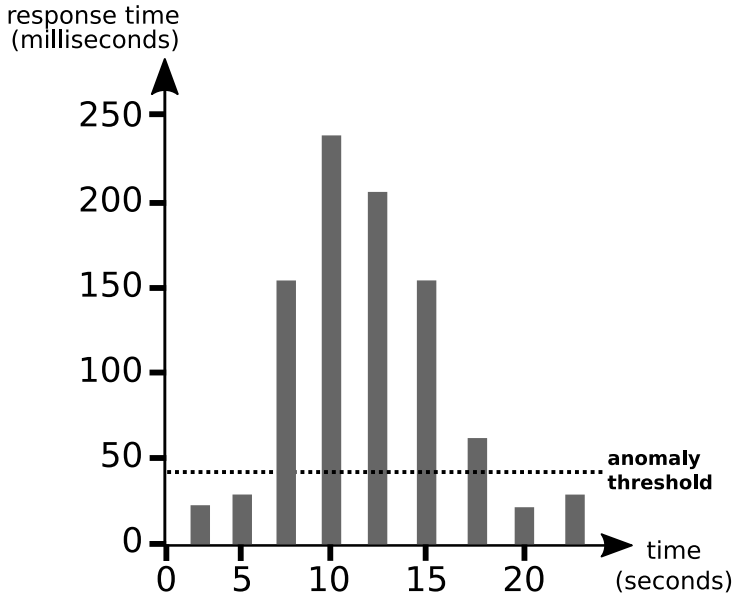

Figure 4.3.: Visualization of a real anomaly

4.3. Assumptions Regarding Changes

In microservice environments it is unusual to modify running microservices. The common practice is to start the modified microservices and stop the old ones afterwards. As the tracking of the information which microservices instances replace which old microservices is a rather complex task and not in the scope of this thesis, we change the

behavior of the existing instances to keep track of the dependencies. By assuming this, we simulate the continuity of the monitoring data which otherwise would have to be collected and put into context.

4.4. Multi-tier Anomaly Evaluation

In order to get a possibly complete picture of the whole environment from different points of view, different views on the application environment will be combined, which consist of the anomaly detection itself covered by ΘPAD, the usage of architectural information to remove anomalies that are induced by transitive calls to other microservices and a new proposed approach that uses event information to reduce anomaly alerts that result from the intended change of the environment (e.g., updates or restarts of microservices).

4.4.1. Anomaly Detection

The aim is to reduce the false positives of anomaly detection in fast-changing microservice environments. To achieve this, the possibly error-prone anomaly alerts are used as the first input to this approach. The anomaly detection features of ΘPAD are used for this task.

Figure 4.4 depicts the measurements of two instances of different services A and B. As one can see, both show a certain peak and a deviation from the original baseline afterwards. Due to these measurements, the peak and the deviated baseline are classified as anomalies for both instances.

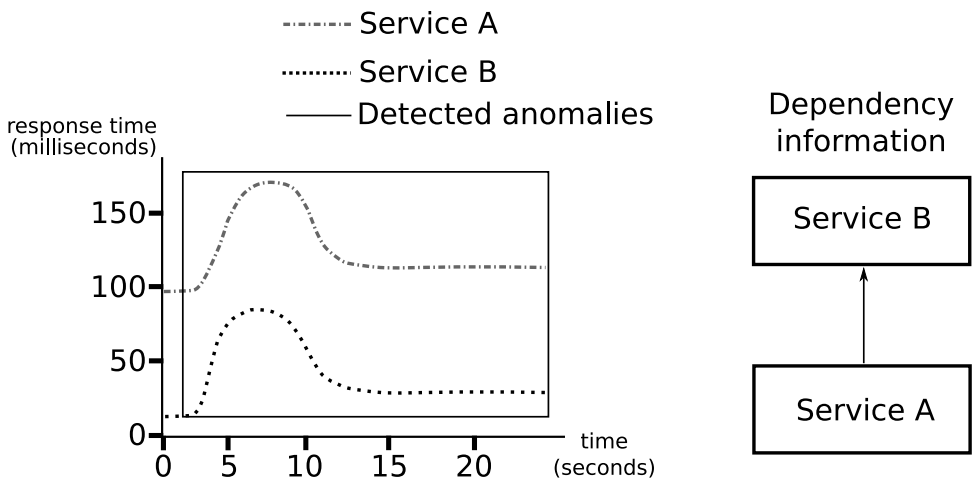

Figure 4.4.: Microservice anomaly detection results

4.4.2. Architectural Information

As microservices heavily depend on other microservices to fulfill their tasks, the information about dependencies between microservice is vital to be taken into account to eliminate false positives that are propagated throughout the dependency tree. Architectural information is the second input for the proposed approach.

Architectural knowledge helps to distinguish transitively propagated problems from real problems. In Figure 4.5 it can be seen that service A calls service B and therefor one can deduct that the source of the peak and the increase in the measurement values afterwards is service B. Applying this to the anomaly detection scenario we can eliminate the deviations from the measurements of service A and reduce the anomaly alert to the service that actually caused the problem.

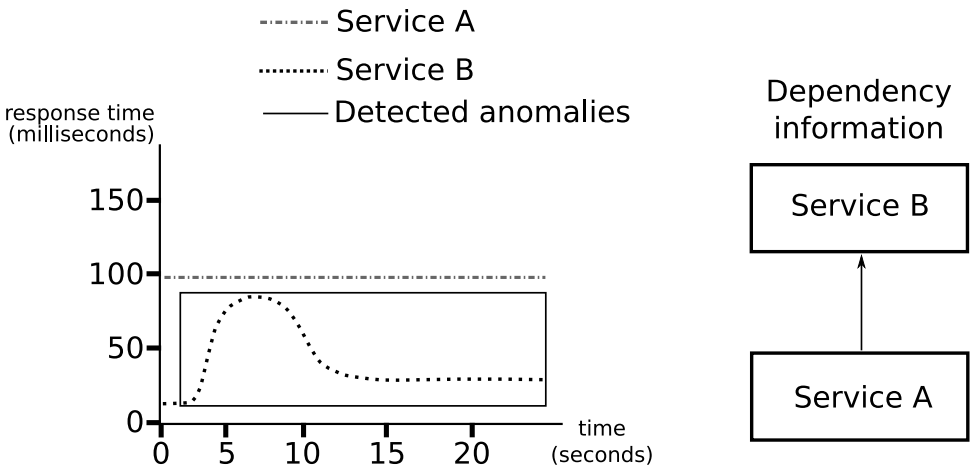

Figure 4.5.: Microservice anomaly detection results after including architectural information

4.4.3. Change Events

Based on the remaining anomaly alerts, system events are taken into account to be informed about changes to the services. Because services might either behave intentionally different because of changes in the code or could show known patterns caused by redeployment (e.g., initialization of the application), the information about changes in the system are vital to reduce the number of false positives. Information about update events is the last input for the approach presented in this thesis.

Figure 4.6 shows the addition of update events into the scenario. Using this knowledge it may be deducted that the peak right after the update might be caused by the initialization

of the service and could be ignored. Applying this to the anomaly detection would further reduce the area of reported anomalies to the baseline deviation some time after the update event.

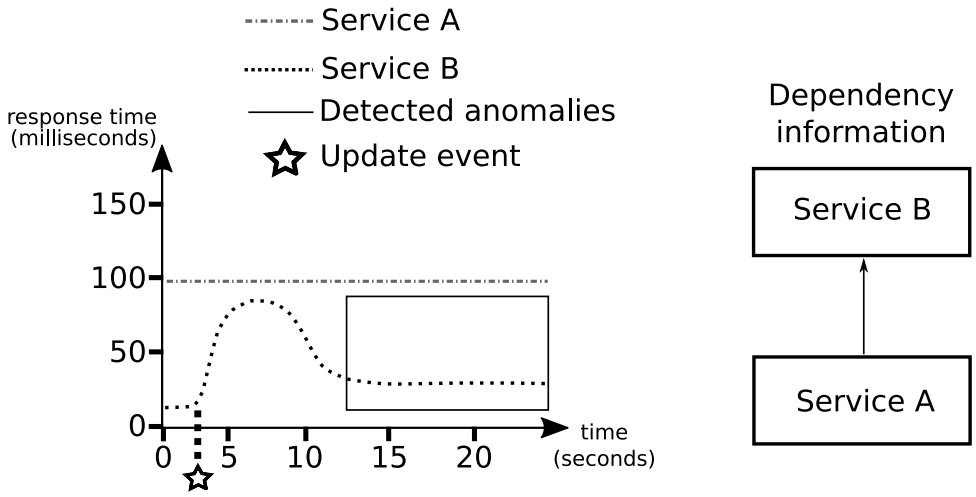

Figure 4.6.: Microservice anomaly detection results after including architectural and event information

4.5. Event-aware Anomaly Revisor (EAR): Incorporating the Acquired Data to Reduce False Positives

The previously presented detection and information gathering approaches provide a reduced number of anomaly alerts already. Based on the information and the alerts, the following approaches are proposed to use this information to further reduce the number of false positive anomaly alerts.

4.5.1. Basic: Temporarily Increase Anomaly Threshold

The idea of the basic approach is to increase the anomaly threshold for a limited time after a change event was recorded. This allows the initialization phase to have a higher anomaly score without being treated as an anomaly.

