Visualization of Performance Anomalies with Kieker

Bachelor’s Thesis

Sören Henning

September 27, 2016

KIEL UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
SOFTWARE ENGINEERING GROUP

Advised by: Prof. Dr. Wilhelm Hasselbring
            Christian Wulf, M.Sc.
Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Kiel, 27. September 2016
Abstract

Reliability and performance are significant and often necessary quality characteristics of large software systems since these systems are often business-critical. Thus, continuous monitoring as well as real-time analysis become more and more important to recognize performance issues as soon as possible.

In this thesis, we present an approach and a corresponding implementation to visualize performance anomalies. Embedded in a web application, the visualization displays the execution times of operation calls and highlights detected anomalies in real time and interactively. Moreover, our approach utilizes the Kieker-based detection of anomalous behavior, which relies on wide parts on the ΘPAD and ΘPADx approaches. However, we had to customize these approaches to achieve an immediate analysis, which is required for the real-time visualization. The anomaly detection procedure is based on a TeeTime Pipe-and-Filter configuration and, in contrast to the monolithic ΘPAD and ΘPADx approaches, uses a microservice architecture.

In a feasibility evaluation, we demonstrate that both the detection of performance anomalies and their visualization work as expected. In a scalability evaluation, we show that our approach scales well to a particular point. For a workload beyond that point, we present solution approaches to increase the scalability.
# Contents

1 Introduction .............................................................. 1
   1.1 Motivation ......................................................... 1
   1.2 Goals .................................................................... 2
      1.2.1 G1: Migrate $Θ$PAD to a TeeTime Configuration ............ 2
      1.2.2 G2: Implement Multiple Forecast Algorithms ................. 2
      1.2.3 G3: Provide a Visualization of Measured Time Series and Detected Anomalies ........................................... 2
      1.2.4 G4: Evaluate our Approach .................................... 3
   1.3 Document Structure ................................................. 3

2 Foundations and Technologies ......................................... 5
   2.1 Time Series Analysis .............................................. 5
   2.2 Anomaly Detection ................................................ 5
   2.3 Performance Metrics ............................................. 5
   2.4 The Microservices Architectural Pattern ....................... 6
   2.5 The Kieker Monitoring Framework ............................... 6
   2.6 The Performance Anomaly Detection Approaches $Θ$PAD and $Θ$PADx ......................................................... 7
   2.7 The Pipe-and-Filter Framework TeeTime ....................... 9
   2.8 The Virtualization Project Docker ............................. 10

3 Approach ................................................................. 11
   3.1 Overview of Our Approach ..................................... 11
      3.1.1 Description of the Components .............................. 11
      3.1.2 Deployment and Execution of the Components .............. 13
   3.2 Time Series Analysis and Anomaly Detection with TeeTime ................................................................. 13
      3.2.1 Provided Algorithms ....................................... 14
      3.2.2 Provided Data Structures ................................ 17
      3.2.3 Anomaly Detection Stage .................................. 18
   3.3 Performance Anomaly Detection Component .................. 20
   3.4 R-based Forecasting Component ................................ 23
   3.5 Database Component ............................................. 24
   3.6 Visualization Provider Component ............................. 25
   3.7 Visualization Component ....................................... 26
      3.7.1 Description of the Visualization .......................... 26
      3.7.2 Architecture of the Component ............................ 28
Chapter 1

Introduction

1.1 Motivation

Many large enterprise applications and systems are running permanently. Since they are often business-critical, it is essential that they are always available and provide adequate performance [Cherkasova et al. 2009]. This especially applies to web applications. However, even modern applications are susceptible to unexpected and unwanted performance behaviors [Pertet and Narasimhan 2005]. The sources of these performance issues are manifold but often results of application faults, hardware failures, or a varying application load [Ibidunmoye et al. 2015].

Performance is a significant quality characteristic of software since a bad performance is related to higher costs. Nowadays, many businesses depend on web applications, whereby a bad user experience leads to leaving clients. For instance, as pointed out by Huang [2011], Amazon experiences 1% sales decrease due to an additional delay of 100 ms in response time while Google reports that 500 ms delay in response time leads to 20% drop in traffic.

Moreover, a bad performance can be an indicator of larger technical issues, which may eventually lead to failures of the whole system [Ibidunmoye et al. 2015]. The results of such a downtime are lost or dissatisfied customers, damage to the company’s reputation, and lost employee productivity [Pertet and Narasimhan 2005].

Therefore, a quick reaction on those issues is important to minimize the consequences, ideally, by rectifying the root of the problem. In many cases, however, a temporary solution such as providing additional computing power or deferring background tasks is more appropriate.

To detect performance problems as soon as possible, the execution time of program sections can be measured continuously and investigated for anomalies. The sooner anomalies are detected, the faster they can be treated. In addition to a simple binary signal, whether an anomaly has occurred or not, a visualization of the measured execution times is useful. Such a visualization enables a user, e.g., a system administrator, to look at the context within this anomaly occurred in detail and to interpret the reason for this anomaly.

With ΘPAD [Bielefeld 2012; Frotscher 2013], an approach was established, which reliably detects anomalies. However, it does not provide a visualization, its configuration is cumbersome, and it shows further difficulties, which we point out in the course of this thesis.
1. Introduction

1.2 Goals

The main goal of this thesis is the development of an approach including an associated implementation, which detects and visualizes performance anomalies. This approach uses the Kieker Monitoring Framework [van Hoorn et al. 2009; 2012] to continuously measure the execution times of operation calls. We rely on ΘPAD on certain parts but also need to adopt the approach.

To enable a realization of this main goal, we subdivide it into multiple, more specific goals.

1.2.1 G1: Migrate ΘPAD to a TeeTime Configuration

The anomaly detection procedure of ΘPAD is based on a Kieker Pipe-and-Filter architecture. As described by Wulf et al. [2014], Kieker’s Pipe-and-Filter framework shows architecture and performance issues. In particular, it is not capable for processing streams with a high amount of data, as it is required for performance anomaly detection. Thus, we want to migrate the current Pipe-and-Filter architecture of ΘPAD to one that uses the Pipe-and-Filter framework TeeTime [Wulf et al. 2014; 2016].

The part of our approach that analyzes monitoring data should be encapsulated and separated from the visualization as well as from the persistence. Therefore, it should be realized as a microservice [Wolff 2015; Newman and Lorenzen 2015; Hasselbring 2016].

1.2.2 G2: Implement Multiple Forecast Algorithms

ΘPAD uses forecast algorithms for its anomaly detection (see Chapter 2). As almost all of these algorithms are implemented with the R programming language, an R server has to be installed in addition to ΘPAD. This leads to a high configuration effort and a weaker performance comparing to a native implementation. Thus, we want to implement multiple forecast algorithms in Java, which use the average, a weighted average and regression to forecast the next value.

Simultaneously, we want to simplify the establishing of an R-connection to be still able to use the wide range of already available R algorithms. As an example, we want to migrate an R algorithm from ΘPAD that uses the ARIMA [Box and Jenkins 1990] implementation of the R forecast package [Hyndman and Khandakar 2008].

1.2.3 G3: Provide a Visualization of Measured Time Series and Detected Anomalies

We want to provide a visualization that shows measured execution times and detected anomalies in a user-friendly way. Therefore, the visualization should be in the form of a diagram that displays the duration of executions in the course of time and highlights detected anomalies. The visualization should be displayed in web browsers. Thus, it is
Another requirement is that the visualization will update itself because the measured values are recorded in real time and during the display of the visualization. It would not be user-friendly if the users have to reload the web page by themselves. Furthermore, the visualization should be updated fast enough. This means that we do not want to have a large delay between the detection of anomalies and its display.

The visualization should be encapsulated and separated from the analysis part and therefore realized as a microservice [Wolff 2015; Newman and Lorenzen 2015; Hasselbring 2016].

### 1.2.4 G4: Evaluate our Approach

We want to test our approach with various scenarios of a software’s execution behavior, which shows anomalies in its performance. Afterwards, we want to evaluate, whether our implementation meets our defined requirements. For this purpose, firstly, we examine, whether our analysis works as expected and, secondly, whether the visualization shows the expected data.

Since an automatic performance anomaly detection primarily is convenient for large software systems, we will evaluate the scalability of our approach. Thus, we examine with which frequency operation calls can be measured and analyzed until our implementation reaches its limits.

### 1.3 Document Structure

First, we describe the foundations of performance anomaly detection our approach relies on in Chapter 2. For this, we also introduce the most significant technologies we use. In Chapter 3 we present our overall approach and describe the individual parts it consists of in detail. The evaluations, which are intended by our goals, are described in Chapter 4. Here, we also show and discuss their results. In Chapter 5 we present related work and, finally, in Chapter 6 we summarize our work and point out future improvements of our approach.
Chapter 2

Foundations and Technologies

2.1 Time Series Analysis

A time series represents a sequence of measurements at regular temporal intervals. This leads to the formal definition as described by Mitsa [2010]: A time series $X = \{x_1, x_2, \ldots, x_n\}$ for $t = t_1, t_2, \ldots, t_n$ is a discrete function with value $x_1$ for time $t_1$, value $x_2$ for time $t_2$, and so on. The single values $x_1, x_2, \ldots, x_n$ are also called time series points.

For a time series, a model can be set up that constitutes its behavior as precise as possible. Such a model is, for instance, the autoregressive integrated moving average (ARIMA) model [Box and Jenkins 1990]. Afterwards, one can make forecasts based on the created model, which describe how the next values of this time series would look.

2.2 Anomaly Detection

As described by Chandola et al. [2009], anomalies are patterns in data which do not conform to the normal behavior.

A common approach to detect anomalies in time series is to make a prediction for every time series point based on its history and compare this prediction with the real measured value. This way one obtains a measure of the strengths of anomalies. Normalizing the differences between prediction and real value for every time series point, one can say how strong the anomaly (anomaly score) is at every time series point. To come to a decision whether an anomaly occurs or not, a threshold can be defined. If the anomaly score exceeds this threshold, the value can be considered as an anomaly. 2.6 details this procedure with an concrete approach.

2.3 Performance Metrics

In terms of computer science, performance is the time behavior and the resource efficiency of a computer system, as defined by Koziolek [2008]. In our work, we only refer to the time behavior which includes, for example, response time, throughput, and reaction time. To detect anomalies in performance of software systems, one can set up a time series of
measurements of a performance metric (response time in this work) and investigate it with the described procedure [Ehlers et al. 2011].