This approach has three variables: anomaly threshold, grace period duration and grace period factor. The anomaly threshold represents the anomaly threshold during normal operation which would deem a measurement to be anomaly if surpassed. The time of the increased anomaly threshold is called grace period. Grace period duration defines

the duration of the grace period while grace period factor stands for the factor that is applied to the normal threshold during the grace period is in place. So the anomaly threshold during grace period th_g is calculated based on the grace period factor f_g and the regular anomaly threshold th_r like the following:

(4.1) $th_g = th_r \cdot f_g$

As soon as the grace period ends, the anomaly threshold is reset to the original value (t_r).

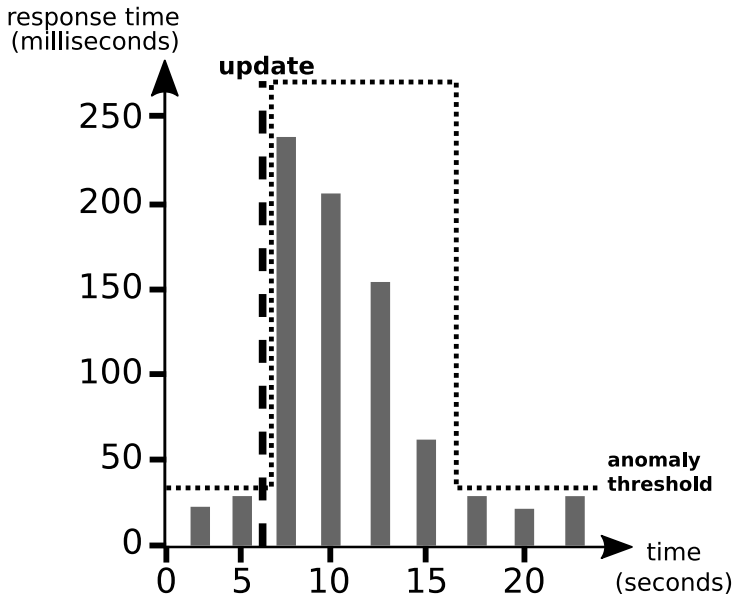

Figure 4.7.: Basic approach using a factor applied to the anomaly score

Figure 4.7 depicts the event and the following application of the grace period.

4.5.2. Advanced: Temporarily Increase and Gradually Reduce Anomaly Threshold

The advanced approach assumes that the anomalous behavior due to changes during the initialization phase is degrading over time until it returns to normal behavior. In contrast to the basic approach it raises the anomaly threshold in the beginning of the grace period and reduces it over time until it has returned to the original anomaly threshold.

This approach has the same variables as the basic approach but uses a different formula to calculate the anomaly threshold during the grace period. The duration of the grace period is defined as d_g. It also uses the following dynamic values:

Timestamp of event occurrence (t_{event})
The time when the update event that triggered the grace period was executed.

Timestamp of the current measurement to evaluate ($t_{current}$)
To determine how high the anomaly threshold for a specific measurement is, it is required to know its timestamp to deduct the gradually reduction of the anomaly threshold.

$$th_g(t_{current}) = \left(1 - \frac{t_{current} - t_{event}}{d_g}\right) \cdot f_g \cdot th_r \tag{4.2}$$

Equation 4.2 calculates the anomaly threshold during the grace period th_g depending on the timestamp $t_{current}$ of the anomaly score of the currently processed item. To do so, it calculates the distance of $t_{current}$ to the time of occurrence of the triggering event t_{event} and divides it by the total duration of the grace period d_g. That value is subtracted from 1 as it should be gradually reduced the more the current timestamp reaches the end of the grace period ($t_{current} + d_g$). Eventually the factor f_g and the regular anomaly threshold th_r are multiplied to the previously obtained value.

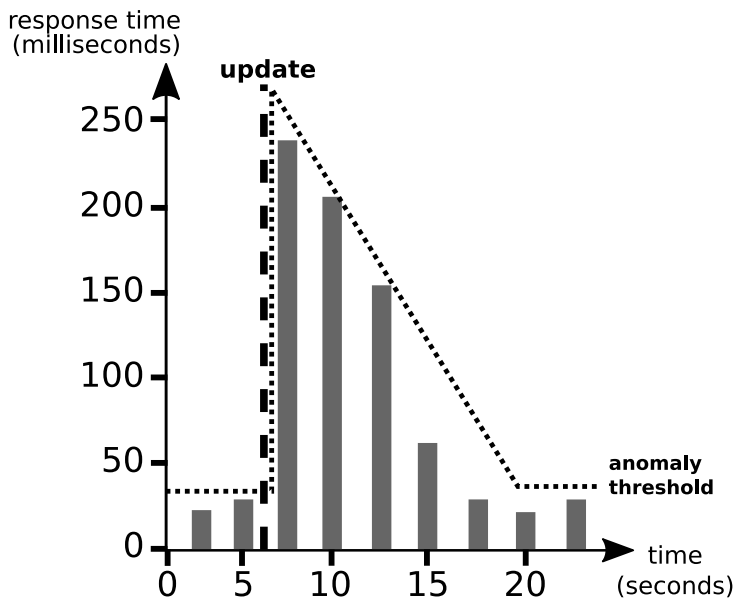

Figure 4.8.: Advanced approach using a gradually reducing anomaly score after an event was detected

Figure 4.8 depicts the event and the following gradually reducing threshold throughout the grace period.

4.6. Implementation

This section describes the important steps of the post processing which is applied to the monitoring data after its extraction from the monitored environment. The items that are prefixed with “Kieker” are existing filters inside the Kieker monitoring framework.

4.6.1. Reconstruct Traces (Kieker: *TraceReconstructionFilter*)

After having read the monitoring data from the file system, the Kieker filter *ExecutionRecordTransofrmationFilter* is used to transform the monitored data in the form of *OperationExecutionRecords* to *Execution* objects. These data structures hold information about the signature of the operation that was called, when it was called and when it returned, on which host name it was executed, a call depth indicator and a sequence counter. This step is only a conversion step which does not alter the underlying data.

4.6.2. Extract Dependencies from Traces (*MessageTraceDependencyExtractor*)

In this step the obtained traces are used to extract the trace information in objects that are used to associate the actual monitored records with the complete execution path it was monitored in. As we assume to have monitoring data from a microservice environment, one may have multiple instances of a microservice. That means, that we have to consider the possibility that single instances may have performance problems. Therefor, we have to take the exact path into account to not mix up anomalies of other instances in the resulting data.

4.6.3. Extract Required Data from Records (Kieker: *ExtractionFilter*)

After having extracted the dependencies and associated them with the records, some parts of the records are not needed anymore. Therefor, all data except the identifying name, which also contains a reference to the stored trace information, the timestamp and the response time is removed.

4.6.4. Aggregation (Kieker: *TimeSeriesPointAggregatorFilter*)

The monitored data is aggregated in order to have equidistant data points. This kind of data points is required for the subsequent steps.

4.6.5. Forecasting (Kieker: *ForecastingFilter*)

The forecasting step uses the previous data points to predict the next data point. This implementation uses the mean forecaster which uses the Java arithmetic mean implementation to deduct the next value. This method was chosen as most of the other approaches assume some kind of seasonality which is not the case here.

4.6.6. Anomaly Score Calculation (Kieker: *AnomalyScoreCalculationFilter*)

Based on the forecasted value from the previous step and the actual measured value, an anomaly score is calculated. This represents the anomaly score output of the ΘPAD approach.

4.6.7. Anomaly Detection (Kieker: *AnomalyDetectionFilter*)

Even though this Kieker filter was initially used as a way to extract only anomalous or non-anomalous data points, it is used here to add the anomaly threshold to the record.

4.6.8. Remapping of ΘPAD Anomaly Scores

To be able to compare the anomaly scores of ΘPAD, RanCorr and our approach, we need to remap the anomaly scores of ΘPAD. ΘPAD uses an anomaly interval of $[0, 1]$ with 0 for no anomaly to 1 for anomaly. RanCorr internally uses an interval of $[-1, 1]$ and is remapped to the ΘPAD interval. To have comparable results, we mapped the ΘPAD anomaly scores to the RanCorr interval and back.

4.6.9. RanCorr

The RanCorr implementation of Düllmann et al. [DEE+14] was used as a reference and was customized as the structural assumptions were partly different. In the existing implementation the multiple hierarchy levels (e.g., host, package, class, operation) were distinguished while the aim in this thesis was to use only the dependency levels between microservice operations. Therefor the RanCorr approach was adapted and partly simplified to fit the requirements in terms of the input data we have.

Algorithmus 4.1 Customized RanCorr Algorithm

```
1:  procedure CUSTOMIZEDRANCORR(currentItem)
2:      currentAnomalyScore ← currentItem.getAnomalyScore()
3:      forwardAnomalyScore ← CustomizedRanCorr(currentItem.getNext())
4:
5:      if forwardAnomalyScore >= currentAnomalyScore then
6:          return 0.5 * (currentAnomalyScore - 1)
7:      else
8:          return max(localAnomalyScore, forwardAnomalyScore)
9:      end if
10: end procedure
```

Algorithm 4.1 describes the simplified RanCorr algorithm in pseudo code which recursively obtains the anomaly scores of the items it depends on, and takes into account if the upstream anomaly scores are higher than the local ones.

4.6.10. EAR

The EAR takes the monitoring data associated with timestamps and the events that were recorded into account. As it is known where updates were conducted, it is possible to apply the effects of the algorithm only to the records that are affected by the events.

Algorithm 4.2 illustrates the basic idea of the basic EAR algorithm. The inputs are the events (e.g., update events) containing the time and the scope of the change (e.g., which service type was affected), the anomaly records which contain the scope of the anomaly record and its timestamp. It adds the default anomaly threshold for all anomaly records by default and applies the increased anomaly score to those anomaly records that are affected by the events. The variable *gracePeriod* defines the duration of the allowed increased anomaly scores and the *anomalyFactor* stands for the factor that is applied to the normal threshold during the grace period. *Note:* Due to time limitations it was not possible to implement the advanced EAR approach.

Algorithmus 4.2 EAAR Basic Algorithm

```
procedure EAARBasic(events, anomalyRecords, threshold)
    gracePeriod ← 15
    anomalyFactor ← 2.0
    anomalyRecords.setThreshold(threshold)

    for all event in events do
        affectedAnomalyRecords ← anomalyRecords.getAffected(event, gracePeriod)
        affectedAnomalyRecords.setThreshold(threshold * anomalyFactor)
    end for
end procedure
```

Chapter 5

Evaluation

5.1. Evaluation Goals

See whether the combined approach performs better than the naive approach. Compare the results with a commercial product (if possible)

Find out whether the approach also performs for big microservice environments with many service instances.

5.1.1. Research Questions

1. Can the anomaly detection quality be improved by incorporating architectural knowledge?

   $H0_{arch}$: Anomaly detection quality does not change or is reduced by incorporating architectural knowledge.

   $H1_{arch}$: Anomaly detection quality is increased by incorporating architectural knowledge.

2. Can the anomaly detection quality be improved by incorporating event knowledge?

   $H0_{event}$: Anomaly detection quality does not change or is reduced by incorporating event knowledge.

   $H1_{event}$: Anomaly detection quality is increased by incorporating event knowledge.

To make sure that the customization of the RanCorr implementation was an improvement, it is necessary to check whether it improved the anomaly detection quality in contrast to the ΘPAD approach. Eventually, the EAR approach is compared with the RanCorr approach as it uses the results of RanCorr to again improve the anomaly detection quality.

5.2. Evaluation Methodology

The generated SUT is executed in a lab experiment on the minikube Kubernetes environment together with supplemental microservice required to monitor the system and provide the load for the microservices in the form of HTTP requests. During the experiment phase anomalies get injected that lead to a delay during the execution of the operations on the SUT resulting in increased response times. The injections can have different scopes which either target all operations of one microservice type or even specific operations of a microservice type. In the beginning of the experiment, the microservice operations get assigned initial delays which are intended to resemble usual response times and are not treated as actual injections. Once the initialization phase of the experiment is finished, the load generation is executed. This leads to the actual monitoring measurements and first gives an impression on the initial state of the SUT. Afterwards, the anomaly injections are made in a step by step manner. Every injection is labeled with an injection type of “real” and “init”. The “real” injection type stands for real anomalies that still should be detected and are not expected. The “init” injection stands for anomalies caused by the initialization phase of a microservice. These events are expected and their starting time is logged.

After the experiment is finished, the monitoring data is extracted and the event and injection logs are stored. The following steps are conducted off-line and the experiment setup is no longer needed at this point. Next, the monitoring data and the event logs are inserted in the evaluation application which processes the data with ΘPAD, RanCorr and the approach of this thesis, the EAR approach to eventually compare their performance in terms of detection quality. Another application applies the data about the injected anomalies labeled with “real” to calculate the quality metrics (TP,FN,TN,FP). Based on this data, we compare the quality of the approaches to see whether the customized RanCorr approach and the EAR approach can improve the detection quality.

5.3. Evaluation Setup

The SUT was generated by using the generator described in Section 3.3.

The SUT is a three tier microservice environment with one instance of each tier. Figure 5.1 shows the dependencies between the operations of the tiers.

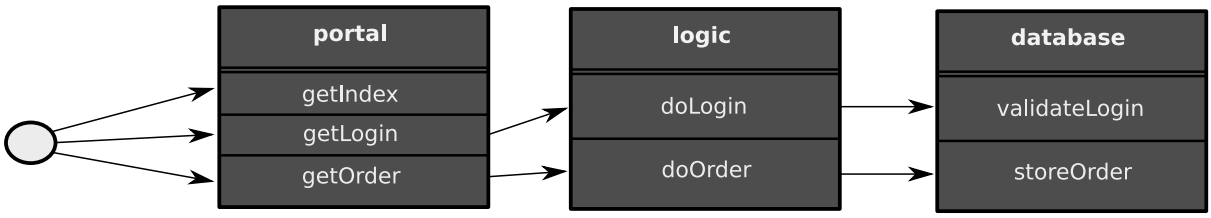

Figure 5.1.: SUT

The portal microservice has three operations where two of them depend on operations of the logic microservice. Again, the logic microservice has dependencies to methods of the database microservice.

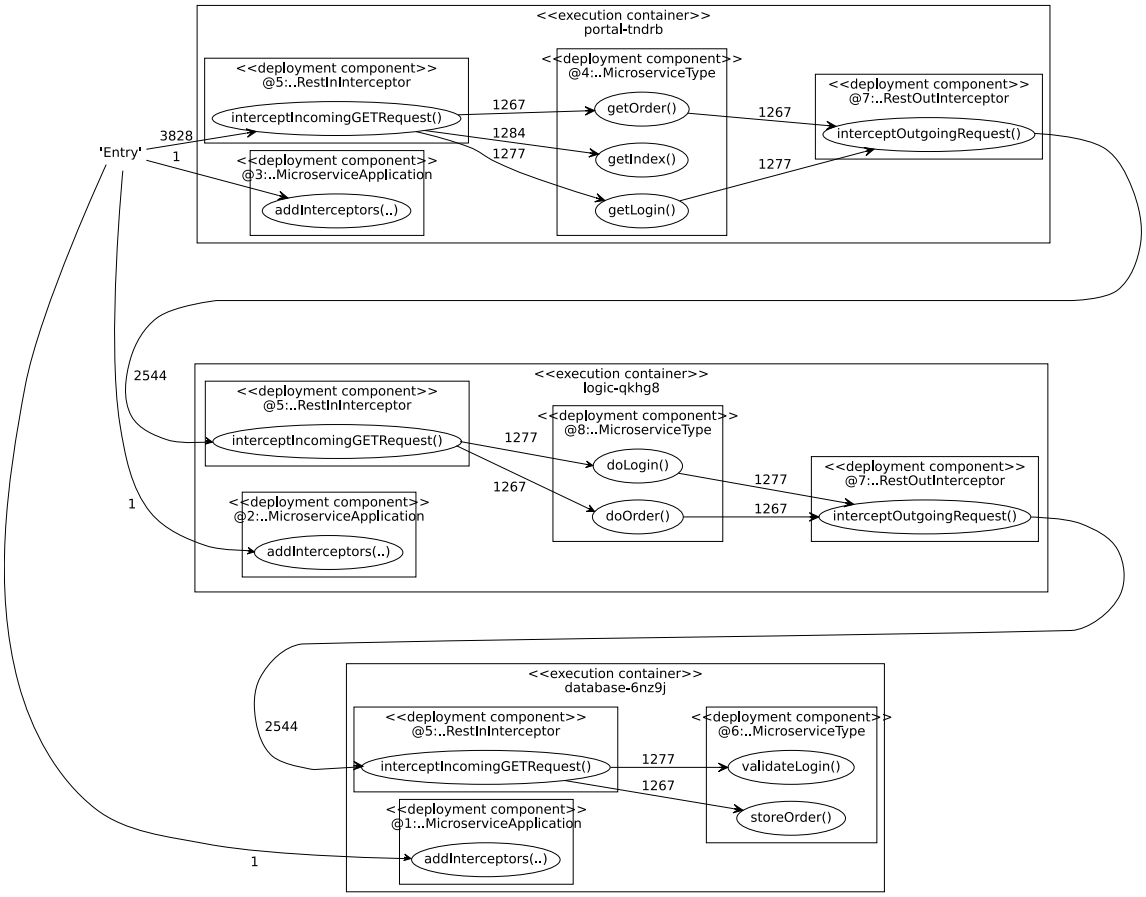

Figure 5.2.: SUT structure generated from the monitoring data

Based on the monitoring data obtained during the experiment, the following diagram was extracted. It shows the call structure and number of requests sent to the operations of the microservices. In addition to the operations shown in 5.1 the interception methods are visible here which are required to be able to track the complete call hierarchy throughout the whole system.

5.3.1. Supplemental Setup

The said setups are put into an evaluation environment which contains several additional microservices which are required for the conduction of the experiment.

Figure 5.3 depicts the supplemental setup which will be described in this section.