2.4 The Microservices Architectural Pattern

The microservices architectural pattern [Wolff 2015; Newman and Lorenzen 2015; Hasselbring 2016] is an approach to modularize software systems. That means, it is split up into modules, the microservices. In contrast to other modularization approaches, microservices are single programs that run as own processes. Every microservice should be limited to one task and can use its own technology stack (e.g., programming languages, database systems). They exclusively communicate via network (often by using REST [Fielding 2000]), thus, only the appropriated interfaces need to be created.

Microservice-based software architectures contrast with monolithic systems, which were pursued for a long time. In monolithic architectures, separation and encapsulating are only implemented inside an entire service. This impedes the maintenance and further development of systems since developers often need a deep knowledge of the entire system. Also a switching to other or new technologies is more difficult for large systems.

According to Tilkov [2014], a Self-contained System (SCS) is a composition of multiple microservices, which stands on its own and should not communicate with other SCS, to the extent possible. A key feature of SCS is that they have their own user interface. By following the SCS approach, software systems are composed of multiple SCS.

As required by our goals, our approach is based on microservices, so components can be developed independently from one another or replaced individually. Moreover, this facilitates the possible integration of our approach as SCS into a comprehensive software monitoring system.

2.5 The Kieker Monitoring Framework

Kieker [van Hoorn et al. 2009; 2012] is a framework for continuous monitoring and dynamic analyzing software systems’ runtime behavior. It consists of a monitoring part, which collects various types of measurements, and an analysis part, which interprets these data. In particular, the monitoring side is relevant for us, since it gathers the information of a running application we need for our approach.

A major advantage of Kieker is that it only produces a small amount of overhead [van Hoorn et al. 2009]. That is especially important for our approach, as we have to process a large number of measurements in a limited time.

Kieker uses aspect-oriented programming [Kiczales et al. 1997] to monitor already compiled applications. Thus, the source code of a software has not to be modified, but instead one can dynamically adjust which methods and components should be monitored.
2.6. The Performance Anomaly Detection Approaches ΘPAD and ΘPADx

Though Kieker was primarily developed for monitoring applications based on the Java platform, adapters for several other platforms exist. Therefore, our approach can be used with applications on all of these platforms.

Kieker provides many options to transfer the measurements to the analysis part. One of those is the TCP writer, which forwards the monitoring records via TCP. Thus, our analysis can run on a different host, so that monitoring and analysis are separated and cannot influence each other.

2.6 The Performance Anomaly Detection Approaches ΘPAD and ΘPADx

In the context of Kieker, the approach ΘPAD [Bielefeld 2012] was developed to detect performance anomalies and later it was extended by ΘPADx [Frotscher 2013]. These approaches were developed and evaluated in cooperation with the career-oriented social network XING. Since the implementations of both approaches were developed under the name ΘPAD, we subsequently use both names synonymously.

ΘPAD uses Kieker to continuously measure the execution times of methods, set up a time series, and check it for anomalies. For that, it uses the described method, which makes a prediction for the next value, compares it with the actual measurement and calculates an anomaly score with these values.

Since in most realistic application scenarios methods are not called in equal temporal distances, ΘPAD normalizes the sequence of timestamp-measurement-pairs to an equidistant time series. This normalization (in ΘPAD called discretization) divides the sequence into subsequences with equal durations and then aggregates the measurements of each subsequence to one value. Figure 2.1 shows an example of this process, whereby always four ticks are aggregated to one by the mean function. ΘPAD provides several different aggregation methods.

In the next step, an algorithm predicts what the next value for this time series would looks like. One can choose from various forecast algorithms and – depending on the algorithm – trends, seasonality and noise are taken into account. Some of the ΘPAD forecasters were implemented and investigated by Herbst et al. [2014] (see Chapter 5). The prediction always uses a sliding window that means something like the last x values or the last x time units. The reason for this is founded in the fact that the computation of a forecast is faster for a shorter time series, usually old values only have a low impact on the forecast, and, in particular, the time series is limited to memory restrictions. Figure 2.2 shows how ΘPAD makes a prediction for a sliding window containing four values by means of a forecaster that calculates the mean of all values.

Next, ΘPAD compares the prediction with the actual value and calculates an anomaly score from this (see Figure 2.3). This score is computed with the difference between prediction and measurement and normalized to the interval [0, 1] using a specific formula.
2. Foundations and Technologies

In Section 3.2.1 we describe this calculation in more detail. Furthermore, we point out to which problems this formula leads and present an alternative anomaly score calculation approach. Finally, a measurement is seen as an anomaly, if its associated anomaly score exceeds a previously defined threshold.

Evaluations of the ΘPAD approaches figured out that a suitable choice of parameters such as algorithms, time spans, and thresholds is strongly dependent on the application scenario and no universal parameters can be chosen.

The approach we developed in this thesis relies on ΘPAD in several points. The appendix Section A.2 presents a table that compares ΘPAD with our approach.
The Pipes-and-Filters pattern [Shaw 1989; Abowd et al. 1993; Taylor et al. 2010; Buschmann et al. 2013] is an architectural pattern for systems that process a stream of data. The individual processing steps are performed in components that are called filters. These filters can be connected by pipelines, so that objects can be sent through these pipelines and pass filter by filter.

TeeTime [Wulf et al. 2014; 2016] is a Java framework for developing software systems that are based on the Pipes-and-Filters pattern. It contains the basic entities stages, ports, pipes, and configurations. Stages are equivalent to the filters in the pattern. The framework provides multiple abstract stage types that can be extended at will. At its execution, a stage reads an object from its input ports, processes it in a defined way, and sends it to its output ports. In addition, the framework provides a number of predefined stages. A pipe connects an output port of one stage with an input port of another stage. In a configuration, stages could be defined and their ports connected. The framework takes care of creating the right pipes between the ports. Figure 2.4 shows a common visualization of Pipe-and-Filter configurations, which is also used hereinafter.

As described in goal G1, we will transform the Pipe-and-Filter architecture from ΘPAD to a TeeTime configuration.
2. Foundations and Technologies

2.8 The Virtualization Project Docker

The open-source project Docker [Docker, Inc. 2016; Boettiger 2014] deploys applications by means of virtualization.

Using Docker, one can wrap applications along with a defined infrastructure in *images*. This infrastructure contains everything the application needs to run. For example, system tools, system libraries, or required resources. Thus, software developers are able to deliver the infrastructure together with the application and, therefore, it does not have to be developed, tested, and maintained for different system configurations. This simplifies the deployment since it guarantees that the software behaves the same, regardless of the environment it is running in. The configuration of such an image is described by a `DOCKERFILE`. For instance, it configures the operating system this image is built on, the way this application should be started, or which interfaces such as ports should be accessible from outside.

Docker uses the containerization features of the Linux Kernel. In contrast to virtual machines, which emulate the entire hardware and all parts of the operating system, Docker produces a significantly lower overhead. Thus, in the context of microservices, single services are often encapsulated in individual Docker images. *Containers* are instances of images, which can be created, started, and stopped as often as desired to provide a scalability of services.

*Docker Compose* is a tool part of Docker to describe how a set of containers should be composed and linked. We use Docker and Docker Compose to containerize our microservice-based approach.
To visualize performance anomalies we take the following three steps: 1. Monitoring of an application to obtain performance measurement data; 2. Analyzing these monitoring data with respect to anomalies; 3. Visualizing of these results.

We use the framework Kieker for application monitoring. Kieker is able to measure the response time of chosen program methods and provide an interface to transmit them to other programs. OPAD already provides an approach to detect performance anomalies. However, one of our goals is to improve this approach and its corresponding implementation in certain criteria. A usable visualization does not exist so far.

3.1 Overview of Our Approach

The architecture of our approach consists of the five components Analysis, Database, R-based Forecasting, Visualization Provider, and Visualization as described in Figure 3.2. Since we use Kieker to collect performance data, the monitoring is not part of our approach and has to be set up additionally. In the following, we give a short overview of the components. A more detailed description of each component, including information about their implementations, is provided in the next sections.

3.1.1 Description of the Components

Analysis  It receives monitoring data and analyzes them regarding anomalies. Depending on its configuration it could invoke the R-forecasting component (see below) for certain algorithms. Afterwards, it stores the measurements and the analysis results in the database.
3. Approach

R-based Forecasting  It provides an R environment, to execute forecast algorithms, which are written in the R programming language. In our case, it is only necessary for the ARIMA forecaster, which is described in Section 3.2.1. However, ΘPAD contains much more R-based algorithms. Whether this component is necessary, depends on the configuration of the analysis component.

Database  It permanently stores the measurements and the analysis results to enable a restarting of the analysis (access to old data) on the one hand and providing an (asynchronous) access to them on the other hand.

Visualization Provider  It retrieves the analyzed time series from the database and provides them for the visualization. Moreover, it is responsible for delivering the visualization component itself, as described below.

Visualization  It visualizes the detected anomalies. This component is executed locally by a user in the web browser. Thus, it is delivered by the visualization provider component and retrieves the required data from it.
3.2. Time Series Analysis and Anomaly Detection with TeeTime

3.1.2 Deployment and Execution of the Components

All components except the visualization run on the server side. The responsibilities of the individual components are separated to an extent that they can be deployed almost independently, e.g., on individual hosts.

For this reason we decided to implement every component as an own microservice. The usage of this architecture pattern allows an uncomplicated replacement of single components. Thus, for example, one can change the database technology or replace the visualization provider by one with more functionalities. Since services communicate with each other exclusively via the network it is effortless to deploy the entire system distributed as well as deploy multiple instances of one service. Regarding our approach, this is relevant because this way we can separate the execution of the time-sensitive analysis from the more permissive visualization provider. Also the computationally intensive R-algorithms could be executed on different hosts (and possibly in multiple instances). The complete system as a composition of the single microservices is implemented as a self-contained system. Thus, it could be later part of a larger Kieker tool.

An important characteristic of microservices is their loose coupling. As described by Killalea [2016], failure of individual services should be expected, whereas this should not lead to crash of the entire system. If a single service becomes unavailable, others which depend on it should react on this and process tasks later or even not at all. In our case, this relates to the database and the R-forecasting service, so our implementation can handle the unavailability of them. Details on these implementations are described more precisely in the corresponding sections.

We created Docker images for all services and built a Docker Compose configuration which connects them to enable a set-up as simple as possible. The user does neither need to deploy all services separately nor install the necessary infrastructure (e.g., database, Java, R and additional extensions). Furthermore, even the operation system is included in the image. This is an additional advantage since, for example, on Windows, the underlying technology of our forecast service can only be used with restrictions (see Section 3.4). Using Docker containers, our implementation requires not more than a Docker installation.

3.2 Time Series Analysis and Anomaly Detection with Tee-Time

The anomaly detection procedure of continuous measurements is neither limited to method response times nor to performance measurements in general. Instead, also other time series can be investigated for anomalies in that way. For this reason, we decided to detach this general approach from the performance anomaly detection process and publish it as a separate open source library.