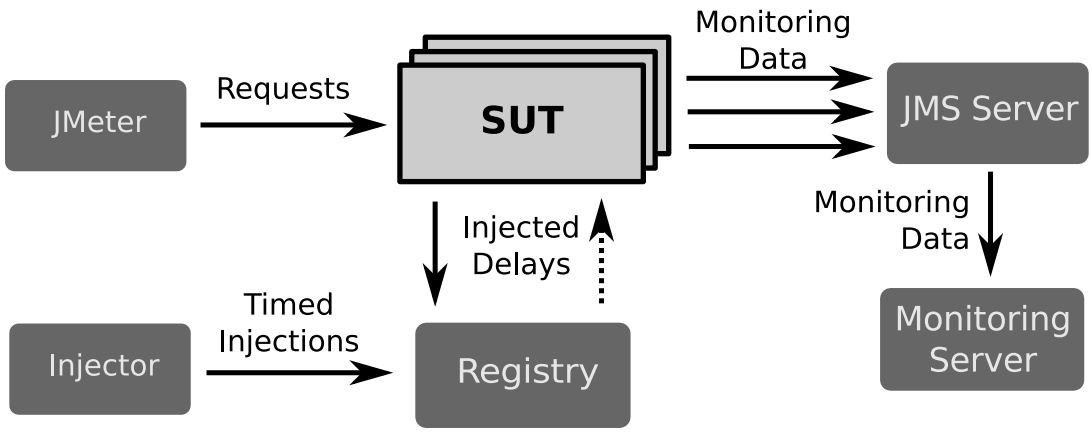

Figure 5.3.: Evaluation Setup for the generated SUT

JMeter
The JMeter microservice provides the load testing that is directed to the SUT. It contains a test plan that is automatically executed after a short delay.

JMS Server
The JMS Server is responsible for collecting the monitoring data from all microservices in the SUT. During the SUT generation process all microservices are instrumented and set up to send their monitoring data to the JMS Server. The JMS Server runs an ActiveMQ[1] server in version 5.14.1 which is configured to create an initial queue for the said monitoring data. This way there is one point where the monitoring data can be read from.

[1] http://activemq.apache.org/

Monitoring Server
The monitoring server uses the Kieker filters to connect to the JMS Server and reads the monitoring data to write them to the local file system. After the experiment is finished, the monitoring data can be copied from it. One way to do so is to use the installed OpenSSH[2] server which provides the possibility to connect to it from a remote system and copy files with the secure copy tool (SCP) which is part of the OpenSSH suite.

Registry and Injector
The registry is responsible for keeping track of the injections. The microservices of the SUT send requests to the registry with information about their type, operation and unique ID. The registry component looks up whether delays are defined for these properties and returns the delay value. That way it is possible to have a central component managing all the injections. If injections with a defined duration are set up, it removes such injections after the duration has expired.

The injector component is placed inside the registry microservice where it executes the experiment plan where the injections are defined. Based on the plan it sends injection requests to the registry microservice which then stores the injections to provide them to the microservices in the SUT.

5.4. Experiment Settings

In the post-processing environment it is possible to set several variables that may influence the outcome of the results.

Maximum Trace Duration (5 seconds)
The maximum trace duration is set in the Kieker filter that extracts the traces from the monitoring data. This means that traces that take longer than the given number of seconds are discarded. In this case the maximum trace duration was set to five seconds. As there are no traces that take longer than this time, it did not lead to any discarded traces.

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

Aggregation Span (5 seconds)
The aggregation span is set in the Kieker filter that aggregates the monitoring data into an equidistant time series. The aggregation span defines the window size for which one representing value is calculated and was set to five seconds That means that out of the monitoring data of every five seconds one representing value is calculated.

Forecasting Method (Java Mean)
The forecasting method was set in the Kieker filter that forecasts the next value based on the historical data. The method was set to Java mean as we do not have seasonal data that would be relevant to use models to predict the values. Apart from that, the Java mean method has a fast execution time.

Event Grace Period (15 seconds)
The event grace period is set in the EAR algorithm to define for how long the grace period should be applied after an event was detected. In the experiment this value was set to 15 seconds.

Event Grace Period Factor (2.0)
The event grace period factor is set in the EAR algorithm to define the factor for the normal threshold during the grace period. In the experiment this value was set to 2.

Anomaly Threshold (variable)
The anomaly threshold is set in ΘPAD, RanCorr and EAR to be added to the records for later evaluation. This value was varied step-wise from 0.05 to 0.95 in steps of 0.05.

5.4.1. Injections

Table 5.1 shows the injections that were executed throughout the experiment. The time is in seconds after the start of the experiment. For all injection types that are not of the type “real” the timestamps of the event occurrence were logged and used as input for the post-processing. As the injections of the type “real” resemble real anomalies, which should be still detected as anomalies while the other types should not. The scope defines which microservices and which operations are affected by this injection. The values in braces in the offset column represent the delay that persists after the end of the duration. This should resemble code changes that affect the performance of the application in the long term.

Time (s)	Type	Scope	Offset (ms)	Duration (s)
0	Start of load generation			
20	real	portal:getIndex	250	30
80	init	logic	200	15
110	init	database	350 (100)	20
160	real	logic	300	30
200	init	portal	330 (30)	15
240	init	database	150	20
280	real	portal:getOrder	250	30
380	init	logic	250 (50)	20
420	real	database	250	30
520	End of the experiment			

Table 5.1.: Injections in the experiment

5.4.2. Load Driver

The requests directed to the SUT were generated by Apache JMeter[3] which is an open source tool for load testing.

The test plan for the experiment was set to a total duration of 500 seconds, using 25 threads which loop forever until the test duration ends. The number of threads was ramped up to the maximum value over 30 seconds (ramp-up period).

Every thread sent HTTP requests in the given order:

- GET http://portal:8080/getIndex
- GET http://portal:8080/getLogin
- GET http://portal:8080/getOrder

5.4.3. Evaluation environment

The experiment was conducted in an minikube (v0.14.0) environment which got assigned 4 CPUs and 8196MB of RAM. The minikube tool uses VirtualBox to run the virtual machine. The installed VirtualBox version was Oracle VirtualBox in version 5.1.12.

[3] https://jmeter.apache.org/

The host system the minikube virtual machine was run in was an Arch Linux system with a quad-core CPU (Intel(R) Core(TM) i7-3630QM CPU @ 2.40GHz) with 16GB RAM in total.

5.5. Description of Results

Based on the F-measure of the datasets of the different thresholds, the one with the highest F-measure is presented here which is the anomaly threshold of 0.1. The dataset containing all values for all measured thresholds is publicly available at [Dül17]. The total number of data points is 718 and the values are rounded to 3 positions after the decimal point.

Approach	TP	TN	FP	FN	TPR	FPR	PPV	F-Measure	Accuracy
ΘPAD (1)	137	218	218	145	0.486	0.5	0.386	0.43	0.494
RanCorr (2)	98	248	187	184	0.348	0.429	0.344	0.346	0.397
EAR (3)	95	257	179	187	0.337	0.411	0.347	0.342	0.411

Table 5.2.: Statistics for the different approaches for an anomaly threshold of 0.1

In Table 5.2 the different approaches and the corresponding statistical metrics for an anomaly threshold of 0.1 are compared. In comparison, the ΘPAD results are the best across all metrics but the false positive rate.

Compared	TP	TN	FP	FN	TPR	FPR	PPV	F-Measure	Accuracy
1/2	-39	+30	-31	+39	-0.138	-0.071	-0.042	-0.084	-0.097
2/3	-3	+9	-8	+3	-0.011	-0.018	+0.003	-0.004	+0.014

Table 5.3.: Comparison of the different approaches for an anomaly threshold of 0.1 (1: ΘPAD, 2: Customized RanCorr, 3: EAR)

Table 5.3 shows the differences between the results of the approaches. 1/2 stands for the comparison between ΘPAD and the customized RanCorr approach and 2/3 represents the comparison between the customized RanCorr and the EAR approach. While the difference between the results of ΘPAD are quite high, the difference between RanCorr and the EAR approach are very small.

5.5.1. Receiver Operation Characteristic (ROC) curves

In order to visualize the results, the results for each of the approaches is plotted using a Receiver Operation Characteristic (ROC) curve which plots the false positive rate on x-axis and the true positive rate on the y-axis. This visualization makes it easier to see how good the classification of the different approaches performs. The diagonal line is the baseline which represents the result a random classification of values would have. Data points lying below that line, perform worse than a random guess. Optimal results should have a high true positive rate and a low false positive rate. That means that the optimal result would be many data points in the upper left corner.

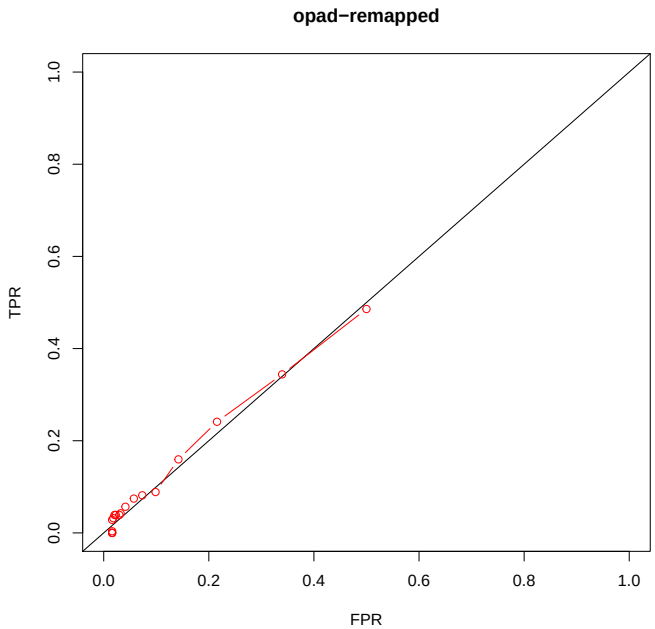

Figure 5.4.: Results of the prediction performance of the ΘPAD approach visualized by a ROC curve

Figure 5.4 shows the ROC curve of the results of the ΘPAD evaluation. Its data points lie close to the indicator line for a random classification and partially even go below it.

Figure 5.5 depicts the ROC curve for the prediction performance of the customized RanCorr approach. It also lies close to the line representing the random guess classification line and crosses it at some point.

In Figure 5.6 the ROC curve for the prediction performance of the EAR approach can be seen. Even though the line lifts off the baseline, it surpasses it at a higher threshold.

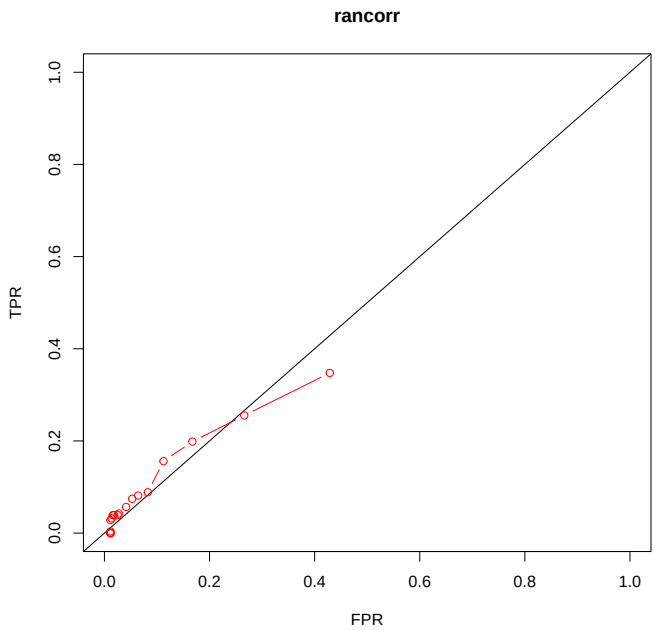

Figure 5.5.: Results of the prediction performance of the customized RanCorr approach visualized by a ROC curve

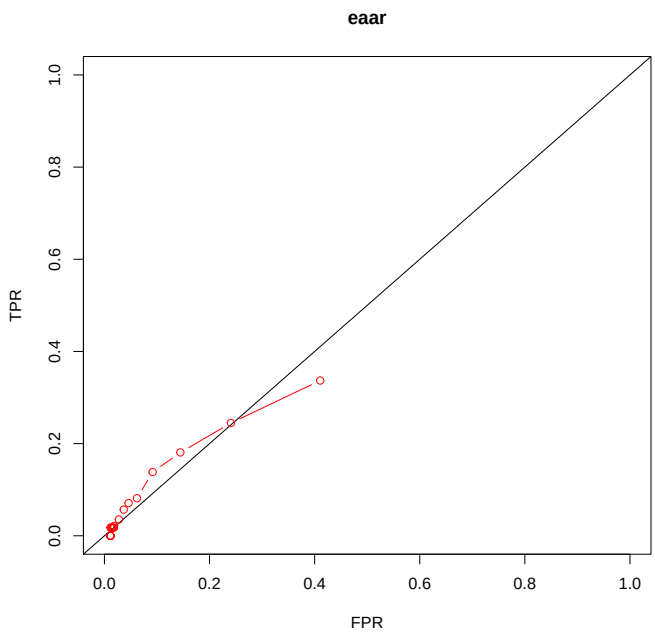

Figure 5.6.: Results of the EAR approach in a ROC curve

5.6. Discussion of Results

Based on the metrics reported in Table 5.2 the general anomaly detection quality is rather bad as all metrics are quite low.

As even the existing ΘPAD approach does not perform very well in this context, there might be some aspects that could be improved to increase the overall anomaly detection quality. This could be the revision of the parameters that can be set in the chain of Kieker ΘPAD that may influence the results to some extent.

Due to the big difference between the ΘPAD approach and the customized RanCorr approach depicted in Table 5.3 we have to assume that the intended positive impact of the customized RanCorr approach did not succeed. Therefor, we have to accept the null hypothesis $H0_{arch}$ and reject $H1_{arch}$ as the customized RanCorr approach did not improve but mainly worsened the overall anomaly prediction performance. Based on the comparison table we can assume that the calculation of the impact of anomalies in components a microservice depends on has to be improved to better take dependencies into account while not reducing the number of correct detections. As the basic calculations of the original approach were adapted, it may be the case that the calculations have to be revised as the original approach was not explicitly designed for distributed systems. The algorithm that was used as a reference was mainly designed to work in systems with multiple layers of granularity (e.g., host level, application level, operation level).

As the EAR approach is based on the anomaly detection scores from the customized RanCorr approach, the performance in contrast to ΘPAD is also rather bad. But, in comparison to the customized RanCorr approach it provides a slight improvement of the detection performance. Only the true positive rate is slightly reduced which also leads to a small reduction of the F-measure. Due to the overall improvement in comparison to the RanCorr approach, we can reject the null hypothesis $H0_{event}$ and can accept the hypothesis $H1_{event}$.

Figure 5.7 which is a selected plot to show the effect of the EAR approach. It shows the anomaly score by ΘPAD in black, the anomaly score by the customized RanCorr in blue with dots and the anomaly threshold set by EAR. One can see that the anomaly threshold increases when events are logged and the anomaly score stays steady on its set value (here: 0.2). In this case the rising anomaly scores are partly covered by the increased anomaly threshold and therefor reduce parts of the false positives for this specific method that would arise if the information about the events would not be present and taken into account. Also, one can see that some of the event-induced anomalies surpass the anomaly threshold.

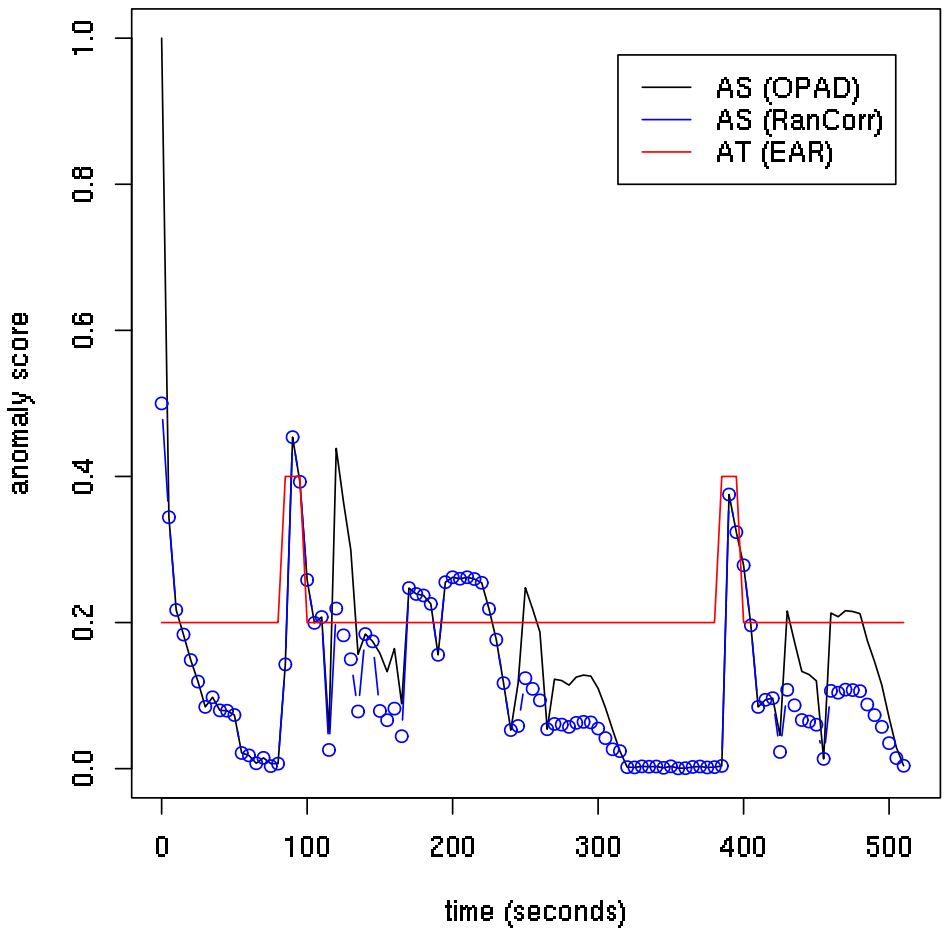

Figure 5.7.: Example for EAR in effect

5.6.1. Threats to Validity

As the evaluation was conducted in a lab experiment, there are several threats to validity that are discussed in the following.