This library provides a TeeTime stage that performs anomaly detection. Therefore, it accepts measurements at its input port and decorates them with an anomaly score based
3. Approach

on their history.

In principle, our anomaly detection approach largely corresponds to the ΘPAD concept consisting of the steps normalization, forecasting, and anomaly score calculation. Thus, we use the same terminology as ΘPAD as described in Chapter 2.

3.2.1 Provided Algorithms

The process of anomaly detection relies on algorithms at various steps. To provide a design that is as generic as possible we implemented interfaces for these algorithms using the strategy pattern [Gamma et al. 1995]. Moreover, we provide implementations for these algorithms, which are described below.

**Aggregator** An aggregator is an algorithm that aggregates multiple values contained in a – not necessarily equidistant – time series to a single value. It is used in the normalization step of our analysis. Table 3.1 shows an overview of the aggregation methods we provide. All implementations are based on the Apache Commons [Apache Software Foundation 2015] math library.

<table>
<thead>
<tr>
<th>Aggregator</th>
<th>Aggregation Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeometricMeanAggregator</td>
<td>Geometric Mean</td>
</tr>
<tr>
<td>MaxAggregator</td>
<td>Maximum</td>
</tr>
<tr>
<td>MeanAggregator</td>
<td>Mean</td>
</tr>
<tr>
<td>MinAggregator</td>
<td>Minimum</td>
</tr>
<tr>
<td>PercentileAggregator</td>
<td>Percentile (configurable)</td>
</tr>
<tr>
<td>ProductAggregator</td>
<td>Product</td>
</tr>
<tr>
<td>SumAggregator</td>
<td>Sum</td>
</tr>
<tr>
<td>SumLogAggregator</td>
<td>Sum of the Natural Logs</td>
</tr>
<tr>
<td>SumSqAggregator</td>
<td>Sum of the Squares</td>
</tr>
<tr>
<td>VarianceAggregator</td>
<td>Variance</td>
</tr>
</tbody>
</table>

**Forecaster** A forecaster predicts the next value of a time series. Figure 3.3 shows the algorithms we have implemented. The MeanForecaster makes a prediction by calculating the average of all time series points. Thus, all points contribute to the resulting value in an equal ratio. We have implemented the abstract class AbstractWeightedForecaster for a weighted average. A forecaster that inherits from it must implement a method that assigns a weight to every time series point. We provide forecasters with exponential (ExponentialWeightedForecaster), linear (LinearWeightedForecaster), and logarithmic (LogarithmicWeightedForecaster) increasing weights, whereby more recent values mean a higher weight. The weighted forecasters are based on the work of Wiechmann [2015]. The
3.2. Time Series Analysis and Anomaly Detection with TeeTime

RegressionForecaster uses simple regression from the Apache Commons [Apache Software Foundation 2015] math library, which estimates an ordinary least squares regression model with one independent variable. The AbstractRForecaster provides an R connection based on Rserve, so one can write and use concrete R-based forecasters. The library contains the ARIMAForecaster, which uses the R function auto.arima() [Hyndman and Khandakar 2008] for forecasting. It automatically sets up an appropriated ARIMA [Box and Jenkins 1990] model for the given time series by using various approximations.

Anomaly score calculator An anomaly score calculator computes a numeric value that indicates how strong an anomaly is based on an actual value and a predicted value. Our library provides the SimpleAnomalyScoreCalculator, which was introduced in ΘPAD, on the one hand and the DeviationAnomalyScoreCalculator, which we have implemented in the context of this thesis, on the other hand.

The ΘPAD calculator computes the anomaly score $A_S$ by means of the following formula, where $m$ is the measurement and $p$ the prediction: $A_S(m, p) = \left| \frac{p - m}{p + m} \right|$. It has the disadvantage that an equal deviation of measurement and prediction up or down leads to different anomaly scores. For instance, if the measured value is 50% less than expected, the anomaly score would be 0.33, but if the measured value is 50% more than expected, it leads to the score 0.2. Moreover, the anomaly score does not scale linearly (see Figure 3.4). Thus, comparing anomaly scores is difficult.

---

3. Approach

![Figure 3.4](image_url)  
Figure 3.4. Course of the anomaly score in relation to the actual value with a fixed prediction of 1 using the ΘPAD anomaly score calculator

Our calculator computes the anomaly score $A_D$ with the deviation between the measurement $m$ and the prediction $p$ relatively to the prediction: $A_D(m, p) = \frac{m - p}{p}$. It follows that the sign of the computed anomaly score indicates whether the anomaly is an outlier on the high or on the low side. If the anomaly score is positive, this means that the actual value is greater than expected; a negative anomaly score shows that the actual value is lower than the prediction. For example, a measurement that is twice as high as expected results in an anomaly score of 1. If prediction and actual value are the same, the anomaly score is 0. A measured value of 0 leads to an anomaly score of $-1$. Hence, we make a different treatment of outliers on the high and on the low side possible. Also, to enable a better interpretation of the score, it can be useful to show this in a provided visualization. Thus, one can see if a measured value is greater or lower than expected without considering the prediction value. If the information in which direction the value differs from the prediction is not important, the absolute value of the anomaly score can be calculated. In contrast to the ΘPAD calculator, an equal deviation of measurement and prediction up or down leads to the same anomaly score, which only varies in its sign. For instance, if the actual value is 50 % less than expected, the anomaly score would be $-0.5$. If the actual value is 50 % more than expected, the score would be 0.5. This anomaly score is not restricted to the interval $[0, 1]$ anymore so that anomaly scores can reach any size. However, unlike the ΘPAD calculator, it scales linearly, i.e., the ratio of actual and predicted value is proportional to the resulting anomaly score, so that comparing anomaly scores is more intuitive (see Figure 3.5).
3.2. Time Series Analysis and Anomaly Detection with TeeTime

![Graph](image)

**Figure 3.5.** Course of the anomaly score in relation to the actual value with a fixed prediction of 1 using our anomaly score calculator

![Class Diagram](image)

**Figure 3.6.** Class diagram of our measurement classes

### 3.2.2 Provided Data Structures

**Measurements** Figure 3.6 shows a class diagram of our measurement classes and shows their hierarchy. A Measurement object consists of a point in time, i.e., an exact point on the time line (implemented as Java Instant\(^2\)), and the measured value (implemented as double). A ForecastedMeasurement additionally has a predicted value and an AnomalyScoredMeasurement has both the predicted value and an anomaly score.

\(^2\)available since Java 8 and located in the package java.time.Instant
3. Approach

![Diagram of time series classes](image)

**Figure 3.7.** Class diagram of our time series classes

**Time series**  As described in Figure 3.7, our implementation provides classes for time series modeling. A TimeSeries corresponds to a time series that is not necessarily equidistant. It contains multiple TimeSeriesPoints, which consist of a point in time and a value (see measurement classes). An EquidistantTimeSeries is a time series, which guarantees that all time series points have equal distances. This corresponds to the formal definition of a time series given in Chapter 2. A BoundedTimeSeries is configured with a certain time interval and can only include time series points whose times are within this interval. If new points are added to the end of the time series (i.e., they have the latest time stamp), the class removes points at the time series beginning until all time series points are within the configured time interval again. More formally, this means we have a time series \( x_1, \ldots, x_n \) with time stamps \( t_1, \ldots, t_n \) and an interval duration \( d \). If a new value \( x_{n+1} \) is added to this interval, all values \( x_i \) with \( t_i \leq t_{n+1} - d \) are removed. For example, in our case, we use this time series for modeling the sliding window.

Both EquidistantTimeSeries and BoundedTimeSeries are subtypes of TimeSeries. Our implementation is inspired by the Java Collections Framework\(^3\). However, we do not support all functionalities of it and added new ones instead.

### 3.2.3 Anomaly Detection Stage

The anomaly detection stage exists in two variants – with storage and without – and can be configured by several parameters as described below.

- **Sliding window**  The time interval, whose measurements are used for the prediction. E.g., the last 10 minutes

- **Normalization interval**  The time interval, which should be aggregated to one value. E.g., always one second

- **Normalization algorithm**  An aggregator that aggregates the values in a normalization interval to one value.

---

\(^3\)https://docs.oracle.com/javase/8/docs/technotes/guides/collections
3.2. Time Series Analysis and Anomaly Detection with TeeTime

**Forecast algorithm** A forecast algorithm that predicts the next value based on a time series.

**Storage adapter** An adapter class that is able to store single measurements and retrieve all stored measurements in a given interval.

The anomaly detection stage is a TeeTime **CompositeStage** that means it is assembled by several single stages. Figure 3.8 shows how this stage is composed.

![TeeTime configuration of the anomaly detection stage](image)

**Figure 3.8.** TeeTime configuration of the anomaly detection stage

At its input port the anomaly detection stage expects an object of type `Measurement`. Therefore, in most cases one has to connect an adapter stage in front of it that converts the data to the required type. The first stage, i.e., the `Distributor`, forwards the data at its input port to all output ports by copying the reference. The `Time Series Loader` holds a reference to the sliding window, which is implemented as a `BoundedTimeSeries`. When this stage receives a measurement at its input port, it sends a copy of the current sliding window to its output port and then appends the new measurement to the sliding window. In the next steps, the `Normalizer` performs the time series normalization and the `Forecaster` predicts the next value for this time series. The stages have to be configured with an aggregation algorithm and a forecast algorithm, respectively. The `Measurement Forecast Decorator` combines the original measurement and the predicted value to a `ForecastedMeasurement`, which is transformed by the `Anomaly Score Calculator` using a configurable algorithm. Next, the following `Distributor` sends the data to the `Storager`, which saves all incoming data, e.g., in a database, and to all set up Threshold Filters. A Threshold Filter forwards its incoming `AnomalyScoredMeasurements` to its output port, if they exceed or fall below a configurable threshold. For example, one can connect a stage to it that sends an email to the system administrator, if the anomaly score exceeds a given threshold.

In order to save the data, a `StorageAdapter` has to be configured, which specifies the way the data will be saved. Therefore, it uses the adapter pattern [Gamma et al. 1995]. In addition, it provides an opportunity to read the saved data in a certain interval. Thus,
3. Approach

the Time Series Loader loads an already existing sliding window (if available) and after
a restart, the analysis does not have to start from scratch. Our library provides such an
adapter for the Cassandra database.

All stages of the anomaly detection stage can be also used separately. Furthermore, our
library provides some utility stages, e.g., for measurement generation or JSON export of the
analyzed data.

Compared with the ΘPAD implementation we changed many implementation details
to achieve a higher reusability and a looser coupling of the single parts and also to simplify
its usage. Our approach executes the normalization and forecast steps for every incoming
measurement, whereas ΘPAD firstly fills one normalization interval and processes it
afterwards. By changing this, we enable the visualization (see Section 3.7) to show all
measurements instead of only showing the aggregated ones. Furthermore, they can be
immediately displayed and not just after a certain aggregation interval is filled.

3.3 Performance Anomaly Detection Component

We implemented the analysis component as a service called KiekPAD-Analysis. It analyzes
the Kieker monitoring data by means of a TeeTime Pipe-and-Filter configuration. For this
purpose, it uses our open source library, which is described in Section 3.2.

![Figure 3.9. TeeTime configuration of the analysis component](image)

Figure 3.9 shows the Pipe-and-Filter configuration. The initial TCP Reader receives the
monitoring records, which are generated by the Kieker monitoring component, via TCP.
The following Flow Record Filter forwards only IFlowRecord objects, which is the current
standard type for monitoring records (Kieker 1.12). Other types are not handled by our
analysis configuration. IFlowRecord is a super type of TraceMetadata, BeforeOperationEvent,
and AfterOperationEvent. The TraceMetadata record is sent at the beginning of a call trace’s
execution and contains metadata, such as the host name. The BeforeOperationEvent and
AfterOperationEvent provide the time when a method is entered and left, respectively.

The Record Reconstructor is responsible for assembling before and after events as well as
metadata to one object that contains the time the operation execution started, the duration
3.3. Performance Anomaly Detection Component

of the execution and all metadata. We do not generate call trees because we only analyze concrete methods and do not consider the recursive context. Kieker monitoring records do not provide an absolute time stamp, from which date and time can be read off, but merely a relative one that allows to determine the temporal distance between records. Since the measurements arrive in real time, we know the recording happens approximately at the same time as the analysis. Therefore, we take the absolute time the first record arrives and set it as time for this record. For all further records we calculate the time by means of the first record’s time and the relative time stamp. In formal terms this means we get the absolute time \( a_x \) for the \( x \)th record as follows, where \( r_x \) is the relative time \( x \) was recorded and \( c_x \) is the time \( x \) is analyzed.

\[
a_x = \begin{cases} 
c_x, & \text{if } x = 1 \\
 c_1 + r_x - r_1, & \text{otherwise}
\end{cases}
\]

In a future version of Kieker the metadata record will contain the absolute system time that can be used to calculate the absolute time for the belonging event records.

The actual anomaly detection takes place in individual branches, whereby every branch handles a custom type of operation calls. In a typical case every branch investigates a different method. For example, in a web application there could be one branch per available web page. Using individual branches has the advantage that the anomaly detection stages do not have to maintain multiple time series to allocate the proper history to a measurement. Another benefit is that every branch can be configured independently and could have its own algorithms and time intervals for normalization and forecasting.

However the division to separate branches can be made by other criteria than the method name so that we introduce a record filter for that. A record filter tests whether operation calls are of a certain type. Mathematically expressed, a filter represents a function \( f : \text{MonitoringRecord} \rightarrow \{\text{true}, \text{false}\} \). Record filters can be configured by the criteria operation signature, class signature, host name, session identifier, and thread identifier. If no value is set for a criteria, this means it is not tested and acts like a wildcard. The procedure to create record filters is derived from the builder [Gamma et al. 1995] and the fluent API [Fowler 2010] patterns. Listing 3.1 gives two examples of a filter creation. \textit{filter1} matches for all records with the operation signature \textit{public void store.CartController.checkout()}. \textit{filter2} restricts additionally the records to those which have the host name \textit{srv01}.

The \textit{Record Distributor} can have several output ports, which are associated with a record filter. It forwards all incoming records to the output ports whose filters accept the record. Internally, the Record Distributor is a simple Distributor that sends its incoming data to all output ports by copying their reference. Every output port is connected to a Filter Stage that uses the associated record filter and forwards only the matching records.

The first step of every single branch is the \textit{Record Converter} which transforms the monitoring records to the more general type \textit{Measurement}. Then the \textit{Anomaly Detector}, which is the anomaly detection stage from our open source library, tests whether this measurement is an anomaly or not.
3. Approach

**Listing 3.1. Examples of filter creations**

```java
RecordFilter filter1 = RecordFilter.builder()
    .operationSignature("public void store.CartController.checkout()")
    .build();

RecordFilter filter2 = RecordFilter.builder()
    .operationSignature("public void store.CartController.checkout()")
    .hostname("srv01")
    .build();
```

**Listing 3.2. Example property file that creates a branch**

```properties
id=checkout
normalizationDuration=PT5S
slidingWindowDuration=PT1H
forecaster=RegressionForecaster
aggregator=MeanAggregator
filter.operationSignature=public void store.CartController.checkout()
filter.hostname=srv01
cassandra.table=measurements
```

The branches are created with Java Property Files (.properties), where exactly one file creates exactly one branch. It specifies the parameters for the anomaly detection on the one hand and the filter’s configuration on the other hand. Listing 3.2 shows an example of a branch configuration’s file. Every branch needs an identifier as defined in the first line. All other properties do not have to be set since there are default values for them. Line 2 and 3 set the normalization interval to five seconds and the sliding window to one hour. The time periods have to be declared in the ISO-8601 [ISO 1988] duration format. The next two lines specify the forecast and aggregation algorithm, which are constructed using the Java Reflection API. By passing fully qualified class names custom algorithms can be used. Line 6 and 7 specify that this branch handles operation calls of the method `public void store.CartController.checkout()` on host `srv01`. The property names starting with `cassandra.` determine the table and column names of the database.

As already described in Section 3.1.2 we have to handle broken connections to the database and to the R-providing component. When the database becomes unavailable, the Cassandra driver for Java continuously tries to reestablish the connection until the database is available again. If a database request is made while it is unavailable, an exception is thrown, which can be caught. One could react to this by attaching the dataset to a queue and save it once the database is available again. However, since this could affect the performance
3.4. R-based Forecasting Component

and it is not important to have every single record saved, we discard it. In addition, a breakdown of a service should be rare regarding the time it is available. We implemented a similar process if the connection to the R component is lost. The current request is rejected to not slow down the analysis and, simultaneously, a new try to reconnect is made.

It is important that the database connection is established before the analysis starts because it may be required to create tables or load an existing sliding window. Hence, the analysis waits for the availability of the database during its start-up phase.

3.4 R-based Forecasting Component

In the context of ΘPAD and related work several forecast algorithms were investigated (see Chapter 5), whose implementations are written in R (e.g., by using the R forecast library [Hyndman and Khandakar 2008]). For most of these forecast algorithms there exist no or only commercial implementations in Java. Since writing own implementations for these algorithms is laborious, we decided to continue relying on R for them.

To execute R scripts from a Java program numerous opportunities exist. Table 3.2 gives an overview of common ones. One of them is the Java library JRI [Urbanek 2015], which uses the Java Native Interface (JNI) to execute R commands.

Rserve [Urbanek 2003] is an R package that provides a server. This server is listening for TCP connection requests on a specific port and processes R commands via this connection. In order to use Rserve, R must be installed and the R server has to be started. Afterwards, the Rserve Java library can be used to execute R commands. ΘPAD relies on Rserve by using RCaller⁴, which attends to start the server. Rserve has the major drawbacks that

⁴https://github.com/jbytecode/rcaller

<table>
<thead>
<tr>
<th>Table 3.2. Overview of different Java libraries to execute R scripts</th>
</tr>
</thead>
<tbody>
<tr>
<td>Communication</td>
</tr>
<tr>
<td>R installation required</td>
</tr>
<tr>
<td>Multiple processes possible</td>
</tr>
<tr>
<td>Multiple hosts possible</td>
</tr>
<tr>
<td>Performance⁵</td>
</tr>
<tr>
<td>Forecast library available</td>
</tr>
</tbody>
</table>

⁵according to Satman [2014]
3. Approach

multiple independent session are not directly possible on Windows. As with JRI, R still has
to be installed and the required environment variables must be set.

Another option is RCaller [Satman 2014], which was developed to provide a more
developer-friendly API. It creates a separate R process that executes R scripts. However, it
is much slower than the previously described alternatives.

Renjin [BeDataDriven B.V. 2016] is an R interpreter for the Java virtual machine (JVM). It
would be very convenient for us, since it runs natively on the JVM, and therefore no
installation of R and further set-up is necessary. Moreover, it should be faster than the other
opportunities as no serialization is required. However, the R forecast package does not
exists for Renjin so far, but if this changes later, this approach would be an opportunity.

Each of the three remaining libraries requires a high configuration effort for the users of
our approach because, ideally, they should not care about installing R. By using Docker, we
can avoid this disadvantage. Following the microservice pattern, we separate the analysis
and the R-based forecasting and wrap both in single containers. This also enables a better
scalability since we can start multiple equal Docker containers if required. We selected
Rserve as the best solution for our approach, since by using Docker we need network
communication anyhow. Otherwise, we would have to implement a service that translates
the network requests to system calls.

3.5 Database Component

After the monitoring data has been inspected for anomalies, these results should be stored
permanently. Thus, they can be accessed asynchronously by the visualization component
on the one hand. On the other hand, this enables a more appropriate restart of the analysis
component since it can rely on already existing data for forecasting. The data that should
be stored, has a simple and plain structure. There only exists one type of data records,
called measurements, which consists of the fields time, measured value, prediction, and
anomaly score.

To store these records, we rely on a database. Since there exist many different databases,
we compared three of them, which base on different paradigms. These are the relational
database MySQL [MySQL AB 2016], the NoSQL databases MongoDB [MongoDB Inc.
2016], which is document-oriented, and Cassandra [Lakshman and Malik 2010], which is
column-oriented.

The major criterion is a fast writing of the records to not inhibit the analysis processing.
In particular, a high throughput (stored records per unit time) is required, whereas latencies
(delays during the saving) are tolerable. If the analysis is not fast enough, this can lead to
a collapse of this component, in contrast to the visualization, where delayed values are
bearable. Thus, we do not focus on a fast reading of records.

According to the performance evaluations of Rabl et al. [2012] and Lee [2011], Cassandra
shows the best results for these requirements. Since we have a large amount of uniform
data, which are not related to each other, a column-oriented database is preferable to the
other two types.

We rely on Cassandra in the form of a Docker container provided by Spotify\(^5\) because it has a shorter startup time than the official one. In our implementation, Cassandra is configured with one node and one table that contains the data records. As described in Section 6.2, this can be adjusted and improved in several aspects.