Internal
The duration of 520 seconds of the experiment and the limited number of injections of various types may be not enough to evaluate the different approaches in an exhaustive manner. Furthermore, the SUT had a size of three microservices and seven operations while having four dependencies. Due to this limitations in size, it might not have the

structure of real microservice environments. Eventually, the load in form of HTTP requests had a simple pattern which may not be representative on the load real microservices see. The evaluation environment was intended to represent microservice environment but did not completely do so. The microservices were structured in a simple manner and had no actual workload. This aspect was tried to be simulated by injecting basic delays.

External

In terms of applicability of the results of this thesis on the real world and real microservices, the evaluation could only represent parts of real microservice environments as the environment the microservices were executed in had similar structures, but the overall size and structure of the SUT itself is likely not representative.

Chapter 6

Conclusion

In the conclusion, the thesis is summarized and the overall outcome is discussed. Eventually, possible aspects for future work are presented.

6.1. Summary

This thesis proposes a meta model for microservice environments and means to generate actual microservices from instances of the said model. Together with the generated microservices an evaluation environment was built that can be used to collect monitoring data from them, put load on them and inject delays into the operations of the individual microservices. Additionally, the generated microservices are instrumented with Kieker to monitor the microservices during execution. Furthermore, we provide a basic anomaly detection approach that takes event information into account. Such events could be update events where one knows when new microservices are started and may cause anomaly alerts during the initialization phase. The evaluation of the basic approach in comparison to the customized RanCorr approach could slightly improve the anomaly detection quality. We also designed an advanced approach which could not be implemented or evaluated due to the time limitations of this thesis.

The supplemental material for this thesis publicly available at [Dül17].

6.2. Discussion

Even though the overall results of the evaluation are not very promising, it might be worth to apply the aspect of event awareness to the scope of anomaly detection. It could be possible to improve the overall results by revising the configuration of the ΘPAD

approach and the implementation of the customized RanCorr approach as their input is processed in the EAR approach. Therefor it is hardly possible to improve the results if the input data quality is rather poor. Due to the time limitations of this thesis it was not possible to do so before the due date.

Throughout the work on this thesis, several obstacles came in the way, which also resulted in a limited extent of the approach and the following evaluation. First of all, it was rather difficult to find out how microservices environments are structured in the real world. During this thesis, we tried to collect such information using a questionnaire which failed due to a neglectable amount of responses. Furthermore, several technical tasks took longer than expected. Especially the implementation of the generation artifacts incorporated some level of complexity that lead to a delay of the whole process. As many of the technologies (i.e., Ecore, Xtend, Spring Boot, Spring Interceptors) had to be understood and put into context to create the tooling for the following steps. Later on, the set up of Kubernetes and the interaction with possible distributed cluster setups was rather complex and incorporated big efforts.

6.3. Future Work

In this thesis some aspects came up that could be interesting to pursue and are presented here.

6.3.1. Set up a More Complex Evaluation Environment

As the evaluation setup in this thesis was quite small, it would be interesting to evaluate more complex microservice environments over a longer time with a broader set of changing variables to get more reliable results.

6.3.2. Implementation of the Advanced EAR Approach

In order to further improve the basic approach, the advanced EAR was proposed but could not be implemented or evaluated. Therefor, it would be interesting to implement and actually evaluate it.

6.3.3. Approach to Take Continuous Changes into Account

The previous approaches focused on anomalies that were caused by the initialization process of a microservice instance. One could also think of another approach that tries to cover changing anomaly scores due to software changes. This may be achieved by taking a time frame during the grace period to deduct a new anomaly threshold if it lies in a defined acceptable range.

6.3.4. Model Extraction

In order to use the meta model to its full potential, it would be desirable to also implement the model extraction from the monitoring data. That way, a model instance could be deducted from a running microservice environment.

Appendix A

Ecore Model Diagrams

A.1. Metamodel Structure

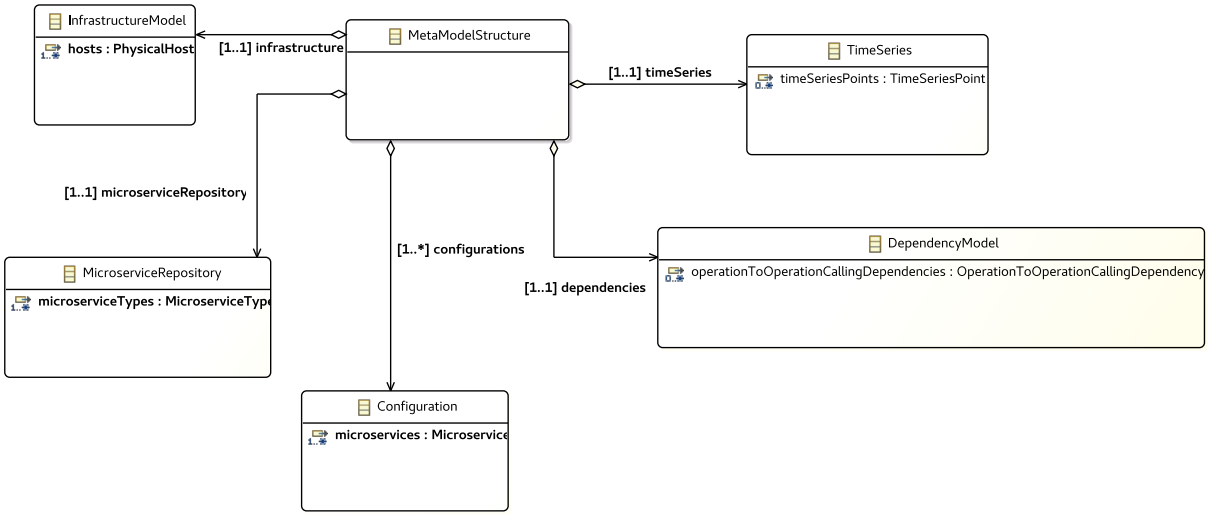

Figure A.1.: Ecore model visualization of the metamodel structure

A.2. Deployment Model

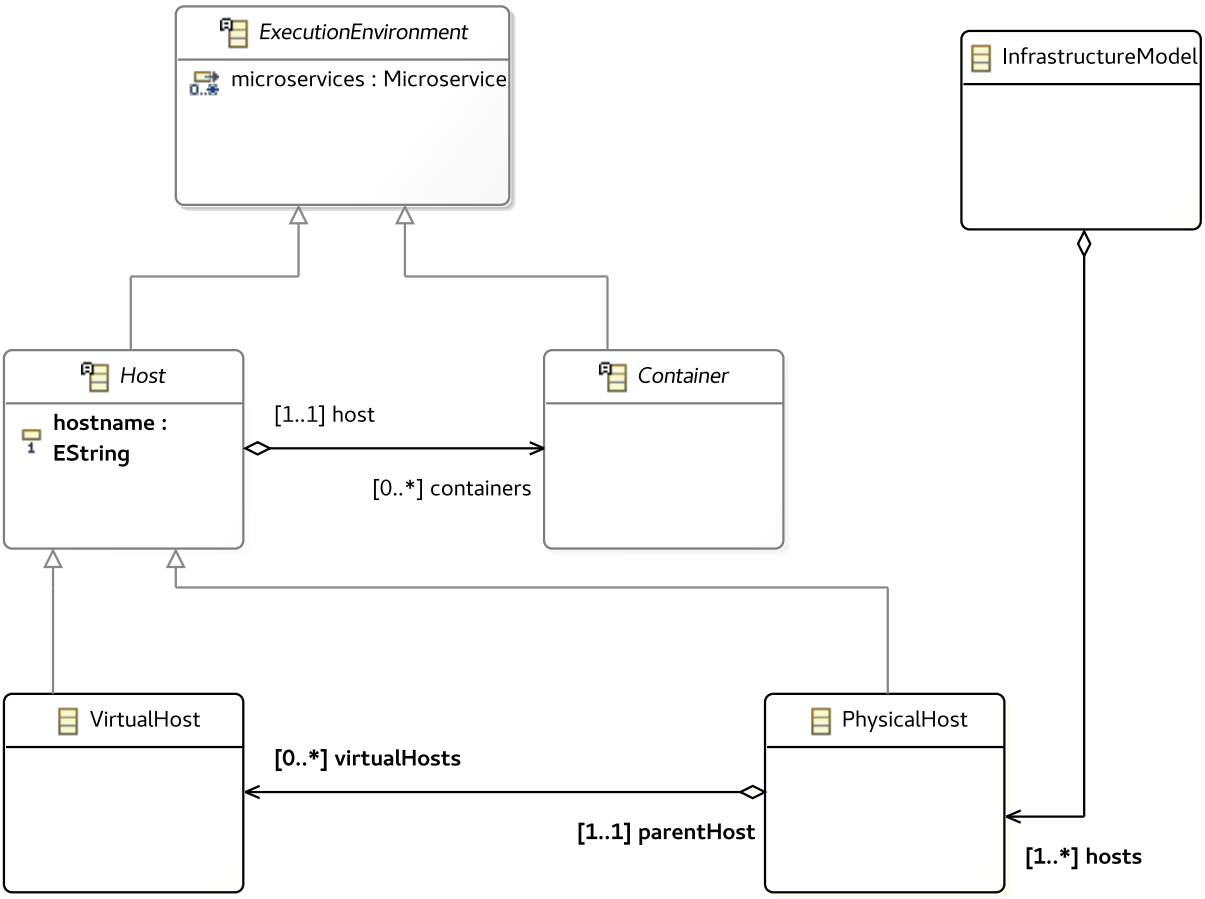

Figure A.2.: Ecore model visualization of the deployment model

A.3. Microservice Model

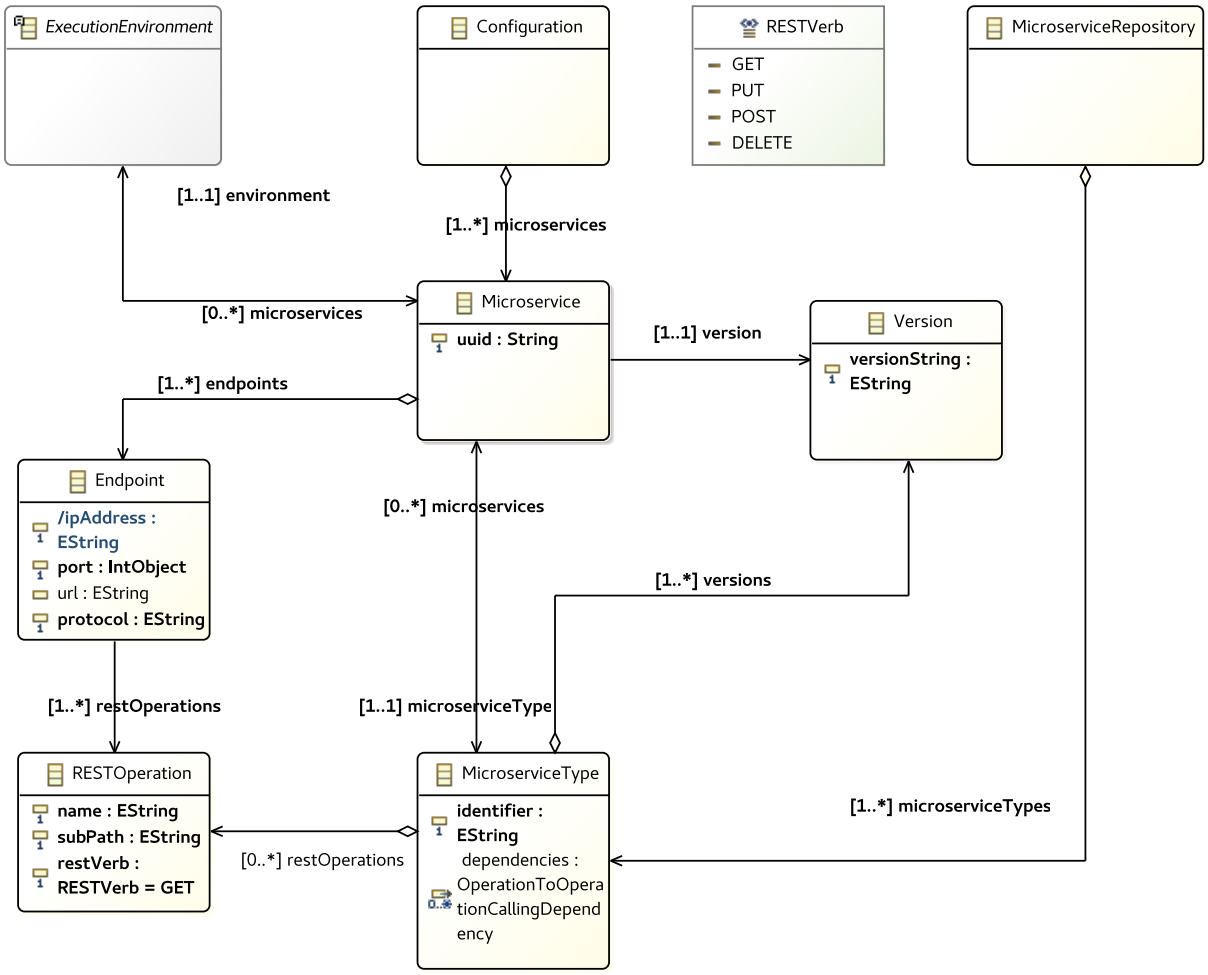

Figure A.3.: Ecore model visualization of the microservice model

A.4. Dependency Model

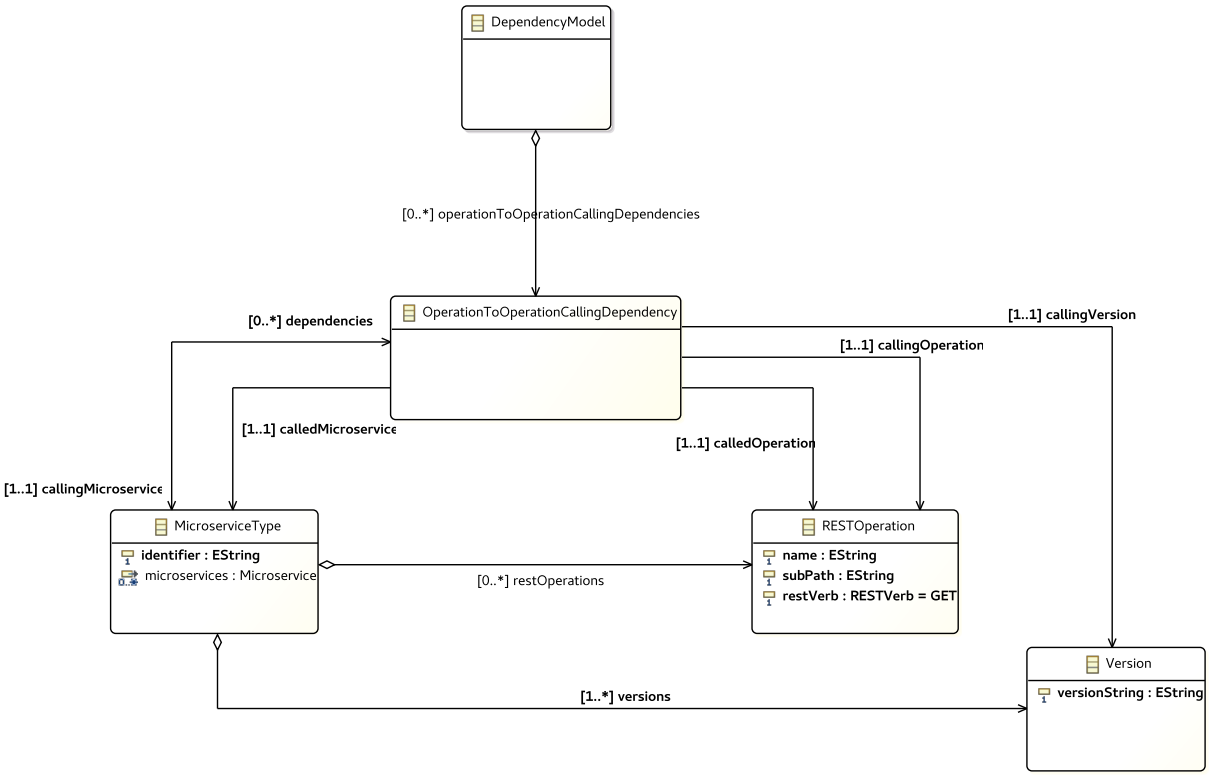

Figure A.4.: Ecore model visualization of the dependency model

A.5. Monitoring Model

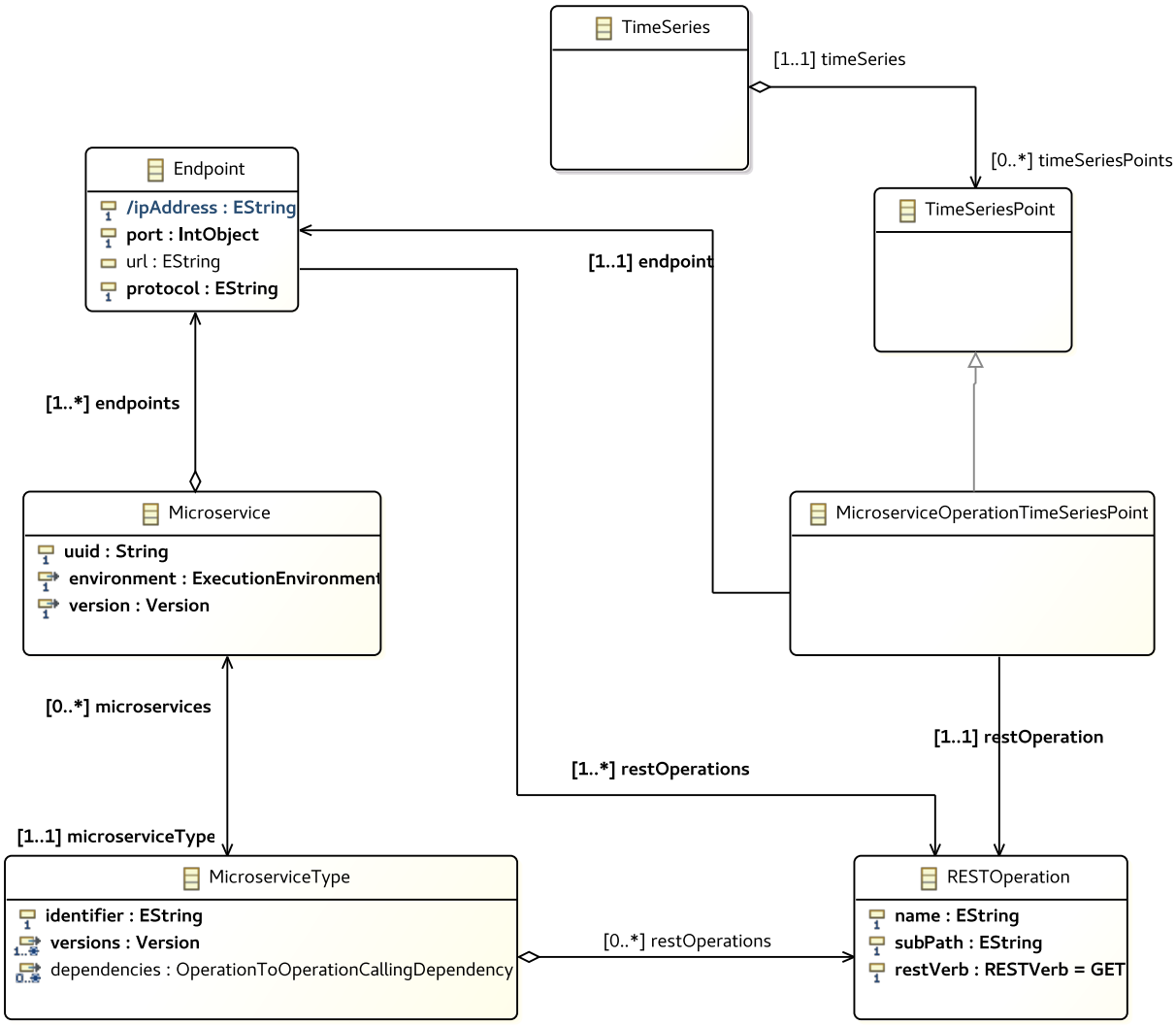

Figure A.5.: Ecore model visualization of the monitoring model

Appendix A

Bibliography

[BHJ16] A. Balalaie, A. Heydarnoori, P. Jamshidi. “Microservices Architecture Enables DevOps: Migration to a Cloud-Native Architecture.” In: *IEEE Softw.* 33.3 (May 2016), pp. 42–52. ISSN: 0740-7459 (cit. on p. 1).

[Bie12] T. C. Bielefeld. “Online performance anomaly detection for large-scale software systems.” Received b+m Software & Systems Engineering Award 2012. Diploma thesis. Kiel University, Mar. 2012 (cit. on pp. 4, 13).

[BWZ15] L. Bass, I. Weber, L. Zhu. *DevOps: A Software Architect’s Perspective*. Addison-Wesley Professional, 2015 (cit. on pp. 2, 8).

[CBK09] V. Chandola, A. Banerjee, V. Kumar. “Anomaly detection: A survey.” In: *ACM Comput. Surv.* 41.3 (2009) (cit. on p. 10).

[Coc15] A. Cockcroft. *Monitoring Microservices & Containers: A Challenge*. English. Gluecon 2015. May 2015 (cit. on p. 8).

[DEE+14] T. Düllmann, A. Eberlein, C. Endres, M. Fetzer, M. Fischer, C. Gregorian, K. Képes, Y. Noller, D. Olp, T. Rudolph, et al. *Online Performance Problem Detection, Diagnosis, and Visualization with Kieker*. 2014 (cit. on pp. 11, 42).

[Dül17] T. F. Düllmann. *Supplemental Material for the Master’s Thesis "Performance Anomaly Detection in Microservice Architectures under Continuous Change"*. https://dx.doi.org/10.5281/zenodo.226060. 2017. DOI: 10.5281/zenodo.226060 (cit. on pp. 5, 52, 59).

[EAD14] F. Erich, C. Amrit, M. Daneva. *Report: DevOps Literature Review*. 2014 (cit. on p. 8).

[Fow15] M. Fowler. *Microservices Resource Guide*. 2015 (cit. on p. 7).

[Fro13] T. Frotscher. “Architecture-Based Multivariate Anomaly Detection for Software Systems.” Master thesis. Kiel University, Oct. 2013 (cit. on pp. 4, 13).

[FSWG15] M. Farshchi, J.-G. Schneider, I. Weber, J. Grundy. “Experience report: Anomaly detection of cloud application operations using log and cloud metric correlation analysis.” In: *Software Reliability Engineering (ISSRE), 2015 IEEE 26th International Symposium on*. IEEE. 2015, pp. 24–34 (cit. on p. 12).

[FT02] R. T. Fielding, R. N. Taylor. “Principled design of the modern Web architecture.” In: *ACM Transactions on Internet Technology (TOIT)* 2.2 (2002), pp. 115–150 (cit. on p. 7).

[HF10] J. Humble, D. Farley. *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010 (cit. on p. 8).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. “Kieker: a framework for application performance monitoring and dynamic software analysis.” In: *Third Joint WOSP/SIPEW International Conference on Performance Engineering, ICPE’12, Boston, MA, USA - April 22 - 25, 2012*. 2012, pp. 247–248 (cit. on p. 14).

[Jen11] J. Jenkins. “Velocity Culture.” In: Velocity Conference. (Santa Clara, California). O’Reilly, 2011 (cit. on p. 1).