### 3.6 Visualization Provider Component

The visualization provider component is implemented as a service called *KiekPAD-Viz-Provider*. It contains an embedded web server to process HTTP requests. Since the visualization component is executed locally in a web browser, the first task of the visualization provider is to provide this component. Therefore, it delivers a static HTML page that contains the visualization component.

The second task is to deliver the required data for the visualization. Thus, it provides a REST API [Fielding 2000] with the following two interfaces:

1. **GET**: `/series`
2. **GET**: `/measurements?series=<series>&after=<after>`

The first interface lists the identifiers of all time series, i.e., analysis branch configurations, that are available in the database. Thus, the visualization can enable the user to select between these time series.

The second interface delivers all measurements of the time series specified by `<series>`. A measurement is a record comprising the actual measured execution time, prediction, anomaly score, corresponding time stamp, and a boolean value whether this record is aggregated (see below). The optional parameter `after` specifies that only measurements are loaded and returned, which are measured after the passed time stamp `<after>`. This time stamp has to be defined as epoch milliseconds, i.e., the number of milliseconds that has elapsed since 00:00:00 (UTC), 1 January 1970.

Since our visualization shows time series with milliseconds precision only – but the analysis and the database work on nanoseconds precision – this component performs an aggregation. Thereby, all data records with the same amount of milliseconds are aggregated to one record. In our case, we aggregate the records to the averages of their value for measurement, prediction, and anomaly score. However, we provide an interface for such aggregators, so other algorithms can be used by applying the strategy pattern. It is not clear whether this is necessary in its practical use. It could be important, since in many application scenarios methods are called more than one time per millisecond. One example is the news feed in a social network that is requested by thousands of users per second. However, if this high frequency of method calls remains over a longer period, an aggregation is already

\(^5\)https://hub.docker.com/r/spotify/cassandra
3. Approach

essential in the analysis because also the procedure of anomaly detection takes time (see Chapter 4 and Chapter 6).

As previously described, the component has to handle the temporary unavailability of other services. The visualization provider component is only linked to the database, so that it only has to react on a breakdown of the database. Since the component cannot retrieve any data without a database, it returns a 503 Service Unavailable HTTP error code in that case. In practice, this works in the same manner as the analysis component (see Section 3.3).

The visualization provider is based on the Java framework Spring Boot, which is part of the Spring project [Pivotal Software 2016]. Similar to the Java REST API (JAX-RS), Spring Boot also allows to map REST requests to method calls including a correct type conversion. Furthermore, Spring Boot allows to start an embedded web server and, thus, to create stand-alone applications that run without an external web server. Spring Boot follows the software design paradigm Convention over Configuration [Chen 2006] to enable a faster development of (often web) applications, since it minimizes the configuration effort. Moreover, it supports several other features that can be used in further developments, for example, an automatic caching of request responses.

3.7 Visualization Component

Our visualization component is a single-page application [Mikowski and Powell 2013] based on HTML and JavaScript that is accessible via a web browser. It visualizes execution times and occurring anomalies in the course of time, after they were investigated and stored by the analysis. For this purpose, the visualization component periodically requests new data sets from the visualization provider component. The visualization is interactive and has a user interface for controlling.

3.7.1 Description of the Visualization

Figure 3.10a shows a screenshot of the visualization. The most important part is the time-values-plot (1). It shows the course of execution times in nanoseconds (vertical axis) in relation to the point in times when they were executed (horizontal axis). On top of it a triangle with exclamation mark (2) is shown, if the value at the associated moment is recognized to be an anomaly.

The displayed temporal window can be moved by click and dragging in the plot. A scrolling within it causes an increase or decrease of the displayed time interval. When new values are added to the plot, the window automatically moves forward, on condition that the user does not have moved the window to the past.

Above the chart, a drop-down button (3) allows to switch between the different time series that are stored in the database. By clicking the gearwheel button (4), further adjustment options are displayed.
3.7. Visualization Component

(a) Visualization in the default view

(b) Visualization with activated prediction and anomaly score plots and advanced UI options

Figure 3.10. Screenshots of the visualization

Figure 3.10b shows these options and a different configured visualization. The button *Anomaly Scores* (5) shows a second chart (6) below the other one, which shows the anomaly scores (vertical axis) in relation to the time (horizontal axis). Both the measurements plot and the anomaly scores plot behave equally on resizing and moving, so that in both plots
equal time stamps are always displayed among each other. By clicking the button Predictions (7), the first chart shows the predicted value (blue) in addition to the actual one (orange).

Using the two input fields for thresholds (8) one can configure at which anomaly score a value should be recognized as an anomaly and the indicator would be displayed. A lower threshold can be defined as well as an upper one. The input field Refresh Interval (9) specifies how often the visualization requests for new data. For instance, a value of 500 means that every 500 milliseconds a request is made.

3.7.2 Architecture of the Component

Figure 3.11. Architecture of the visualization component

Figure 3.11 shows a graphical overview of our visualization’s architecture. The JavaScript plotting library CanvasPlot [Johanson 2016] creates the plots in the web browser including the drawing of axes, grids, labels, and the actual curves. Moreover, it provides the interactive zooming and moving. CanvasPlot is based on D3 [Bostock et al. 2011], but has been adopted to visualize large data sets with a better performance. This is achieved by relying on a mixture of SVG and the HTML canvas element, instead of D3’s pure SVG solution. We extended CanvasPlot by the functionality to show indicators in the upper area of the plot.

Like the anomaly detection process, the visualization of anomalies is not restricted to performance measurements. Therefore, we have outsourced as much as possible to an open source library namely Anomaliz.js. It is based on CanvasPlot and provides a JavaScript object that represents the browser-based visualization and can be customized, also at runtime, in many parts. Unless it is otherwise configured, it creates the two plots and is responsible for keeping them synchronous. Also it manages the data sets and supports the dynamically adding of new records, i.e., measurement, prediction, anomaly score, and time. Furthermore, it shows the indicators, if the thresholds are exceeded, and updates the displayed temporal window if necessary.

Our actual visualization component consists of two parts. The first is responsible for the handling of user interactions, for example, switching between time series or changing the thresholds. The second part is responsible for the communication with the server, namely
the REST API of the visualization provider component. During start-up, it asynchronously requests (AJAX) the identifiers of all available time series at once to allow switching between these times series. Subsequently, it makes asynchronous requests in the user-defined interval to receive all records of the selected time series that are more recent than the latest displayed values.
Chapter 4

Evaluation

As required in our goals we test our approach to analyze how practicable it is for real deployment scenarios. Therefore, on the one hand, we evaluate whether the analysis and the visualization meet our defined requirements and deliver suitable results. On the other hand, we evaluate how large the application we are monitoring and investigating can be regarding to the frequency of requests it handles. For both evaluations we describe the underlying concept of the experiment and its tested scenarios. Afterwards, we show the results and discuss them. Finally, we display the experiments’ threats to validity.

4.1 Evaluation Environment

4.1.1 Evaluation Tool

The monitoring data sets, which our approach processes, are continuously occurring measurements, whereby the starting time varies as well as the execution time.

In order to monitor different scenarios of a software’s behavior, we developed a tool that simulates the execution of a software according to a defined pattern. Specifically this means that it executes a set of methods, where each is described by a configuration. A configuration consists of four parameters as described below.

Call time function The method shall be called at intended points of time. For this purpose, a function \( t : \mathbb{N} \rightarrow \mathbb{N} \) describes the time span between method calls. For a number of milliseconds since the application’s start \( x \), it supplies the time span until the next method call in milliseconds \( t(x) \), if the method is called at \( x \). For instance, the function \( t(x) = 100 \) declares that the method is called every 100 milliseconds. By using a function as parameter instead of a static value, distances between method calls can be chosen that vary and are not always equal.

Duration function Also, the duration of the method executions is determined by a function \( d : \mathbb{N} \rightarrow \mathbb{N} \). For a number of milliseconds since the application’s start \( x \), it supplies the execution time in milliseconds \( d(x) \).
4. Evaluation

**Method** The method, which should be called, has to be configured. All our evaluations use the same method that does nothing but sleeping for the calculated duration.

**Multi-threaded** When using multiple threads a dynamic thread pool will be generated so methods can be executed concurrently. If only a single thread is used, methods could be called delayed, in particular if the execution time is longer than the time between the method calls. Thus, we use multiple threads for all evaluations. However, multiple threads are not always necessary and could decrease the performance, thus, they can be disabled.

For both parameter functions $t$ and $d$, we use the following auxiliary functions, for which we also provide implementations.

$\text{noise}_{v,s} : \mathbb{N} \to \mathbb{N}; v, s \in \mathbb{N}$ It returns a pseudo-random value in the interval $[-v, +v]$. This pseudo-randomness appears to be random but produces always the same values for the same seed $s$. Thus, our tests are reproducible. By adding the result of this function a noise will be generated.

$\text{staticAnomaly}_{v,T} : \mathbb{N} \to \mathbb{N}; v \in \mathbb{Z}; T \subseteq \mathbb{N}$ It transforms an input value $x$ to a value $v$ for $x = t, t \in T$. Otherwise, it returns 0. By adding the result of this function to the values of a time series, anomalies will be generated at the time stamps $T$.

4.1.2 Hardware and Software Environment

We evaluated our approach on a cloud node at the Software Performance Engineering Lab\(^1\) of the University of Kiel. To make our results comparable, all experiments are executed on the same system, whose hardware and software configuration is described in Table 4.1. The memory of this system is large enough to not limit our evaluation. Moreover, the high number of CPU cores means more real parallelization so our single components should not influence each other.

4.2 Feasibility Evaluation

We adopted the ΘPAD approach and developed new implementations for most parts. To show that our approach is still feasible, we perform an evaluation. Thus, we developed scenarios of a software’s execution, which we have tested and evaluated, whereby we selected one of those to describe their results in detail.

A significant process in our approach is the time series forecasting. Since the quality of forecast algorithms was already examined in other works (see Chapter 5), we focus on the feasibility of our entire approach.

\(^1\)http://www.se.informatik.uni-kiel.de/en/research/software-performance-engineering-lab.spel

32
4.2. Feasibility Evaluation

Table 4.1. Hardware and software configuration of our evaluation system

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Number of CPUs</td>
<td>2</td>
</tr>
<tr>
<td>CPU</td>
<td>Intel Xeon CPU E5-2650</td>
</tr>
<tr>
<td>Clock Frequency</td>
<td>2.80 GHz</td>
</tr>
<tr>
<td>Number of Cores</td>
<td>16</td>
</tr>
<tr>
<td>Number of Threads</td>
<td>32</td>
</tr>
<tr>
<td>RAM</td>
<td>128 GB</td>
</tr>
<tr>
<td>OS</td>
<td>Debian GNU/Linux 8 (jessie)</td>
</tr>
<tr>
<td>Kernel Version</td>
<td>3.16.0 – 4</td>
</tr>
<tr>
<td>JVM Version</td>
<td>Java HotSpot Server VM 25.102-b14</td>
</tr>
<tr>
<td>Java Runtime Version</td>
<td>1.8.0_102-b14</td>
</tr>
</tbody>
</table>

Firstly, we demonstrate that the analysis basically processes monitoring records and calculates anomaly scores for them. In addition, we show that the visualization displays the analyzed data to demonstrate its correct functioning.

4.2.1 Methodology and Test Scenarios

We developed four scenarios that simulate typical execution behavior of software. Figure 4.1 shows how the single scenarios perform in the course of time and, in the following, we describe them in detail.

In each of the scenarios, a method is called every 50 to 150 milliseconds. For the analysis, we use a sliding window with a capacity of 10 seconds and a normalization interval with a duration of 300 milliseconds. Thus, on the average 3 measurements are aggregated to one and the forecast step uses 33 records. We evaluate each scenario with all of our six forecasters. The MeanAggregator is used for normalization. Each scenario is executed for 41 seconds, so the analysis generates completely different sliding windows for at least four times. Table 4.2 lists all of the evaluation parameters.

Table 4.2. Parameters of the feasibility evaluation

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call time Function</td>
<td>$t(x) = 100 + \text{noise}_{50s}(x)$ $^a$</td>
</tr>
<tr>
<td>Duration Function</td>
<td>see scenario descriptions</td>
</tr>
<tr>
<td>Execution Duration</td>
<td>41 s</td>
</tr>
<tr>
<td>Sliding Window</td>
<td>10 s</td>
</tr>
<tr>
<td>Normalization Interval</td>
<td>300 ms</td>
</tr>
<tr>
<td>Forecaster</td>
<td>all available forecasters</td>
</tr>
<tr>
<td>Aggregator</td>
<td>MeanAggregator</td>
</tr>
</tbody>
</table>

$^a$s depending on the scenario
4. Evaluation

In the following section we describe the single scenarios. In each of the following mathematic functions, \( s \) corresponds to a seed that is used to generate the pseudo-random values. The used seeds can be found in the implementation.

**S1: Constant with anomaly** In this scenario, the method is executed for constant 100 milliseconds. To make it more realistic we add a permanent noise, so the execution time ranges from 90 to 110 milliseconds. After 30 seconds, an anomaly is created by extending the response time by 100 milliseconds. A constant execution time is a typical scenario of an application that is equally loaded or scales adequately. The duration function of this scenario is described by:

\[
d(x) = 100 + \text{noise}_{10,s}(x) + \text{staticAnomaly}_{100,30000}(x)
\]

**S2: Linearly increasing with anomaly** The response times increase linearly from 100 to 500 milliseconds. Also this scenario has a constant noise and an anomaly after 30 seconds, where the execution time is 100 milliseconds higher. A linearly growth of response times is a realistic behavior for applications, when their data pool growths and they are not scaling.
4.2. Feasibility Evaluation

adequately, so operation on these data takes a longer time. The duration function of this scenario is described by:

\[ d(x) = \frac{400}{40,000} + 100 + \text{noise}_{10,s}(x) + \text{staticAnomaly}_{100,\{30000\}}(x) \]

**S3: Oscillating with anomaly** In this scenario the execution times are described by a sine curve, which oscillates between 100 and 300 milliseconds. As well as the previously described scenario this scenario has noise and an anomaly 30 seconds after start. This time the execution is 100 milliseconds shorter than described by the function. Thus, this scenario describes a seasonal execution behavior. For instance, most applications are requested in different degrees depending on the time of day or year. If such software systems do not scale, this could affect their response times. The duration function of this scenario is described by:

\[ d(x) = 100 \cdot \sin \left( \frac{2\pi \cdot x}{10,000} \right) + 200 + \text{noise}_{10,s}(x) + \text{staticAnomaly}_{-100,\{30000\}}(x) \]

**S4: Exponential increasing** The execution times increase exponentially so that primary in the last quarter a sharp rise can be observed. This scenario has the same noise as the others. Since no static anomaly occurs from one moment to another, it is probably possible that most forecast algorithms do not recognize this as an anomaly, because they assess it as normal behavior. The duration function of this scenario is described by:

\[ d(x) = e^{\frac{x}{40,000}} + 100 + \text{noise}_{10,s}(x) \]

4.2.2 Results and Discussion

In the following, we present and discuss the results of our evaluation scenario S3 Oscillating with anomaly. We chose it since its behavior is more complex and the forecast algorithms show different behavior. In the appendices (Section A.3) we visualize the execution times and the anomaly scores for each scenario and also show a more detailed view on the generated anomalies.

**Feasibility of the Analysis**

Figure 4.2 shows the course of the anomaly score for different forecast algorithms. The vertical axis represents the milliseconds since the evaluation started and the horizontal axis shows the calculated anomaly score.

Regardless of the chosen forecast algorithm, the forecasters continuously produce an anomaly score, however, the calculated scores considerably differ in part. The forecasters can be divided into two groups: The first group consists of the linear weighted, logarithmic
4. Evaluation

Figure 4.2. Anomaly scores of the scenario S3 *Oscillating with anomaly* in the course of time

weighted, and mean forecaster. It synchronously oscillates with the execution time of the evaluation scenario. The second group, which consists of the exponential weighted, regression, and ARIMA forecaster, produces relatively stable anomaly scores. The scores of all algorithms severely decrease after 30 seconds when the anomaly is simulated. Some algorithms do not produce an anomaly score at the beginning of the analysis and, by using the ARIMA forecaster, outliers periodically arises.

The first group recognizes seasonality insufficiently, while the second group subtracts oscillations successfully. However, the ARIMA forecaster reveals problems, if the investigated executions times turn from an increase to decrease. All algorithms detect the simulated anomaly. The peculiar scores in the beginning are explained by the forecasters’ characteristics, since some of them need a certain number of values to make a (sensible) prediction. Since an anomaly score is continuously produced, we conclude that the analysis component functions as expected.

Feasibility of the Visualization

After we demonstrated that the anomaly detection produces sensible results for all implemented forecast algorithms, we evaluated the visualization’s feasibility. Since the visualization component is designed to be independent from the forecast algorithms, we present the results for only one, namely the regression, forecaster.

Figure 4.3 shows a screenshot of the visualization after the evaluation scenario was accomplished. In the first chart, the orange curve represents the course of the execution times, while the blue curve draws the predicted value for each point in time. The second chart shows the associated calculated anomaly score. 30 seconds after the start, an indicator for a detected anomaly is displayed.

The course of the execution times corresponds to the generated behavior and the
4.2. Feasibility Evaluation

Figure 4.3. Screenshot of the visualization component for the scenario S3 Oscillating with anomaly by using the regression forecaster.

The course of the anomaly scores corresponds to the values that are calculated by the analysis component. In the visualization component we chose the thresholds $0.3$ and $0.3$. This means, that an outliers is recognizable and displayed as an anomaly if it is $30\%$ higher or lower than its prediction. By using these values, the generated abnormal behavior after $30$ seconds was successfully detected as an anomaly since the anomaly score is ca. $0.4$. We conclude that the visualization works as required.

4.2.3 Threats to Validity

**Internal** We do not see any internal threats to validity in the feasibility evaluation.

**External** We only evaluated the forecast algorithms that we have implemented. Our approach is designed to be also used with other algorithms, as they are used, for example, in OPAD. To increase the external validity, we also have to evaluate these algorithms.

In our evaluation, we only used one configuration for the hardware and software environment. An execution on different systems would increase the external validity.
4. Evaluation

4.3 Scalability Evaluation

The process of anomaly detection takes a certain time and requires a chronological treatment of the monitoring data. Thus, the analysis component cannot process any amount of them. If the monitoring part forwards too many measurements within a too short time interval to the analysis, it cannot analyze all measurements and thus appends the unprocessed ones to a queue. A temporary peak is not a problem, since this queue can be processed later. However, for the long term a proper functioning can only be achieved, if only the amount of data arrives that can be processed.

For this reason, we will figure out with which distances measurements can occur, so that the analysis can still process them. This limit depends on multiple parameters:

*Sliding window* The size of the sliding window effects the number of values that are used in the forecast step.

*Normalization interval* The normalization interval effects the number of values that are aggregated in the normalization step and thus the number of values that are used in the forecast step also.

*Method call frequency* The frequency with which methods are called effects the number of values in each normalization interval.

*Forecaster and aggregator* Algorithms for forecasting and aggregation can differ in complexity and thus have various computing times.

4.3.1 Methodology and Test Scenarios

To determine the minimal distances between operation calls we measure the time the anomaly detection stage needs. It is not possible to start the monitoring in an earlier phase of the Pipe-and-Filter configuration, since the record reconstruction has to be done beforehand. Consequently, the actual processing time is slightly higher. As mentioned, the limit depends on several parameters so we set up various scenarios. For each of them, we process several records, measure their processing times, and calculate an average of the times. Subsequently, we compare the average processing time with the distance between operation calls. If the processing time is less than the call distance, this configuration is suitable. Otherwise, a queue will be formed as described and, eventually, the monitoring will be aborted.

Since TeeTime does not provide any runtime performance measurements, we modified the analysis configuration as shown in Figure 4.4. The *Stopwatch* is a stage that measures the elapsed time from a start signal sent by the Record Converter stage to a stop signal that is sent by the Anomaly Detection stage after it finished its analysis. Afterwards, the Stopwatch sends the elapsed time to the *Execution Times File Writer* that writes it to a file.

To receive an impression of how long the analysis takes we did some pretests and then stated various evaluation scenarios. It is likely that the choice whether R is used or not
4.3. Scalability Evaluation

Figure 4.4. Modified TeeTime configuration of the analysis component to measure the processing time of the anomaly detection stage

makes a great difference since the network transport probably impacts the performance. Therefore, we test our scenarios with both a Java forecaster, which is the regression forecaster, and an R forecaster, which is the ARIMA forecaster. As most of our aggregation algorithms probably have a similar, namely linear, runtime, we confine our evaluations to one algorithm. Table 4.3 shows the chosen values for each parameter. This leads to 588 scenarios that we have evaluated.

Table 4.3. Parameters for our scalability evaluation. We evaluated each combination of these parameters.

<table>
<thead>
<tr>
<th>Call Distance</th>
<th>Sliding Window</th>
<th>Normalization Interval</th>
<th>Forecaster</th>
<th>Aggregator</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ms</td>
<td>10,000 ms</td>
<td>10 ms</td>
<td>ARIMA</td>
<td>Mean</td>
</tr>
<tr>
<td>5 ms</td>
<td>50,000 ms</td>
<td>20 ms</td>
<td>Regression</td>
<td></td>
</tr>
<tr>
<td>10 ms</td>
<td>100,000 ms</td>
<td>100 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50 ms</td>
<td>150,000 ms</td>
<td>200 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>100 ms</td>
<td>200,000 ms</td>
<td>500 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>150 ms</td>
<td>400,000 ms</td>
<td>1,000 ms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>200 ms</td>
<td>2,000 ms</td>
<td>2,000 ms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To evaluate these scenarios we used our previously introduced evaluation tool with the call time function $t(x) = \Delta_c$, where $\Delta_c$ is the call distance, and the duration function $d(x) = 1$. Since the execution time of operation calls is not relevant for the scalability, we set it as 1 to generate as few as possible overhead of sleeping threads. We execute each scenario for a duration that is twice as large as the sliding window. The first half of this duration serves for filling the sliding window time series and also as warm-up phase of the JVM, while in the second half the actual measurements are collected. We measure the processing time for each monitoring record and calculate the average of them. To minimize the random error we execute all scenarios five times.

After the execution of the scenario is finished, we have to consider the amount of collected measurements. If less than one sliding window is filled, we do not have meaningful data, because the first records after the analysis starts are processed faster. Consequently, we also do not use the values of the first sliding window for the average calculation.
4. Evaluation

Table 4.4. Exemplary results of the scalability evaluation

<table>
<thead>
<tr>
<th>$\Delta_{call}$</th>
<th>$T_{f\text{cst}}$</th>
<th>$T_{\text{norm}}$</th>
<th>Forecaster</th>
<th>#$_{\text{norm}}$</th>
<th>#$_{f\text{cst}}$</th>
<th>meanf.</th>
<th>$t_{\text{proc}}$</th>
<th>success</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 ms</td>
<td>10 s</td>
<td>20 ms</td>
<td>Regression</td>
<td>10</td>
<td>500</td>
<td>false</td>
<td>3.0 ms</td>
<td>false</td>
</tr>
<tr>
<td>2 ms</td>
<td>10 s</td>
<td>1000 ms</td>
<td>Regression</td>
<td>500</td>
<td>10</td>
<td>false</td>
<td>3.1 ms</td>
<td>false</td>
</tr>
<tr>
<td>2 ms</td>
<td>100 s</td>
<td>20 ms</td>
<td>Regression</td>
<td>10</td>
<td>5000</td>
<td>false</td>
<td>3.0 ms</td>
<td>false</td>
</tr>
<tr>
<td>2 ms</td>
<td>100 s</td>
<td>1000 ms</td>
<td>Regression</td>
<td>500</td>
<td>100</td>
<td>false</td>
<td>3.0 ms</td>
<td>false</td>
</tr>
<tr>
<td>2 ms</td>
<td>400 s</td>
<td>20 ms</td>
<td>Regression</td>
<td>10</td>
<td>20000</td>
<td>false</td>
<td>2.7 ms</td>
<td>false</td>
</tr>
<tr>
<td>2 ms</td>
<td>400 s</td>
<td>1000 ms</td>
<td>Regression</td>
<td>500</td>
<td>400</td>
<td>false</td>
<td>2.8 ms</td>
<td>false</td>
</tr>
<tr>
<td>5 ms</td>
<td>10 s</td>
<td>20 ms</td>
<td>Regression</td>
<td>4</td>
<td>500</td>
<td>false</td>
<td>1.9 ms</td>
<td>true</td>
</tr>
<tr>
<td>5 ms</td>
<td>10 s</td>
<td>1000 ms</td>
<td>Regression</td>
<td>200</td>
<td>10</td>
<td>true</td>
<td>1.6 ms</td>
<td>true</td>
</tr>
<tr>
<td>5 ms</td>
<td>100 s</td>
<td>20 ms</td>
<td>Regression</td>
<td>4</td>
<td>5000</td>
<td>true</td>
<td>2.2 ms</td>
<td>true</td>
</tr>
<tr>
<td>5 ms</td>
<td>100 s</td>
<td>1000 ms</td>
<td>Regression</td>
<td>200</td>
<td>100</td>
<td>false</td>
<td>2.4 ms</td>
<td>true</td>
</tr>
<tr>
<td>5 ms</td>
<td>400 s</td>
<td>20 ms</td>
<td>Regression</td>
<td>4</td>
<td>20000</td>
<td>true</td>
<td>4.4 ms</td>
<td>true</td>
</tr>
<tr>
<td>5 ms</td>
<td>400 s</td>
<td>1000 ms</td>
<td>Regression</td>
<td>200</td>
<td>400</td>
<td>true</td>
<td>4.1 ms</td>
<td>true</td>
</tr>
<tr>
<td>50 ms</td>
<td>10 s</td>
<td>200 ms</td>
<td>ARIMA</td>
<td>4</td>
<td>50</td>
<td>true</td>
<td>89.1 ms</td>
<td>false</td>
</tr>
<tr>
<td>50 ms</td>
<td>10 s</td>
<td>500 ms</td>
<td>ARIMA</td>
<td>10</td>
<td>20</td>
<td>true</td>
<td>92.0 ms</td>
<td>false</td>
</tr>
<tr>
<td>50 ms</td>
<td>100 s</td>
<td>200 ms</td>
<td>ARIMA</td>
<td>4</td>
<td>500</td>
<td>false</td>
<td>118.5 ms</td>
<td>false</td>
</tr>
<tr>
<td>50 ms</td>
<td>100 s</td>
<td>500 ms</td>
<td>ARIMA</td>
<td>10</td>
<td>200</td>
<td>false</td>
<td>140.0 ms</td>
<td>false</td>
</tr>
<tr>
<td>100 ms</td>
<td>10 s</td>
<td>200 ms</td>
<td>ARIMA</td>
<td>2</td>
<td>50</td>
<td>true</td>
<td>103.4 ms</td>
<td>false</td>
</tr>
<tr>
<td>100 ms</td>
<td>10 s</td>
<td>500 ms</td>
<td>ARIMA</td>
<td>5</td>
<td>20</td>
<td>true</td>
<td>92.1 ms</td>
<td>true</td>
</tr>
<tr>
<td>100 ms</td>
<td>100 s</td>
<td>200 ms</td>
<td>ARIMA</td>
<td>2</td>
<td>500</td>
<td>true</td>
<td>106.6 ms</td>
<td>false</td>
</tr>
<tr>
<td>100 ms</td>
<td>100 s</td>
<td>500 ms</td>
<td>ARIMA</td>
<td>5</td>
<td>200</td>
<td>true</td>
<td>86.4 ms</td>
<td>true</td>
</tr>
<tr>
<td>150 ms</td>
<td>10 s</td>
<td>200 ms</td>
<td>ARIMA</td>
<td>1.3</td>
<td>50</td>
<td>true</td>
<td>130.7 ms</td>
<td>true</td>
</tr>
<tr>
<td>150 ms</td>
<td>10 s</td>
<td>500 ms</td>
<td>ARIMA</td>
<td>3.3</td>
<td>20</td>
<td>true</td>
<td>104.7 ms</td>
<td>true</td>
</tr>
<tr>
<td>150 ms</td>
<td>100 s</td>
<td>200 ms</td>
<td>ARIMA</td>
<td>1.3</td>
<td>500</td>
<td>true</td>
<td>135.6 ms</td>
<td>true</td>
</tr>
<tr>
<td>150 ms</td>
<td>100 s</td>
<td>500 ms</td>
<td>ARIMA</td>
<td>3.3</td>
<td>200</td>
<td>true</td>
<td>95.2 ms</td>
<td>true</td>
</tr>
</tbody>
</table>

4.3.2 Results and Discussion

Table 4.3 shows few selected results. The first four columns specify the configuration of a scenario. $\Delta_{call}$ corresponds to the call distance, $T_{f\text{cst}}$ is the sliding window, and $T_{\text{norm}}$ the normalization interval. The next two columns show the number of values that are used in the normalization step ($\#_{\text{norm}}$) and in the forecasting step ($\#_{f\text{cst}}$) according to this configuration. The column meanf. indicates if meaningful data sets were generated for this scenario. In concrete terms this means, whether for all of the five executions enough calls could be generated to fill at least one sliding window, or whether instead at least one was prematurely terminated.

Finally, $t_{\text{proc}}$ shows the average processing time for this configuration and success indicates whether this scenario is executable in the long-term by testing if $t_{\text{proc}}$ is less than $\Delta_{call}$. The complete table containing all evaluation results is available via the following Git repository: https://build.se.informatik.uni-kiel.de/stu114708/bsc-evaluation-results

One result of this evaluation is that the processing times are heavily dependent on the
applied configuration and a universal conclusion about the call frequency limits cannot be made.

However, as a general rule, the ARIMA forecaster is significantly slower than the regression forecaster. While the regression algorithm has a computation time of 1.60 ms in the fastest case, ARIMA needs at least 49 ms. In the slowest case the regression forecaster needs 4.37 ms, whereas ARIMA needs 220 ms to finish the analysis execution. For one thing, this could reside in the fact that the used network connection decreases the performance. Another reason could be that the ARIMA algorithm is much more complex than the regression algorithm, especially, the automatic creation of the ARIMA model.

Furthermore, there is a wide dispersion in the single execution times of scenarios that use the ARIMA forecaster. The reason for this may be explained by the fact that all services, i.e., the Docker containers, communicate over the same network interface and, therefore, this could lead to high latencies.

Increasing the sliding window with same call distance and aggregation duration leads to higher processing times for both the ARIMA and the regression forecaster. This behavior is in line with our expectations since more values are used to calculate the forecast. If the call distance and the sliding window are fixed, increasing the normalization interval results in shorter processing times for the ARIMA forecaster. By using the regression forecaster, the processing times decrease only slightly. This would indicate that the number of values is not significant for the regression forecaster.

When using the regression forecaster, a call distance of 2 ms is always too small, even when the actual processing time is lower (see column meanf in Table 4.3). We observed that the buffer size of the monitoring-analysis-connection reaches its limit after a short time, so the monitoring part aborts the sending of data. This might be due to the fact that in the beginning a high amount of data is generated before the analysis has even started its processing. A possible solution could be enhancing the buffer size.

For nearly all configurations, the distance between method calls could be between 2 ms and 5 ms. This would be suitable values since the entire analysis needs a certain time and cannot be arbitrarily fast. However, if more operation calls occur, the approach has to be adopted (see Section 6.2).

When using the ARIMA forecaster, for all scenarios the limit is significantly higher. The call distance has to be between 50 ms and 100 ms for smaller sliding windows. For larger sliding windows, the call distance has to be even larger. This is very slow since, for example, a web application that is called more than 10 times per second could not be analyzed with our approach. The problem could be remedied by using Java forecasters (either the existing algorithms or implementation of new ones) or by adapting our approach as proposed in future work.

4.3.3 Threats to Validity

Internal The methodology of our evaluation is based on the assumption, that the anomaly detection procedure is the most time-consuming part of the analysis. To verify this, addition-
4. Evaluation

ally the receiving of monitoring records, their reconstruction, and also the asynchronous writing to the database have to be taken into account. However, this would require a fundamental revision of this evaluation.

**External** We evaluated our approach on a system with a large number of processing unit cores to enable an execution of the components as autonomous as possible. In particular, our evaluation tool and its monitoring should be separated from the analysis. However, since the individual components are executed on the same system, they influence each other nevertheless. The operating system does not guarantee a parallel execution and, furthermore, the components are still sharing system resources. Especially, the entire communication is handled via the same network. Thus, in future evaluations, the monitoring and the analysis part could be separated to individual hosts.

As our evaluation has shown, the results are heavily dependent on the configuration. It is likely that there are realistic deployment scenarios we did not have evaluated and, thus, we cannot make definite statements about them. For example, we did not consider very large time periods such as several hours.

Furthermore, we only considered one analysis branch. In principle, the processing time would be multiplied with the number of branches (as long as the branches are not executed in parallel). However, this may lead to side effects, which influence the processing time.
Chapter 5

Related Work

As already described in detail, ΘPAD [Bielefeld 2012] is an approach to detect performance anomalies online with Kieker. It is widely configurable, for example, in terms of used algorithms. In a case study, ΘPAD was evaluated in the environment of the social network XING, where it satisfactorily detects anomalies. However, to improve these results, Frotscher [2013] made several extensions to ΘPAD in the ΘPADx approach. This covers inter alia the transformation to a Pipe-and-Filter architecture. Also ΘPADx was evaluated in the XING environment and it turns out that it led to improvements in several aspects. Both ΘPAD and an ΘPADx are designed as plugins for the Kieker monitoring framework and are now maintained as part of the Kieker project. We adopted many parts from these approaches for our anomaly detection approach.

ExplorViz [Fittkau et al. 2016; Fittkau 2015] is an approach to visualize large software landscapes systems. Therefore, it provides a hierarchical perspective on the landscape level, on the one hand, and an application-level perspective following the 3D software city metaphor, on the other hand. To evaluate the extensibility of ExplorViz, Fittkau et al. [2014] developed a semi-automatic control center concept, which was integrated into ExplorViz by project-external developers. In this context Mannstedt [2015] developed an approach to detect and visualize anomalies on the architectural layer. The anomaly detection of this approach relies on ΘPAD.

Marwede et al. [2009] presented an approach to detect abnormal behavior of architecture components and not just on calls of single methods. Therefore, this, RanCorr called, approach, computes anomaly scores on the operation, component, and deployment context level. For this purpose, it correlates the anomaly scores for operation executions based on architectural dependency graphs. RanCorr also visualizes the diagnosis results in architectural diagrams.

Herbst et al. [2014] evaluated various time series forecasting methods. In this survey the authors describe the requirements and objectives of single methods, pointed out their strengths, weaknesses and computation effort, and also highlighted their optimal deployment scenario. Moreover, they presented a self-adaptive approach that selects suitable forecasting methods for a given context. The investigated forecasting methods have been implemented in ΘPAD.

Also, there are some industrial approaches to detect performance anomalies. One them is EGADS [Laptev et al. 2015], which is used at Yahoo. It only depends on Java and pursues a similar concept as we do. Among other algorithms, it supports ARIMA- and
5. Related Work

regression-based forecasters. Another approach is presented by Vallis et al. [2014], which is used at Twitter. A comparison of both approaches can be found by Ettl [2015].
Chapter 6

Conclusions and Future Work

6.1 Conclusions

In the early phase of this work we defined four goals that we wanted to reach. As described in goal G3 we developed a visualization of performance anomalies. It presents the execution times of application functions in the course of time and highlights identified anomalies. The visualization is embedded in a single page application and designed to provide a comfortable user-experience. Thus, it is continuously and automatically updated, so users (e.g., system administrators) can observe the execution behavior of their software in real-time. Furthermore, it is interactive that means users can dynamically increase or decrease the displayed time window or shift it to the past to investigate former behavior.

We have extended the $\Theta$PAD approach. As one aspect, we transformed its monolithic architecture into a design following the microservice pattern. By means of Docker its deployment is less laborious. As described in goal G1 the stepwise anomaly detection procedure is now based on TeeTime. The real-time visualization requires that the monitoring data is processed immediately and are not aggregated previously. In this regard, we adopted $\Theta$PAD, but, as our evaluation (goal G4) has revealed, this led to performance issues for the R-forecast algorithms. Thus, further studies have to be made to examine how both can be united. One possible concept is proposed in Section 6.2. In our goal G2 we intended to implement multiple forecast algorithms in native Java to achieve forecast results faster and more simple. Our implementation provides Java-based algorithms, which use the average, weighted averages, and regression. Moreover, we provide an ARIMA forecaster as an example of an R-based algorithm. In the appendices (Section A.2), a table compares our approach with $\Theta$PAD concerning various criteria.

We published all implementation as open source. Most of them are available on GitHub\(^1\) and Docker Hub\(^2\). Since our implementation consists of many single projects, we provide a table in the appendices (Section A.1) that shows the locations for each component, library, or tool.

---

\(^1\)https://github.com/SoerenHenning
\(^2\)https://hub.docker.com/u/soerenhenning
6. Conclusions and Future Work

6.2 Future Work

Our scalability evaluation pointed out that monitoring measurements can only occur with limited frequency to be processed in time. For many realistic application scenarios the limits we detected are too high. Therefore, future works could research how a faster processing of measurements can be achieved. An option would be to use the ΘPAD approach again, which firstly aggregates the data for a pre-defined interval and analyses this aggregated measurement only after this interval is full. However, this leads to the major drawback that a real-time visualization would no longer be possible. Instead, a new value could only be displayed once the normalization interval is full and, moreover, only aggregated values would be displayed and not the original ones. Thus, we propose to cache the time series operations, i.e., normalization and forecasting, and recalculate them asynchronously. Hence, the further analysis steps do not have to wait for the forecast, but can use the last, already available value for calculating the anomaly score. While these would not be results that perfectly fit to the corresponding measurement, for many application scenarios this would not make any difference, because most forecast algorithms generate a smoothing anyway. An additional benefit of encapsulating the time series operations is that thus completely different types of forecasting are enabled such as machine learning.

Another option to speed up the analysis is to parallelize it or even execute it distributed on multiple hosts. TeeTime already provides the opportunity to execute stages in parallel or concurrently. A distribution of TeeTime configurations is currently under research. In particular, the individual analysis branches could be executed simultaneously in this manner.

In future works the database component could be further developed. Cassandra provides many features for storing large datasets, especially, on distributed nodes. Thus, it could be investigated how we could benefit from such features. Another suggestion is that Cassandra pushes changes to a read-only database (e.g., Elasticsearch [Elasticsearch BV 2016]) from which potentially many visualizations can read data. Moreover, it can be examined whether time series databases [Lautenschlager et al. 2015] are an alternative for the storage of measurements.

Our approach uses Java property file to configure the analysis, which the user has to create. To enhance the usability of our approach a REST interface can be developed. Therefore, a graphical user interface can be created through which one can dynamically adjust what is monitored and which analysis parameters should be used.

Furthermore, our approach could be integrated as a Self-contained System into a Kieker-based application, which provides a monitoring information dashboard for software systems.
Bibliography


Bibliography


Bibliography


Bibliography


50


Appendix A

Appendices

A.1 List of Implementations

Docker Compose Configuration  KiekPAD-Compose
  Source code: https://build.se.informatik.uni-kiel.de/stu114708/KiekPAD-Compose

Analysis Component  KiekPAD-Analysis
  Source code: https://github.com/SoerenHenning/KiekPAD-Analysis
  Docker image: https://hub.docker.com/r/soerenhenning/kiekpad-analysis

Visualization Provider Component and Visualization Component  KiekPAD-Viz-Provider
  Source code: https://github.com/SoerenHenning/KiekPAD-Viz-Provider
  Docker image: https://hub.docker.com/r/soerenhenning/kiekpad-viz-provider

Docker R-forecast  Docker RServe Forecast
  Source code: https://github.com/SoerenHenning/docker-rserve-forecast
  Docker image: https://hub.docker.com/r/soerenhenning/rserve-forecast

Anomaly Detection TeeTime Library  TeeAD
  Source code: https://github.com/SoerenHenning/TeeAD

Time Series and Anomaly Visualization Library  Anomaliz.js
  Source code: https://github.com/SoerenHenning/Anomaliz.js

Evaluation Tool  WatchMe!
  Source code: https://build.se.informatik.uni-kiel.de/stu114708/watch-me
### A.2 Comparison to ΘPAD

<table>
<thead>
<tr>
<th></th>
<th>ΘPAD</th>
<th>Our Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>Architecture</td>
<td>monolithic</td>
<td>microservices</td>
</tr>
<tr>
<td>Anomaly scores</td>
<td>bounded to $[0,1]$, no</td>
<td>unbounded range, proportional scaling</td>
</tr>
<tr>
<td>calculator</td>
<td>proportional scaling</td>
<td></td>
</tr>
<tr>
<td>Monitoring record</td>
<td>after filled interval</td>
<td>immediately</td>
</tr>
<tr>
<td>processing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multiple time series</td>
<td>separation in every single stage</td>
<td>separation to single branches</td>
</tr>
<tr>
<td>Pipe-and-Filter</td>
<td>Kieker’s internal one</td>
<td>TeeTime</td>
</tr>
<tr>
<td>framework</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of implemented forecasters</td>
<td>1 Java-based, 8 R-based$^a$</td>
<td>5 Java-based, 1 R-based</td>
</tr>
<tr>
<td>Database</td>
<td>MongoDB</td>
<td>Cassandra</td>
</tr>
</tbody>
</table>

$^a$most of them introduced by Herbst et al. [2014]
A.3. Results of the Feasibility Evaluation

A.3 Results of the Feasibility Evaluation

Figure A.1. Feasibility evaluation scenario S1: Constant with anomaly
Figure A.2. Feasibility evaluation scenario S2: Linearly increasing with anomaly
Figure A.3. Feasibility evaluation scenario S3: Oscillating with anomaly
A. Appendices

Figure A.4. Feasibility evaluation scenario S4: Exponential increasing