[KBS14] G. Kim, K. Behr, G. Spafford. *The Phoenix Project: A Novel About IT, DevOps, and Helping Your Business Win*. IT Revolution Press, 2014 (cit. on p. 8).

[LAF15] N. Laptev, S. Amizadeh, I. Flint. “Generic and Scalable Framework for Automated Time-series Anomaly Detection.” In: *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM. 2015, pp. 1939–1947 (cit. on p. 12).

[MRCW01] G. Mayle, J. Reves, J. Clubb, L. Wilson. *Automated adaptive baselining and thresholding method and system*. US Patent 6,182,022. Jan. 2001 (cit. on p. 2).

[MRHH09] N. S. Marwede, M. Rohr, A. van Hoorn, W. Hasselbring. “Automatic Failure Diagnosis in Distributed Large-Scale Software Systems based on Timing Behavior Anomaly Correlation.” In: *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR 2009)*. Ed. by A. Winter, R. Ferenc, J. Knodel. IEEE Computer Society, Mar. 2009, pp. 47–57 (cit. on pp. 4, 11).

[Net16] Netflix. *Completing the Netflix Cloud Migration*. Ed. by Y. Izrailevsky, S. Vlaovic, R. Meshenberg. Feb. 11, 2016 (cit. on p. 1).

[New15] S. Newman. *Building Microservices*. O'Reilly Media, Inc., 2015 (cit. on p. 7).

[Ott16] Otto. *A Completely Biased Summary of the Assets of Microservice Architectures*. Ed. by G. Steinacker. Mar. 20, 2016 (cit. on p. 1).

[POHG16] T. Pitakrat, D. Okanovic, A. van Hoorn, L. Grunske. "An Architecture-Aware Approach to Hierarchical Online Failure Prediction." In: *12th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA 2016, Venice, Italy, April 5-8, 2016*. 2016, pp. 60–69 (cit. on p. 12).

[PWC15] J. F. Pérez, W. Wang, G. Casale. "Towards a DevOps Approach for Software Quality Engineering." In: *Proceedings of the 2015 Workshop on Challenges in Performance Methods for Software Development, WOSP-C'15, Austin, TX, USA, January 31, 2015*. 2015, pp. 5–10 (cit. on p. 8).

[RBKR12] C. Rathfelder, S. Becker, K. Krogmann, R. Reussner. "Workload-aware system monitoring using performance predictions applied to a large-scale e-mail system." In: *Software Architecture (WICSA) and European Conference on Software Architecture (ECSA), 2012 Joint Working IEEE/IFIP Conference on*. IEEE. 2012, pp. 31–40 (cit. on p. 12).

[Roh15] M. Rohr. "Workload-sensitive Timing Behavior Analysis for Fault Localization in Software Systems." Dissertation (PhD thesis). Kiel, Germany: Faculty of Engineering, Kiel University, Jan. 2015. ISBN: 978-3-7347-4516-4 (cit. on p. 12).

[Sal94] P. H. Salus. *A quarter century of UNIX*. Addison-Wesley Reading, Massachusetts, 1994 (cit. on p. 7).

[SBPM09] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF. Eclipse Modeling Framework*. Addison Wesley, Jan. 12, 2009. ISBN: 0321331885 (cit. on p. 15).

[SS10] R. H. Shumway, D. S. Stoffer. *Time series analysis and its applications: with R examples*. Springer Science & Business Media, 2010 (cit. on p. 9).

[XZF+15] X. Xu, L. Zhu, M. Fu, D. Sun, A. B. Tran, P. Rimba, S. Dwarakanathan, L. Bass. "Crying wolf and meaning it: Reducing false alarms in monitoring of sporadic operations through pod-monitor." In: *Proceedings of the First International Workshop on Complex faUlts and Failures in LargE Software Systems*. IEEE Press. 2015, pp. 69–75 (cit. on p. 11).

[Zal16] Zalando. *From Monolith to Microservices*. Ed. by R. Schaefer. Feb. 10, 2016 (cit. on p. 1).

All links were last followed on January 2, 2017.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature