Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Development Project

Online Performance Problem Detection, Diagnosis, and Visualization with Kieker

T. Düllmann, A. Eberlein, C. Endres, M. Fetzer, M. Fischer, C. Gregorian, K. Képes, Y. Noller, D. Olp, T. Rudolph, A. Scherer, M. Scholz

Course of Study: Software Engineering

Examiner: Prof. Dr. Lars Grunske

Supervisor: Dipl.-Inform. André van Hoorn

Commenced: 2013/11/18

Completed: 2014/03/31

CR-Classification: H.3.4, H.5.2

Abstract

With increasingly large systems Online Performance Monitoring becomes more and more a necessity to find, predict, and recover from failures. The Kieker monitoring tool enables the monitoring and analysis of applications. It allows to gather live data about the systems utilization like RAM-load, Swap-load, CPU-load as well as the latency of executed operations and their qualified name. ΘPADx provides means to detect anomalous behaviour and RanCorr allows the correlation of anomalies to identifiy the root cause of an anomaly. This project implements the RanCorr approach and extends the ΘPAD implementation with new forecast algorithms. Also the Kieker-WebGUI is extended to visualize the architecture, discovered by RanCorr, and other metrics by using dynamic diagrams. Additionally, an automated test framework is introduced that enables data generation and evaluation of the implemented forecasting and anomaly detection approach.

Contents

Chapter 1

Introduction

1.1 Motivation and Goals

Our motivation for this project lies within not only expanding the functionality of Kieker, but also in the visualization of data. Huge amounts of data are of little use if they cannot be seen and rapidly evaluated by a human. Therefore it is important to prepare our data to allow ease of access and analysis for the GUI, as well as formatting said GUI to display the most vital information in an easy to understand manner. As mentioned, we also wish to expand the functionality of Kieker by implementing a new plugin to allow a better live analysis of running systems that allows not only detecting errors, but also delivering accurate data for solving issues.

- Goal 1: Expand Kieker Functionality
  - Goal 1a: Extend ΘPADx implementation by adding a Self-Tuning Algorithm.
    While ΘPAD is functional, it lacks the ability to change its parameters at runtime, an important step for implementing adaptability in any algorithm. It is necessary to implement both the ability to allow the values to be changed as well as the learning algorithms that control the variables containing those values.
  - Goal 1b: Create Plugin using the RanCorr algorithm provided.
    In order to further understanding of the data created by Kieker, RanCorr will be implemented leading to a correlation between anomalies and the structures in which those anomalies occur. The root cause analysis is a useful tool for system analysis, cutting search times by pointing out culprits in case of errors or slowdowns.
  - Goal 1c: Validate both ΘPADx and RanCorr outputs. Simply implementing self-tuning ΘPADx and RanCorr is not sufficient. In order to verify the changes it is necessary to take measurements considering the actual accuracy of the changes versus the results they delivered beforehand.
- Goal 2: Create Frontend to display gathered data in a useful manner
  - Goal 2a: Get rid of the Pipes and Filter view to set up the monitoring and create a simpler way for the specific purpose addressed in this project while keeping all possibilities to configure the backend.
  - Goal 2b: Show the architecture of the monitored system and offer the possibility to examine the components of the system using a deep dive approach.

- Goal 2c: Offer the possibility to create dynamically new diagrams. This will replace the old cockpit view to show the data of the different metrics.

Christopher Gregorian and Martin Scholz

1.2 Document Structure

This document continues with Chapter 2 which presents the foundations gained in the seminar talks. The Chapter 3 describes the requirements for frontend, backend and the interface between them. Then the Chapter 4 shows how the project was organized and the technical support. Section 4.3 provides the task distribution across the team members. Afterwards Section 4.4 presents the different organizational roles which were defined in the project. Then Chapter 5 shows the overall architecture designed in this project. Chapter 6 describes the current state before the project started. The Chapter 7 continues with the description about the actual implementation work. Afterwards the Chapter 8 provides the validation description and interpretation. Finally Chapter 9 shows the possible future work which could be done in order to extend this project.

Yannic Noller

Chapter 2

Foundations and Technologies

2.1 ΘPAD and RanCorr

As described in chapter 1.1 this project aims to extend the ΘPAD approach among others by implementing the idea of RanCorr. Therefore it's necessary to understand the basics of anomaly detection and especially the ideas of ΘPAD and RanCorr.

2.1.1 Online Performance Anomaly Detection (ΘPAD)

The ΘPAD approach was introduced in the diploma thesis by Bielefeld [2012] and means *Online Performance Anomaly Detection*. Frotscher [2013] extended the ΘPAD approach in his master thesis and named it ΘPADx. In this document we still talk about the ΘPAD approach, but mean the latest version. ΘPAD detects anomalies in performance data based on discrete time series analysis. It's divided in five steps like shown in Figure 2.1 suitable for Kieker's Pipes-and-Filter architecture.

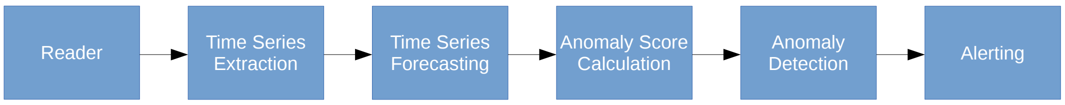


Figure 2.1. Analysis steps of ΘPAD approach based on Bielefeld [2012] consisting of: (1) reading of measurement data, (2) extraction of discrete time series, (3) forecasting of next observation based on historical data, (4) calculation of anomaly score based on difference between observations and foretold reference model, (5) detection of anomaly based on anomaly score threshold and (6) alerting in case of an anomaly.

The first step *Reader* reads the input data of ΘPAD which are the raw measurements of a performance metric like response time in continuous time series. The second step *Time Series Extraction* converts the continuous time series in a discrete one by aggregating the data in discrete time periods. For the aggregation a simple unweighted arithmetic mean function could be used. The third step *Time Series Forecasting* calculates a forecasting value for the next time period. ΘPAD supports multiple forecasting methods. The simplest one is called *Moving Average* which calculates the unweighted arithmetic mean of a sliding time window as forecasting value. More methods are *ARIMA* or *Single Exponential Smoothing (SES)*. The fourth step *Anomaly Score Calculation* uses the forecasting values of the previous step as reference model in order to compare it with the actual observations. The difference value is mapped to range between 0 and 1 and is called anomaly score. The fifth step *Anomaly Detection* uses a predefined anomaly threshold to detect an anomaly: if the anomaly score exceeds the threshold, the measured values will be considered as an anomaly. The last step *Alerting* can be used to notify e.g. an administrator about the detected anomaly. ΘPAD works fine for point anomalies and contextual anomalies, but for collective anomalies (see Figure 2.2) the reference model contains to much mutated values and the system generates false positive results. This can lead to frustrated users and bad user acceptance. Therefore Frotscher [2013] introduced with ΘPADx a new forecasting method called *Pattern Checking* which searches in case of collective anomalies a better reference model in the historical data. However, the ΘPAD approach provides only the detection of anomalies and not a root cause analysis of them. This needs a correlation of anomaly scores like described in the next section.

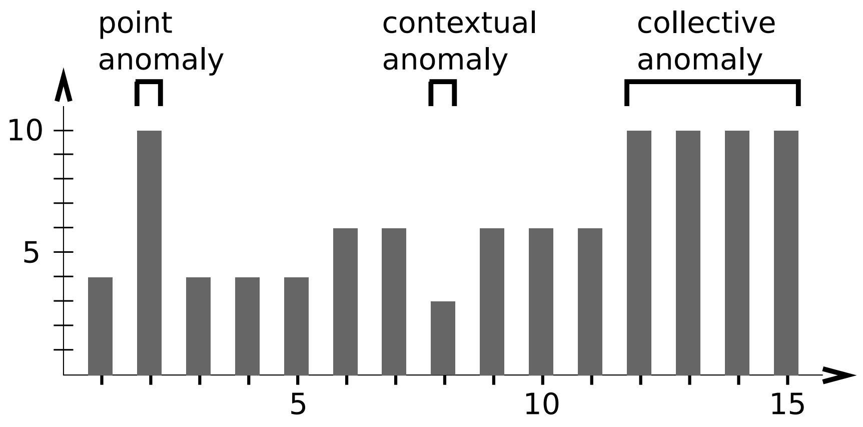
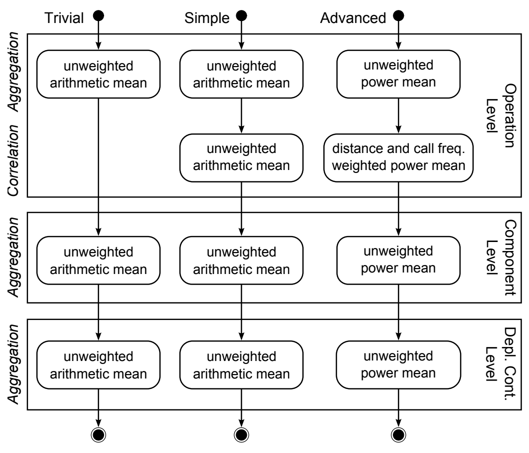


Figure 2.2. Overview about the three anomaly types: point, contextual and collective anomalies (based on Frotscher [2013]).

2.1.2 Anomaly Correlation: RanCorr

In order to localize the reason for an anomaly it's possible to correlate the calculated anomaly scores. This theory was investigated by Marwede et al. [2009] and the resulted approach is called *RanCorr*. RanCorr uses the anomaly scores, which are calculated by another system e.g. ΘPAD, and uses the calling dependencies between components to correlate the scores. RanCorr defines four activities: (1) model building, (2) aggregation of anomaly scores, (3) correlation to anomaly ratings and (4) visualization. The approach defines three algorithms: trivial, simple and advanced. They differ in how they aggregate and correlate the anomaly values (see Figure 2.3). The more complex the algorithm is, the more accurate are the results.



Trivial
Simple
Advanced
Aggregation
Correlation
unweighted arithmetic mean
unweighted arithmetic mean
unweighted power mean
Operation Level
unweighted arithmetic mean
distance and call freq. weighted power mean
Aggregation
unweighted arithmetic mean
unweighted arithmetic mean
unweighted power mean
Component Level
Aggregation
unweighted arithmetic mean
unweighted arithmetic mean
unweighted power mean
Depl. Cont. Level

Figure 2.3. Aggregation and correlation steps of RanCorr shown for every algorithm (trivial, simple and advanced) based on Marwede et al. [2009]. The first row shows the aggregation function for the local anomaly scores of an operation. In case of the trivial algorithm this aggregated anomaly score is already the anomaly ranking, but in case of the simple and the advanced algorithm there is an additional correlation step between the dependencies of the considered operation. Afterwards the anomaly ranking will be aggregated for all hierarchy levels.

Yannic Noller

2.2 CoCoME

In order to create real application data, we took several Common Component Modeling Example (CoCoME) [Herold et al., 2008] implementations into consideration. It is a real world application example used for many different purposes in research. CoCoME is a model of a supermarket trading system that consists of several modules (e.g. enterprise, inventory, store, cashdesk, bank) which interact with each other. CoCoME is modeled to be able to act as a distributed system, which would be useful for testing purposes.

To evaluate whether a CoCoME implementation could be useful for our testing and data generation purposes we obtained several implementations. We tried to use the CoCoME CoC [b] implementation to serve as data generation application which provided multiple Graphical User Interfaces (GUIs) to control the example (e.g. inventory lists, cashdesk process).

Unfortunately it was not possible to get the other implementations (CoCoME2 CoC [a], SOFA Shop Sof) to run. Due to tool or package requirements and quite small manuals we could not start them properly. This might also be caused by the specific implementation of the applications as different research groups use CoCoME for different purposes:

- monitoring and reverse engineering tools (e.g. Kieker Jung et al. [2013])
- Cloud Computing/Service Oriented Architecture (SOA) Jung et al. [2013]; Hasselbring et al. [2013]
- Model Driven Engineering
- etc.

The main problem was that the documentation of how to set the implementations up and how to use them was quite small. Therefor sometimes it was not clear how to use the applications correctly. Due to the relatively high effort that would have to be made to understand the workflows within those implementations we decided not to use CoCoME implementations for test data generation.

Thomas Düllmann

2.3 Kieker and Kieker WebGUI

2.3.1 Kieker

Kieker is an open source framework that is designed to monitor performance of large software application during run-time. The Kieker project is developed by by the University of Kiel. The source code of Kieker is provided by the Kieker homepage [Kie]. The framework can be used to show performance problems with no need to have a deeper understanding of the source code. The Kieker framework, shown in the picture below, is divided into a monitoring and an analysis part. With the monitoring part Kieker is able to collect different data during run-time like the response time of methods, calling dependencies or CPU data. These data can be analyzed with the analysis part of the framework. Because of the pipes and filter structure, Kieker is very flexible. That advantage can be used within the analysis part. Filters can be combined to get an analysis designed for different use cases. This architecture provides the possibility to create plugins to enhance Kieker.

ΘPADx is an important part of the Kiekeriki project, which is based on Kieker. So Kieker is the main dependency of Kiekeriki as well. For a deeper understanding of Kieker it is recommended to read the Kieker User Guide [Project, 2013].

Tobias Rudolph

2.3.2 WebGUI

The Kieker WebGUI is a web page which is able to visualize the results of Kieker and to configure the analysis of Kieker. It is able to manage different projects. Each of them has its own analysis view, which can manage the analysis of this project. The configuration of the analysis, can be done in the analysis editor. This visualizes the pipes and filter architecture of Kieker as a graph. Readers and filters are represented as nodes, the connections between them by edges. The nodes can have input- and output ports which are represented by arrows

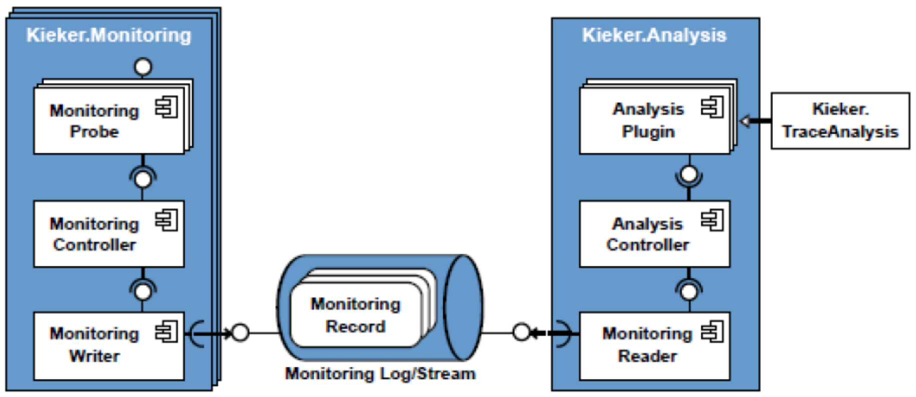


Figure 2.4. Overview of the architecture of Kieker

on the nodes. It is possible to create new edges between the readers and filters. In this way, new analyses can be created.

Another view is the cockpit view. In this page, several metrics are shown using different kinds of diagrams. The diagrams use the data gathered by the analysis. The cockpit view can be configured, too. It is possible to place different diagrams on the cockpit view, depending on which metrics are gathered.

The Kieker WebGUI uses PrimeFaces for displaying the content. It runs best on a jetty webserver. The WebGUI can be downloaded on the kieker homepage as executable binary file or as eclipse/netbeans project. Several example projects are included. See chapter X for the architecture of the WebGUI.

2.4 APM Tools

Application Performance Management(also kown as Application Performance Monitoring) crosses a wide range of IT disciplines in order to detect, prioritize and resolve performance and availability problems that affect business applications (Shields [2010]). A low overhead production and diagnosis capabilities are essential. Gartner defined the five dimensions end user experience, runtime application architecture, business transactions, deep dive component monitoring and analysis/reporting for evaluating the APM tool market. He revealed that in 2013, the best commercial vendor was AppDynamics. Some features of this tool which can be relevant to our project are described in the following. First, the deep dive approach into components is helpful to get from a high level view into a detailed view (e.g. on method level). This enables multiple troubleshooting mechanisms. Next, end user monitoring which analyzes

]

how end users perceive the actual performance is an approach which gains importance over the last years. However, it will not be implemented in this project. Defining health rules for servers or business transactions gives a good overall view of the monitored system. Therefore, similar features could be relevant for our project. Then, alert and respond features after an anomaly or an exception occurred are important in production mode, e.g. for sending an e-Mail to responsible persons. To limit the tasks for this project, this will not be part of the product backlog. Finally, business transactions are fundamental for APM tools as they reveal important information about the business impact of user requests. Detecting business transactions and showing them shall be part of the upcoming work. Showing live data in the GUI is feauture, all important APM tools are capable of.

Anton Scherer

2.5 Failure Diagnosis

Modern computer systems are getting larger and more complex over time due to frequent updates, repairs or through new components getting integrated and other components getting separated. To prevent system failures or limit the potential damage caused by system failures we try to predict if failures are about to happen in the near future and induce necessary steps to prevent them or recover fast. This section is a brief summary of Salfner et al. [2010].

2.5.1 Proactive Fault Management

The term ”Proactive Fault Management” describes a technique to cope with occuring failures. The main motivation is, that a system is made aware of upcoming failures and can then induce actions to prevent failures from happening or to prepare measures for a fast recovery. Proactive Fault Management consists of four basic steps:

1. **Online Failure Prediction** tries to identify situations that have a high probability that a failure is about to happen. The result of Online Failure Prediction can be binary decision or another measure. E.g. ”There is a 70% probability that a failure will occure within the next 5 minutes”.

2. **Diagnosis** is the task of finding out, what exactly is about to happen. It may be relevant to know where the failure is located and what exactly the failure is. E.g. ”That server will go offline because its database is flooded”.

3. **Action Scheduling** tries to determine what actions have to be executed to best resolve the current situation. The decision is based on the outcome of the Online Failure Prediction as well as the Diagnosis. Relevant factors in the decision process are the cost of action, confidence in the prediction, and effectiveness and complexity of the actions. E.g. a high cost action will only be executed when failure occurence is almost certain.

4. **Execution of Actions** is the last step in Proactive Fault Management.

2.5.2 Definitions

- A *failure* is "an event that occurs when the delivered service deviates from the correct service". A failure can be observed by any party that uses a system. There is no failure as long as the output is as specified.
- An *error* occurs when the current system state differs from the correct system state. An error may lead to a system failure.
- A *fault* is the cause of an error. Faults can be existing dormant in a system. Their activation causes an incorrect system state.
- Errors can be further classified in *undetected* and *detected* errors. Undetected errors become detected errors when an incorrect system state is identified.
- Undetected or detected errors may not only be the cause of failures but also of symptoms. A *symptom* is an out-of-norm behaviour of system parameters caused by errors.

2.5.3 Online Prediction

Online Prediction is the task of predicting the potential of failure occurrence for a time in the future (*lead time*) based on the current state of the system. The system state is assesed through monitoring within a window of certain length. Any prediction is only valid for a certain amount of time (*prediction period*).
Increasing the prediction period results in increased probability that a failure is predicted correctly. But if the increased prediction period is too long, the information becomes irrelevant due to its inaccuracy.
A leadtime that is larger than the systems reaction time to avoid a failure, is not very sensible. Therefore a *minimal warning time* is introduced. If the lead time is shorter than the minimal warning time, there is not sufficient time for the system to react to a upcoming failure.

Markus Fischer

2.6 Design of Performance Experiments

2.7 Trashing JPetStore

While Kieker currently collects several metrics (including but not limited to CPU usage, Memory and Latency), we are concentrating on the Latency of individual function calls. Data gathered via this metric is analyzed byΘPADx and RanCorr implementations for anomalies, which is then forwarded to the GUI.

Both our Front- and Back-end will allow automatic testing of their features. In the case of the Back-end, this will include Input Generation, Input Manipulation, Anomaly Insertion and automated evaluation of the results delivered. By creating automated testing tools, it is possible to support rapid development not only by our team, but by future users of our work.

In this section we give an introduction on JPetStore and some applications of Apache JMeter Plans. JPetStore was originally a release and demo by Sun, which was showcase of Java BestPractices and JavaEE. With the time is became also one reference application for JavaEE. With the time vendors like Mircosoft their own versions of the PetStore using the .NET Framework. As the requirements of both reference applications were similar, some performance comparisons between the Java and .NET worlds were made. Another version of JPetStore is available, MyBatis released their own version of the store. MyBatis is a framework that allows decoupling of database statements from the code, atop of this the MyBatis version is build using the Spring Framework, a Model-View-Controller framework with the same goal as Java EE and additionaly the Stripes framework is used for MVC Web Applications.

The general access and usage of the MyBatis JPetStore is based on a HTTP API consisting out of multiple resources.

- The catalog is the main resource for the pets in the store
  - `http://localhost:8080/jpetstore/actions/Catalog.action`
- The following queries for example allow requests for the pet information pages
  - `?viewCategory=&categoryId=FISH`, will show all pets that are fish in the store
  - `?viewProduct=&productId=FI-FW-01`, would should a undercategory of a kind of animals
  - `?viewItem=&itemId=EST-4`, itemId is used for a single entity of animal
- Adding and removing animals from the cart is done by the following URI's:
  - `/jpetstore/actions/Cart.action?removeItemFromCart=&cartItem=EST-6`
  - `/jpetstore/actions/Cart.action?addItemToCart=&workingItemId=EST-26`

To generate data with the JPetStore that represent anomalies Apache JMeter was used. JMeter is a tool which can be used for performance and load testing purposes. For that, JMeter allows to define threads in a Test Plan, where a single thread can have an assigned plan of steps it should execute. The steps can include SOAP, HTTP and FTP requests or database statements. The general concept was to verify whether JPetStore was suitable to use for anomaly checking with the metrics that had to be implemented by the development team. The results where enough for anomaly detection. With 100.000 requests on the same resource of the store, anomalies with deviatons of 5.400 milliseconds where possible, which is enough for testing purposes.

2.8 Tooling

Agile software development such as Scrum is facilitated by choosing and using the right tools. These allow programmers and project managers to work independently and transparently by utilizing a central maintenance.

The distribution of tasks to individual or multiple programmers for example, is done via an issue tracking system (ITS). Using ITS, all participants in the project can both manage tasks and plan the project schedule. This includes the sprint planning, the scheduling of milestones or the classification of the development process in developing steps. Likewise, working time and effort are documented.

Encountered errors must be recorded when testing the software components or using code parts of other software developers. In this way the risk of losing sight of the error in complex software drastically decreases. This is simplified if a Bug tracking system (BTS) is used, which ideally is closely linked to the ITS. The responsible developer may make observations in the BTS or request additional details that are needed to reproduce the error.

Issue tracking systems and bug tracking systems primarily serve the organization of the development project, but are not directly related to the program code. In order to facilitate concurrent and efficient work of several developers, a software version control system (VCS) can be used. A VCS manages the program code centrally and recognizes all the innovations and changes that the developers made.

Andreas Eberlein

2.8.1 Version Control System

A version control system (VCS) is a system that tracks all changes to a file or folder which is under version control. Developers can retrieve the current state of the software they are working on and submit their local changes to the version control system. This allows developers to work on the same project without bothering with the exchange of files and changes. As version control keeps track of changes, it is possible to revert back to older revisions of files and/or changes. To enable a proper collaboration between the developers, the usage of a VCS is inevitable.

Types of version control systems:

There are several version control systems available, of which the developers could chose from. Those systems can be divided into three main categories:

- **Local version control systems**: The easiest way of keeping several revisions of a file is, to manually create copies of the files the developer is working on (see Figure 2.5). All the changes and revisions are only available to the developer who created those revisions. This allows the developer to revert back to any state he (explicitly) saved.

- **Centralized version control systems**: Centralized version control keeps revisions (or changesets) at a central location (see Figure 2.6). Developers retrieve the current version and later on submit their changes. This allows several developers to work on the same files at the same time.

- **Distributed version control systems**: Distributed version control systems can be considered a hybrid between local and centralized versioning systems (see Figure 2.7). Developers retrieve the current state of the developed software, including all its revisions and changesets. They can work on this repository just like with a local version control system. Later on they can submit all their changsets (not only the updated files) to a central repository, which allows other developers to retrieve all their changes and files.

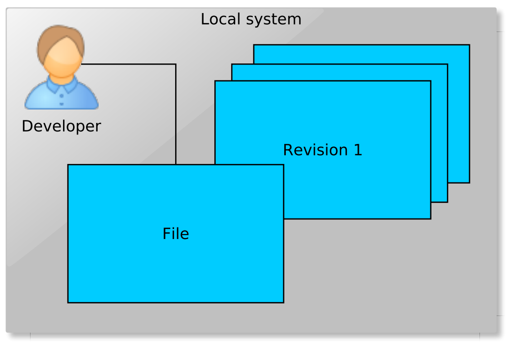


Figure 2.5. Local version control system

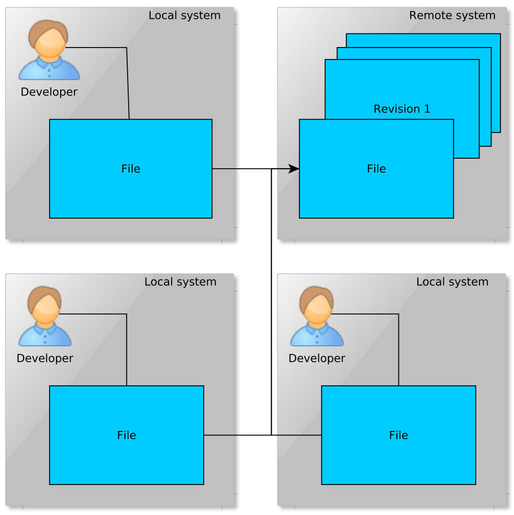


Figure 2.6. Centralized version control system

Matthias Fetzer

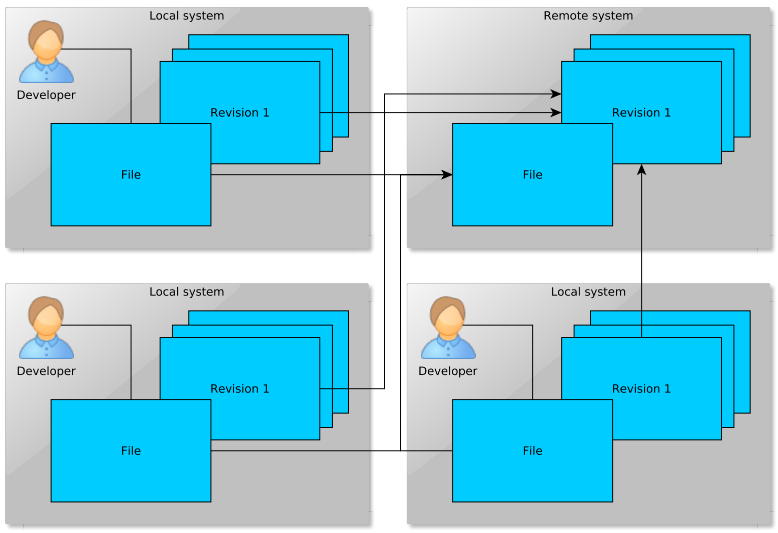


Figure 2.7. Distributed version control system

2.8.2 Continuous Integration

The systems mentioned in 2.8 allow the developers to utilize continous integration (CI). Figure 2.8 shows a general overview on how CI works. The main goal of CI is to continuously test and integrate the software during its development. This helps to ensure that a working version of the software always exists. The workflow with CI is as follows:

1. The developers upload their code to a (central) version control system.
2. The CI-Server retrieves the source code from the version control system, either periodically or triggered.
3. The CI-Server initiates the building of the retrieved source code.
4. The CI-Server notifies the developers in case of build failures, or any other critical events.
5. If the build was successful, the CI-Server deploys the software onto a test environment.

Matthias Fetzer

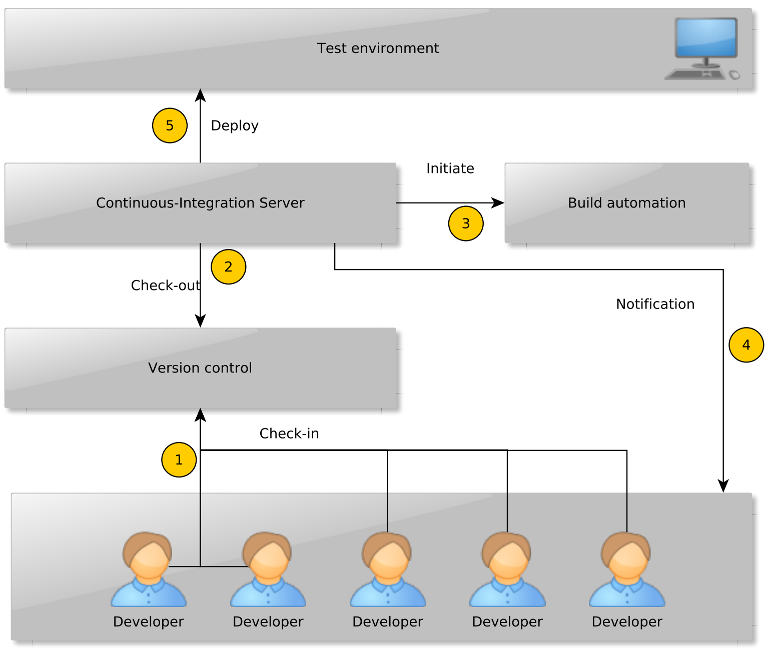


Figure 2.8. Sample build-automation setup.

Chapter 3

Requirements

The requirements for that project were identified by interviews between the two product owners and the supervisor. According to our development process (see chapter 4.2) we created a Product Backlog to report the requirements as user stories. The Product Backlog is divided in frontend, backend and interface between them. Frontend means the graphical user interface of our system, the backend means the analysis plug-ins and extensions in Kieker and the interface means the communication part between frontend and backend. Every user story owns an ID which is also mentioned in our issue tracker Redmine (see chapter 4.1), the priority and the number of the related Scrum Sprint (see chapter 4.2.1) in which the requirements was fullfilled.

Yannic Noller

3.1 Frontend Requirements

Table 3.1. Overview of the frontend requirements

ID	Title	Priority	Sprint
F1	Architecture overview	1	1
F2	Deep dive approach	1	2-4
F3	Diagram View	1	2-4
F4	Metrics Definition	2	3-4
F5	Configurations	4	-
F6	Displaying Business Transactions	5	-
F7	Showing Anomaly Detection of RanCorr	2	3
F8	Troubleshooting mechanisms	4	-
F9	Displaying live and historical data	3	3-4

F1 Architecture overview

Description: After opening a project, the user sees a rough overview layout of the web application according to the predefined GUI-Mockup (Figure 7.8). The page contains the architecture view where the components of the monitored system are displayed. We are required to check whether the graph component of the existing Kieker WebGUI is suitible for our purpose. The diagram view and the tab view which should not be implemented, must be considered when designing the layout. A sub-requirement by the supervisor is to use the existing Kieker WebGUI which must be reduced to a basic level.

Acceptance Criteria: A first runnable prototype of the web application should be the outcome of this requirement. The architecture view should display its first components on the highest level(servers).

***F2* Deep dive approach**

Description: Regarding the architecture graph which was built in F1, this component has to be extended, so the user can deep dive into multiple levels of the monitored system. The displayed components are connected to each other, illustrating the caller-callee-relationship. The connection to the transfer database has to be established for displaying real data. On the highest level, the user is able to see hosts. After selecting one specific host, his view is deep dived, so he sees all applications running on this host. The further level sequence is package (multiple package-levels possible), class and finally the operation-level. The user must have the opportunity to jump back to an arbitrary previous level.

Acceptance Criteria: The user can deep dive into any level in the architecture graph. The used icons for illustrating the components must be under a free license.

***F3* Diagram View**

Description: The user is able to add and delete diagrams dynamically. The user sees diagrams displaying data of the transfer database. For adding a new diagram some options must be available. At least, the following configurations must be selectable: The metric to show, the diagram type and live or historical data. Furthermore, diagrams must be changeable after they have been added to the view. Diagrams must be sticked to one specific level, e.g. diagrams for host level. Moreover, after the user left this view, he must see the diagrams he created when returning again. On every level default diagrams must be shown.

Acceptance Criteria: An automatical acceptance test (e.g. WebDriver) verifies if the displayed data are correct. Furthermore, all combinations of options must be checked for correctness.

***F4* Metrics Definition**

Description: When using the diagram view, some default metrics are available for the user. However, he must have the opportunity to define new metrics based on the raw data provided by the database.

Acceptance Criteria: If the user is able to define a new metric and he can choose this metric to add a new diagram which is displayed correctly, this requirement is fulfilled.

F5 **Configurations**

Description: RanCorr and ΘPADx which run in the backend, should be configurable at runtime. Therefore, a suitable configuration mask should be provided to the user. Moreover, another configuration mask must be added to change options of the GUI (e.g. styling information).

Acceptance Criteria: The connection to the database is established and changed configurations on the GUI lead to correct adjustments on the backend.

F6 **Displaying Business Transactions**

Description: Business Transactions can be displayed in an extra view where a list of all identified business transactions is displayed. The user is able to search for items in this list. This view shows the transaction in a subset of the architecture view. The edges of this graph are labelled with the average duration of each step of the transaction.

Acceptance Criteria: The business transaction view illustrates the correct components which are involved in a business transaction.

F7 **Showing Anomaly Detection of RanCorr**

Description: In addition to the metric of RanCorr (Component Anomaly Score) which can be illustrated by a diagram, this requirement demands to show the RanCorr results directly in the architecture graph by colouring the background of a system component if an anomaly was found within that component.

Acceptance Criteria: Components are coloured in an appropriate colour range when the database provides a higher anomaly score for one specific component than its threshold.

F8 **Troubleshooting mechanisms**

Description: If the user detects an anomaly within a component, he must be able to track the root cause of this anomaly, e.g. if an exception has caused the anomaly, this specific stacktrace must be shown to the user. Otherwise at least the latency diagram for the causing operation must be displayed.

Acceptance Criteria: The user is able to navigate to the anomaly origin of the shown anomalies.

F9 **Displaying live and historical data**

Description: Regarding diagrams, the user has the possiblity to add a new diagram illustrating data monitored in a past time range or displaying live data. In the second case, the polling time of diagrams should be configurable. The architecture graph should always work on live data. If a new component is instrumented by the backend and newly written into the database, the newly incoming component must also be displayed in the architecture graph.

Acceptance Criteria: The user is able to choose if live or historical data should be displayed in diagrams which then show the right time buckets. The architecture graph is dynamic including new components.

Anton Scherer

3.2 Backend

Table 3.2. Overview about the backend requirements

ID	**Title**	**Priority**	**Sprint**
B1	Anomaly Correlation with RanCorr	1	1-3
B2	ΘPAD self-tuning	2	1-4
B3	ΘPAD default configuration	2	-
B4	Experiment framework	2	1-4
B5	New ΘPAD forecasting algorithm based on ARIMA and GARCH	3	-
B6	New ΘPAD forecasting algorithm based on decision trees	3	2-4
B7	Business transactions	4	-
B8	Measure correlation	5	-
B9	Software Aging	5	-
B10	Exception Monitoring	4	-
B11	Metrics providing	1	3-4
B12	User detection	5	-

B1 Anomaly Correlation with RanCorr

Description: User can analyze the system with a root cause analysis employing architectural information on calling dependencies based on RanCorr (see chapter 2.1.2)

Acceptance Criteria: RanCorr is available as a Kieker plug-in. It's possible to run the RanCorr plug-in in the Kieker environment. The results of RanCorr are saved in a database and are available for the frontend. The calculation and the aggregation of ranking values is well tested.

B2 ΘPAD self-tuning

Description: ΘPAD shall be extended to be able to reconfigure itself.

Acceptance Criteria: It's possible to run ΘPAD with an self-tuning algorithm, which adjusts the configuration of ΘPAD at run-time. Therefore the filters of ΘPAD are now configurable and there is an algorithm which automatically adjusts the configurations. The execution of this new combination of filters is well tested.

B3 **ΘPAD default configuration**

Description: ΘPAD shall be extended by a default configuration which can be used by the user.

Acceptance Criteria: Default configuration for ΘPAD is identified, available and retrievable (especially for frontend).

B4 **Experiment framework**

Description: The analysis steps by ΘPAD and RanCorr shall be embedded in an experiment framework to set the input of test data, change configurations, check test results and provide appropriate test metrics.

Acceptance Criteria: There is an automatic input creation, execution, and output evaluation for ΘPAD and RanCorr. The experiment framework runs and works.

B5 **New ΘPAD forecasting algorithm based on ARIMA and GARCH**

Description: ΘPAD shall be extended with a new forecast algorithm based on the integration of ARIMA und GARCH models.

Acceptance Criteria: New forecast algorithm is available as extension of ΘPAD. New forecast algorithm can be used in ΘPAD and is well tested.

B6 **New ΘPAD forecasting algorithm based on decision trees**

Description: ΘPAD shall be extended with a new forecast approach which selects suitable forecast algorithms for a given context based on a decision tree and direct feedback cycles.

Acceptance Criteria: New forecast algorithm is available as extension of ΘPAD. New forecast algorithm can be used in ΘPAD and is well tested.

B7 **Business transactions**

Description: Business transactions can be identified by the sequence of called methods (dynamic identification).

Acceptance Criteria: Business Transaction Detection is available as Kieker plug-in. It's runnable, works and is well tested. Business Transactions are stored in a database and is available for the frontend.

B8 **Measure correlation**

Description: The analysis shall be extended by a step which correlates multiple measures (e.g. response times and workload).

Acceptance Criteria: The analysis is available as a Kieker plug-in. It's runnable, works and is well tested. Results are stored in a database and are available for the frontend.

B9 **Software Aging**

Description: The analysis shall be extended to detect software aging.

Acceptance Criteria: New anomaly detection algorithm is available as a Kieker plug-in. It's runnable, works and is well tested. Results are stored in a database and are available for the frontend.

B10 **Exception Monitoring**

Description: User can monitor various exceptions which occur in the monitored system.

Acceptance Criteria: Exceptions are monitored and stored in a database to provide them for the frontend. The Exception catching works and is well tested.

B11 **Metrics providing**

Description: User can access the monitored response times of operations, CPU load, memory usage (memory + swap), architecture, anomaly scores, anomaly rankings.

Acceptance Criteria: Response times of operations are monitored and stored in a database (accessible for the frontend). CPU load of all monitored machines are monitored and stored in a database (accessible for the frontend). Memory usage (memory + swap) of all monitored machines are monitored and stored in a database (accessible for the frontend). The architectures of all monitored machines are monitored and stored in a database (accessible for the frontend). Anomaly scores and anomaly rankings of all monitored machines are monitored and stored in a database (accessible for the frontend). The backend uses the provided exchange interface (see I1) to transmit the data. The data exchange mechanism is documented and well tested.

B12 **User detection**

Description: It's possible to detect which user is currently using the monitored system and which operation calls are caused by that user.

Acceptance Criteria: Users are detected during run-time. It will be detected when a new user accesses the system and what he is actually doing, i.e. which operation calls he causes. It will also be detected when an user isn't using the system anymore. A user should be identified uniquely. The detection mechanism is documented and well tested.

Yannic Noller

3.3 Interface

Table 3.3. Overview about the interface requirements

ID	Title	Priority	Sprint
I1	Data Exchange between Frontend and Backend	1	1-4
I2	Calling modalities from Frontend to Backend	2	3-4

I1 **Data Exchange between Frontend and Backend**

Description: It's possible to exchange data between frontend and backend site of Kiekeriki. Therefore two interfaces are necessary: the interface called by the backend to push created data and get already existent information, and the interface called by the frontend to get the created information by the backend. The data flow dependency is only in one direction: from backend to frontend, e.g. anomaly score and anomaly ranking values of monitored system.

Acceptance Criteria: It's possible to store data from backend via the created interface. It's possible that the frontend can access the stored data from the backend via the created interface. The data storage is persistent, so that no monitoring data will be lost. The interfaces are well documented and expandable. The data exchange mechanism is full automatic, so that the user doesn't need to trigger something.

I2 **Calling modalities from Frontend to Backend**

Description: The user can start and stop the monitoring, which is located in the backend of Kiekeriki, via an graphical user interface provided by the frontend. The interface should be expandable to build in the configuration of monitoring processes.

Acceptance Criteria: There is a graphical user interface provided by the frontend to start and stop the monitoring of a system. The backend will be notified about the start and stop triggers and starts and stops accordingly and correctly the monitoring. The created interface to communicate between frontend and backend is well documented and expandable in order to build in the configuration of backend.

Yannic Noller

Chapter 4

Project Management

4.1 Tooling

This section describes the underlying setup for the development project. The environment follows the continuous integration techniques and their requirements described in Section 2.8.

4.1.1 Hardware Specification

The tooling setup has been realized on a EX60-Server [Het, a] sponsored by Hetzner Online AG [Het, b].

The detailed hardware specifications are as follows:

- CPU: Intel(R) Core(TM) i7 CPU 920 (2.67GHz)
- RAM: 48 GB DDR3 RAM
- HDD: 2x 2TB (Western Digital WD2000FYYZ)
  (in software raid1)
- Bandwith: 1Gbit/s

Many thanks to Hetzner Online AG for the sponsorship.

Matthias Fetzer

4.1.2 Version Control System

Before the actual project started, the team had to chose between three, widely spread, version control systems:

- Git [Git] - A distributed version control system (see Section 2.8.1).
- Mercurial [Mer] - A distributed version control system (see Section 2.8.1).
- Apache Subversion [Sub] - A centralized version control system (see Section 2.8.1).

The team chose to use Git, as it offers, unlike SVN, a distributed version control. Furthermore the team chose Git over Mercurial, as several team members had previous experience/expertise with Git.

The specific setup:

The specific setup was implemented on the dedicated server (see 4.1.1), using redmine (see 4.1.3) to manage the repositories and the access control. Each sub-team obtained a dedicated repository:

- Backend
- Documentation
- Experiments
- Frontend

Matthias Fetzer

4.1.3 Ticket-Management

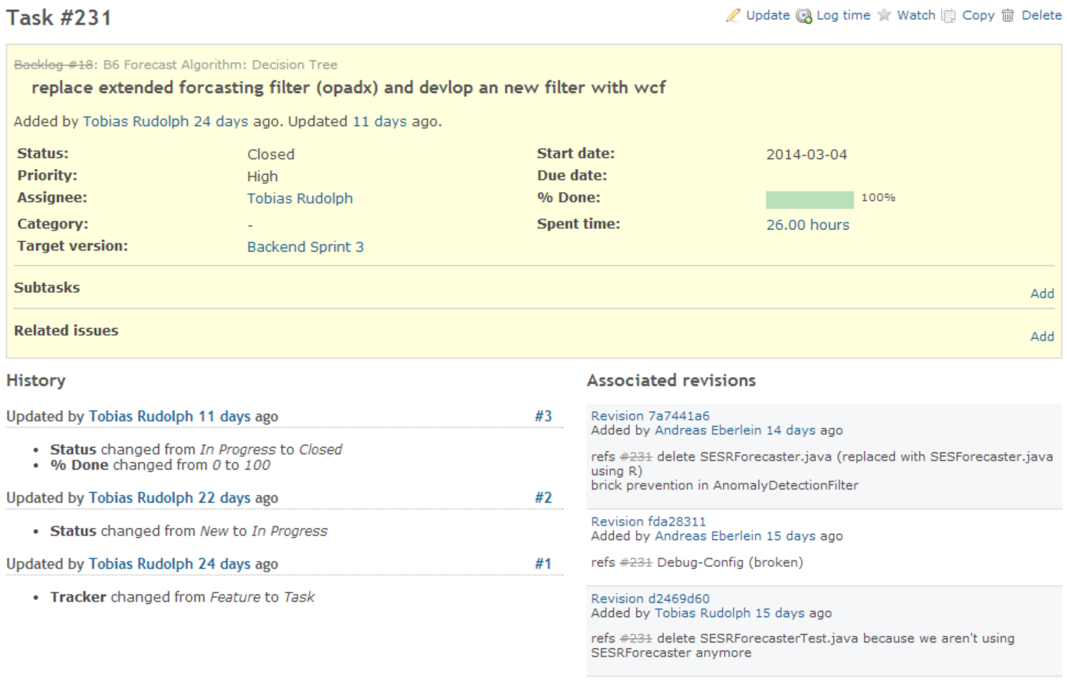
In addition to the central administration of the source code, a stable and clear management system is required for the organization of a large software project. For this purpose we could choose between two open source products "Trac" [Tra] and "Redmine" [Red]. They plot changes to the source code while providing the possibility to link revisions with the project planning. The project plan is managed in this software and serves as a central information platform for developers.

Both Trac and Redmine are extensible through plugins and can be adapted to the individual needs of the project stakeholders. Here, especially the longer established Trac is characterized by a large variety of enhancements that were developed for it in the course of time. But many extensions, particularly useful for SCRUM, are not under active development and can only be used with older versions of Trac. As a result of the recently mentioned information and our positive experiences, the stakeholders chose to use Redmine.

Redmine already provides useful modules in its original state and requires little configuration effort before operational capability. To link Redmine to the continuous integration system, provide advanced integration into development environments and enhance usability, the following plugins were installed.

- Redmine Git Hosting Plugin v0.6.2
- Redmine Hudson plugin v2.1.2
- Redmine LaTeX MathJax v0.1.0
- Mylyn Connector plugin v2.8.2
- Redmine plugin views revisions plugin v0.0.1

The most important plugin for the project tracking is the "Redmine Git hosting plugin" (RGHP). The RGHP acts as an interface between Redmine and the Git version control system (VCS), imports changes from the VCS and assigns them to the corresponding Redmine tickets. A fixed rule (git-hook) ensures that only commits can be sent to the VCS, which include a ticket number and thus are clearly assigned. Because of this limitation, the developers are urged to commit progressions individually and thereby support clarity.


Task #231

~~Backlog #18~~: B6 Forecast Algorithm: Decision Tree

replace extended forcasting filter (opadx) and devlop an new filter with wcf

Added by Tobias Rudolph 24 days ago. Updated 11 days ago.

Status:	Closed	**Start date:**	2014-03-04
Priority:	High	**Due date:**	
Assignee:	Tobias Rudolph	**% Done:**	100%
Category:	-	**Spent time:**	26.00 hours
Target version:	Backend Sprint 3		

Subtasks Add

Related issues Add

History

Updated by Tobias Rudolph 11 days ago #3

- **Status** changed from *In Progress* to *Closed*
- **% Done** changed from *0* to *100*

Updated by Tobias Rudolph 22 days ago #2

- **Status** changed from *New* to *In Progress*

Updated by Tobias Rudolph 24 days ago #1

- **Tracker** changed from *Feature* to *Task*

Associated revisions

Revision 7a7441a6
Added by Andreas Eberlein 14 days ago

refs ~~#231~~ delete SESRForecaster.java (replaced with SESForecaster.java using R)
brick prevention in AnomalyDetectionFilter

Revision fda28311
Added by Andreas Eberlein 15 days ago

refs ~~#231~~ Debug-Config (broken)

Revision d2469d60
Added by Tobias Rudolph 15 days ago

refs ~~#231~~ delete SESRForecasterTest.java because we aren't using SESRForecaster anymore

Figure 4.1. Redmine task example

As shown in Figure 4.1, backlog items defined by the organization staff were later divided into smaller subtasks to be assigned to individual developers. The backlog item shown in this case "Backlog #18: B6 Forecast Algorithm: Decision Tree" includes the task "replace extended forcasting filter (opadx) and to devlop new filter with wcf". The code changes referenced by the RGHP can be observed and tracked in the "Associated revisions".
Based on this information, the developers can set the status of the task and thus provide information to the public. For example the persons responsible for the test project orient themselves on this information and can automatically receive e-mail when the state reaches a certain level if desired. The ticket states, we use are as follows:

- New
- In Progress
- Resolved
- Feedback
- Closed
- Rejected
- Estimated

Furthermore Redmine is used to exchange generic information and how-tos via the integrated wiki and provide configuration files about the file and document module.

Andreas Eberlein

4.1.4 Build-Server

For the most other necessary tools and server services we had alternatives, which were presented to the stakeholders for decision. But there was no free usable alternative for the build server, who offered themselves for Java development. While with "Apache Continuum" [Apa] there are also other continuous integration environments, none of the project participants had positive experience with those in the context of reliability and usability. Therefore we use "Jenkins" [Jen] with the following custom plugins to automate and test our software projects.

- Ant Plugin
- Checkstyle Plug-in
- Copy Artifact Plugin
- FindBugs Plugin
- Git Client Plugin
- Git Plugin
- Jenkins Redmine plugin
- Maven Integration plugin
- PMD Plug-in
- Static Code Analysis Plug-ins

In Jenkins all parts of the project that are located in different Git repositories are created and configured with the associated build scripts. Jenkins gets the current file content of each repository and executes the build script (Section 4.1.5) that produces results periodically to the defined periods. These results are reported in case of failure or unacceptable code quality to the developer who caused them, as well as Redmine, so that the errors can be quickly identified and resolved. The build can also be started manually from the Redmine- or the Jenkins user interface at any time.

Andreas Eberlein

4.1.5 Build Automation

Build automation enables developers to automate certain tasks in software development, such as:

- Code compilation
- Package generation
- Unit testing
- Automated deployment
- Javadoc generation
- Code-Style checking
- etc.

There are several tools which can aid in achieving these tasks. During the project start the team had to chose between the following build systems:

- Apache Ant[ant]
- Apache Buildr[bui]
- Gradle[gra]
- Apache Maven[mav]

The team chose Maven as their build-tool, as a few team members had previous experience with Maven and because of its comfortable dependency management. Another advantage of using Maven was the amount of know how that can be obtained by searching the internet, compared to Buildr or Gradle. Besides the standard Java- and JUnit-plugins for Maven, we additionally used the following plugins:

- Maven FindBugs Plugin[Mav, b]
- Maven CheckStyle Plugin[Mav, a]
- Maven PMD Plugin[Mav, c]

Matthias Fetzer

4.1.6 Supplemental Tooling

Besides the Java-tooling, other technical-infrastructure tasks had to be set up and maintained. This section gives an overview of the accomplished tasks, which were not covered in the sections above. Figure 4.2 shows an overview of the complete infrastructure that we use for the development project.

Andreas Eberlein

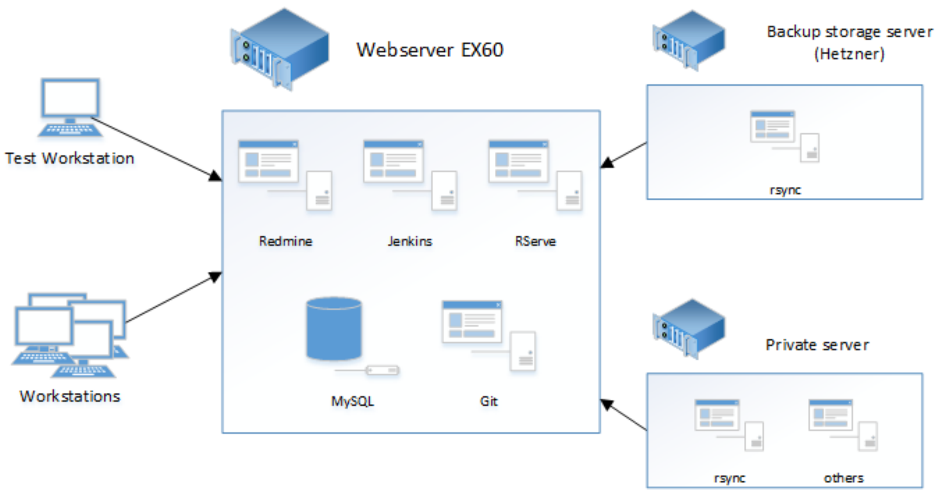


Figure 4.2. Infrastructure

R-Server

A high-performance, free and cross-platform usable implementation of the respective algorithms is needed for statistical and complex calculations. To keep them up to date and be also independent of the Kiekeriki backend, the specialized software environment for statistical computing "R" [RPr] is used at this point. The R software environment includes

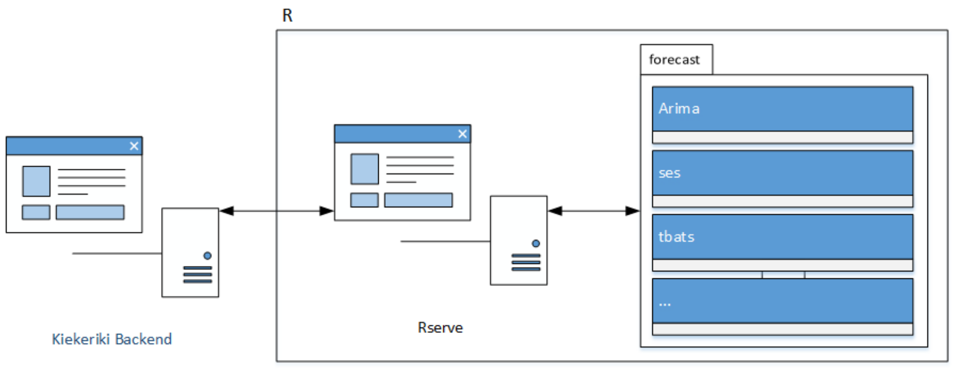


Figure 4.3. "R" execution environment

the server service "Rserve" version 3.0.3 as shown in Figure 4.3 which runs in the context of the development project on the continuous integration server and the evaluation workstation. The backend requires the R-package "forecast", which is used in version 5.3. Since this is continually enhanced and as the upcoming release can contain further optimization to forecasters, especially this should be kept up to date.

The service "Rserve" has to be run on the same server on which the Kiekeriki backend is running due to the fixed local destination address in ΘPAD.

Andreas Eberlein

SQL-Server

Due to the fact that the communication between the frondend and backend is realized via a central SQL-Database, a MySQL-Server has been installed (and slightly tuned for performance with the help of mysqltuner[mys].

Matthias Fetzer

Disaster Recovery

To ensure that the team can continue to work in case of a hardware or software failure, the following (automated) backup mechanisms have been installed:

- Semi-hourly backups of the MySQL-Databases and its procedures via mysqldump.
- Daily file-based backups to the backup-storage located at the Hetzner-Datacenter.
- Daily file-based backups to a Backup-System, located at a private appartement.

The file-based backups have been done with rsnapshot[1]. This enables us to keep several revisions of all changed files, even the ones which are not under git-version control. The retention settings for rsnapshot are:

- 7 daily backups
- 4 weekly backups
- 12 monthly backups

Matthias Fetzer

4.2 Scrum

In order to produce good results in the short time of this project we decided to use an agile software developing process. We chose scrum because it is a lightweight process structure with few specifications and therefore customizable to our needs. The following subsections describe how scrum is implemented in our development project.

[1]http://www.rsnapshot.org/

4.2.1 Basics

We have divided our project *Team* into two *Teams* with 5 persons (frontend) and 7 persons (backend). The frontend *Team* is responsible for implementing the user interface as a web application. The backend *Team* on the other hand implements the requested algorithms and provides data for the visualization for the user interface.

Roles

There are three different roles in our approach of the *Scrum* process: *Product Owner* (separate persons for frontend and backend), *Team* and *Scrummaster*. For more information about these roles see Section 4.4.

Sprint

A *Sprint* is a fixed-length „subproject“ in which features of the project are developed. It is repeated throughout the projects lifetime and combines the design and implementation phase. In our case we chose *Sprint* lengths of either 7 or 14 days to have multiple *Sprints* during the project. Both *Teams* can plan the number of *Sprints* as well as their corresponding length independently. This allows both *Teams* to find *Sprint* lengths which suits their needs best. Before the start of every *Sprint* there is the *Sprint Planning Meeting* in which the *Product Owner* presents the *Product Backlog Items* with the highest priority. For further information about the *Sprint Planning Meeting* see Section 4.2.1. At the end of one *Sprint* there is a *Sprint Review Meeting* in which the *Team* demonstrates the developed features on a running system to the *Product Owner*. The *Product Owner* can then express additional ideas and alter *Product Backlog Items* to make sure that the system is developed exactly the way he needs it. The next *Sprint* starts then again with a *Sprint Planning Meeting* until the project is finished.

Artefacts

Multiple artefacts should be created according to the *Scrum* approach:

User Story A short (only a few sentences) and high-level description of a feature which the customer needs. In our case we tried to create the *User Stories* according to the *INVEST* approach. This approach states that each *User Story* should be: Independent, Negotiable, Valuable, Estimable, Small and Testable.

Product Backlog A prioritized list of requirements (*Backlog Items)* which need to be done in the course of the project. Usually the *Backlog Items* start as the *User Stories* and are expanded by adding more detailed descriptions of the feature. The *Product Backlog* is flexible, which means that the *Product Owner* can add, delete or modify items at any given time.

Sprint Backlog The *Sprint Backlog* contains the items which will be developed during the current *Sprint*. After the *Sprint Planning Meeting* a list of *Backlog Items* are removed from the *Product Backlog* and added to the *Sprint Backlog*. Compared to the *Product Backlog* the *Sprint Backlog* is a lot smaller and, more importantly, not flexible. This means that items in the *Sprint Backlog* can not be altered, removed or added. This leads to stable requirements for the developers which in turn leads to better results because the developers do not need to change features while they are being implemented.

Meetings

Sprint Planning Meeting The *Sprint Planning Meeting* is always at the beginning of a *Sprint*. During this meeting the *Product Owner* presents the high priority items from the *Product Backlog* that need to be implemented next. He also gives more detailed information and answers questions from the *Team* about the items. For every relevant *Backlog Item* the *Team* then uses an approach called *Planning Poker* to estimate how many days are necessary to implement this specific item. During the *Planning Poker* the developers simultaneously show cards with a number on according to their estimated time requirement for a *Backlog Item*. The developers with highest and lowest estimation than discuss briefly ($\leqslant$ 2 min) why they chose their corresponding card. After that the whole *Team* estimates again. This is repeated until every developer estimates the same amount (shows the same card). After all relevant *Backlog Items* have been estimated the *Team* chooses a set of items which will be implemented in the next *Sprint* and thereby move the items from the *Product Backlog* to the *Sprint Backlog*. That, in turn, means that these items can not be modified any longer until the *Sprint* is finished.

Sprint Review Meeting This meeting is always at the end of a *Sprint*. Usually the *Team* demonstrates the created features in a live demo to the *Product Owners*. Because both *Product Owners* are part of the *Team* this is not necessary in our project. Instead we discuss how much of our goals have been reached and whether or not it is necessary to re-insert and re-estimate specific items into the *Sprint Backlog* for the next *Sprint*.

Daily Scrum The *Daily Scrum* is a short ($\leqslant$ 15min) Meeting that takes place every day for both groups separately. Each developer states what he has done since the last meeting, what he is about to do and (if necessary) what prevents him from doing so.

4.2.2 Process

The first step in our *Scrum* approach is the gathering of the requirements. Responsible for this are the *Product Owners*. Their goal is to collect the required information and report them as *Backlog Items* in our issue tracking system (ITS). These *Backlog Items* are short, high-level descriptions of functionalities which the customer wants. They are further segmented into more fine-grained sub-features and finally into tasks which can then be assigned to

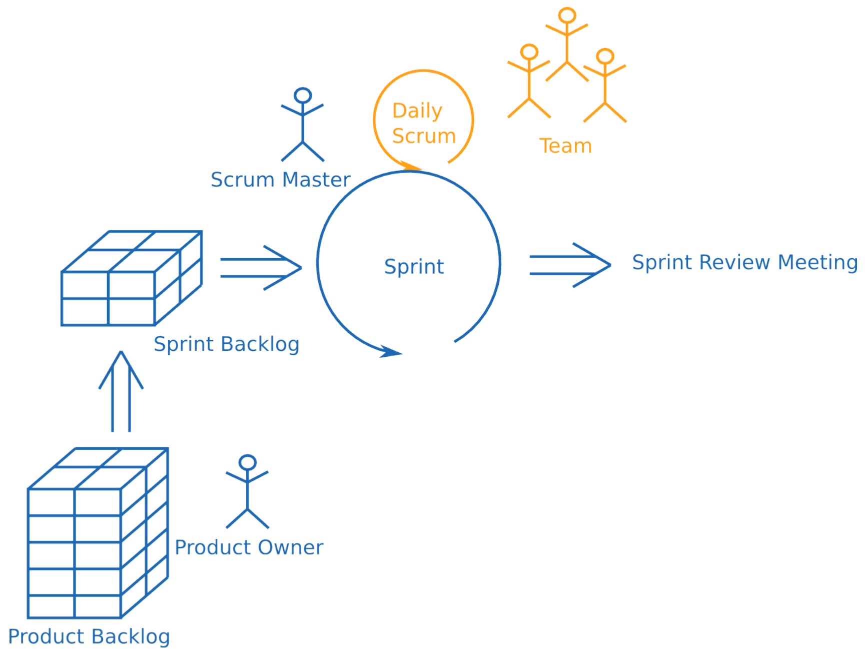


Figure 4.4. Overview of the *Scrum* process

Team members. The segmentation and creation of tasks is done by the corresponding *Product Owners*. This is not necessarily applicable for every Scrum-related project, but in our case we have the advantage that the *Product Owners* are developers as well and can therefore reckon which sub-features and tasks are necessary to implement the functionality.

After the initial set of sub-features and tasks are created the first *Sprint* starts. At the beginning of the *Sprint* there is always a *Sprint Planning Meeting* where the *Teams* estimate and choose the features which will be implemented. During the *Sprint* both *Teams* work on the items in the *Sprint Backlog* and have separate *Daily Scrums* every day to keep every *Team* member informed about the current progress. See also Figure 4.4

At the end of the *Sprint* there is the *Sprint Review Meeting* as discussed previously (Section 4.2.1). Furthermore, every *Sprint Backlog Item* should be completed by that time. Otherwise it needs to be transferred into the *Sprint Backlog* for the upcoming *Sprint*.

4.2.3 Sprint Overview

The following shows the *Sprints* as well as the tasks which were chosen by both *Teams*.

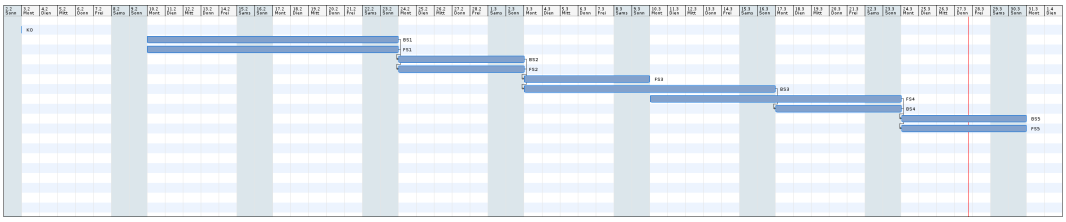


Figure 4.5. Sprints for both *Teams* during the project.

Table 4.1. This table shows the backend tasks and when they were completed.

Backlog Item	**Task**	**Sprint**
RanCorr	Implement RanCorr	BS1
	Provide Kieker Plug-In	BS1
	Database extension	BS3
Self-Tuning	Implement runtime configuration	BS1
	Develop composite filter plugin	BS3
Default-Configuration	Find and provide default values	BS5
Experiment Framework	Identify format of testdata	BS1
	Manually build testdata	BS2
	Automatically build testdata	BS3
	Build experiment framework around ΘPAD and RanCorr	BS4
	Automate evaluation of ΘPAD and RanCorr	BS4
Forecast Algorithm: Decision Tree	Implement WCF	BS4
	Replace existing forecaster	BS4
Various Metric Monitoring	Store CPU Load	BS4
	Store RAM Load	BS4
	Store Swap Load	BS4
	Store Latencies	BS4

Table 4.2. This table shows the frontend tasks and when they were completed.

Backlog Item	**Task**	**Sprint**
Architecture Overview	Create clean Web-GUI version	FS1
	Create dashboard overview	FS1
	Display aggregated information	FS1
	Get data from DB	FS1
	GUI sketch	FS1
Deep dive approach	Build model	FS2
	Connection to transfer DB	FS4
	Extend architecture view	FS4
Diagram View	Create diagrams dynamically	FS2
	Database connection for diagrams	FS2
	Create configuration dialogue	FS3
	Save diagram config in DB	FS3
Metrics Definition	Define default metrics	FS3
	Create settings tab content	FS4
Showing Anomaly Detection (RanCorr)	Show results in architecture graph	FS3
Displaying live and historical data	Display current data in architecture graph	FS3
	Implement polling mechanism for diagrams	FS4

Dominik Olp

4.3 Team

In the following is a mapping of the most important tasks during the project's lifetime and the corresponding team members who worked on it.

Table 4.3. Legend

●	worked mainly on this item
❍	worked partially on this item
	didn't work on this item

Table 4.4. This table shows the tooling tasks and who worked on them.

Task description	**Andreas Eberlein**	**Matthias Fetzer**
VCS (Git)	●	●
Ticket System (Redmine)	●	●
Build-Automation (Jenkins)	❍	●
Build-Tool (Maven)	●	●
SQL Server		●
R Server	●	
Backups	❍	●
Support	●	●

Table 4.5. This table shows the frontend tasks and who worked on them.

Task description	**Christian Endres**	**Matthias Fetzer**	**Kálmán Képes**	**Anton Scherer**	**Martin Scholz**
F1 (Arch. View) Clean Web-GUI version			●		●
F1 (Arch. View) Create dashboard overview		●	●	●	●
F1 (Arch. View) Show aggregated info		●	●	●	●
F1 (Arch. View) Get data from DB	●				❍
F1 (Arch. View) GUI sketch	❍	❍	❍	●	❍
F2 (Deep Dive) Build model	●				
F2 (Deep Dive) Connection to transfer DB	●				
F2 (Deep Dive) Extend architecture view		●	❍		
F3 (Diag. View) Dynamic diagrams				●	
F3 (Diag. View) DB conn for diagrams	●				
F3 (Diag. View) Create config dialogue			●		
F3 (Diag. View) Save diagram config in DB					●
F4 (Metrics Def.) Define def. metrics	●				
F4 (Metrics Def.) Create settings tab				●	
F7 (RanCorr) Show results in arch. graph		●			❍
F9 (Live & hist. data) Show data	❍	●			
F9 (Live & hist. data) Implement polling				●	●
Experts Feedback	●			●	
Benchmarks / Validation Frontend	●				
Transfer Database (Backend/Frontend)	●				
Layout & UI-Design		❍	●	●	❍
Cache		●			
Unit Tests	●				●
I1 Data Exchange FE-BE	●				
I2 Calling modalities FE-BE	●				

Table 4.6. This table shows the backend tasks and who worked on them.

Task description	**Thomas Düllmann**	**Andreas Eberlein**	**Markus Fischer**	**Christopher Gregorian**	**Yannic Noller**	**Dominik Olp**	**Tobias Rudolph**
B1 (RanCorr) Implementation			❍		●	●	
B1 (RanCorr) Provide Kieker Plug-In					●	●	
B1 (RanCorr) Database Extension			❍		●	●	
B1 (RanCorr) Testing			●	●	●	●	
B2 (ΘPAD) Runtime Configuration	●		●				●
B2 (ΘPAD) Composite Filter Plug-In	●		●				
B2 (ΘPAD) Testing	●	●	●	●		●	●
B4 (Experiment) Identify Format of Testdata				●			
B4 (Experiment) Manually Build Testdata				●			
B4 (Experiment) Automatically Build Testdata				●			
B4 (Experiment) Framework RanCorr & ΘPAD				●			
B4 (Experiment) Evaluation RanCorr & ΘPAD	●		●	●	●	●	
B6 (ΘPADx) Implement WCF		●					●
B6 (ΘPADx) Replace existing forecaster		●					●
B11 (Metrics) Store CPU Load					●	●	
B11 (Metrics) Store RAM Load					●	●	
B11 (Metrics) Store Swap Load					●	●	
B11 (Metrics) Store Latencies	●				●	●	
I1 Data Exchange FE-BE	❍				❍	❍	
I2 Calling modalities FE-BE	●						

Dominik Olp

4.4 Roles

We defined multiple roles in our project. In the following there will be a list of roles as well as their scope of work.

Quality Assurance Engineer: Enforces quality assurance measures. Makes sure all developers write test cases for their corresponding area. Verifies that all developers write their code according to a common style guide.
Responsible: Markus Fischer

Document Representative: Creates and maintains document structures. Additionally, takes care that all created documents are up to date. Makes sure documents get finished in time.
Responsible: Martin Scholz

Product Owner(s): Act as the interface between the project *Team* and the supervisor/customer. They are responsible for creating and maintaining a list of requirements. Furthermore, they need to answer upcoming questions from the *Team* about requirements. Additionally they decide the priority of requirements and therefore influence which requirements are dealt with in any *Sprint* (for more information about *Sprints* and our development process see: Section 4.2).
Responsible: Anton Scherer (Frontend), Yannic Noller (Backend)

Infrastructure Representative(s): Handle the infrastructure as well as the software needed to realize the project. Especially following topics are under the supervision of the infrastructure representatives:
- Set-up and maintenance of a versioning system
- Integrating issue tracker
- Set-up of a buildsystem
- Enable continuous integration & automated testing
- Set-up and maintenance of databases

Responsible: Andreas Eberlein, Matthias Fetzer

Scrummaster/Projectleader: Enforces a scrum-conform approach and acts as a moderator in meetings to avoid long discussion and thereby ensures that time constraints are met.
Responsible: Dominik Olp

Developer (aka *Team*): Implementation of the requirements specified by the *Product Owner*.
Responsible: Everybody

Dominik Olp

Chapter 5

Architecture and Concepts

Due to the different foundations front- and backend are based on, the architecture chapter is split in separate sections. To put them into context, we will give an overview first.

5.1 Overview

The following overview (Figure 5.1) shows the overall structure of the Kiekeriki implementation.

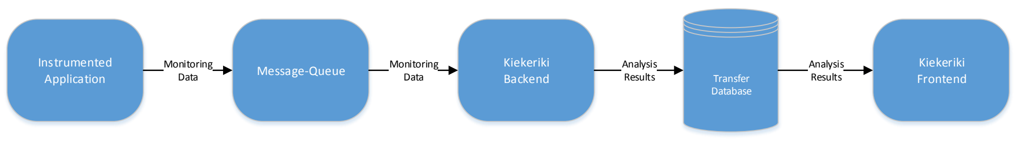


Figure 5.1. Overview of the architecture of Kiekeriki

First of all the instrumented application generates monitoring data which is sent to a message queue. From there it is read by the Kiekeriki backend to analyze the received data. Afterwards the results of the analysis are written to the transfer database which is the main interface between the Kieker front- and backend. The stored results then can be read by the Kiekeriki frontend to display them in a web-based frontend. The inner structure of front- and backend is explained in detail in the following sections.

Thomas Düllmann

5.2 Frontend

This chapter describes the architecture of the frontend of Kiekeriki and the connection between the frontend and the backend via the connecting database. An outline of the architecture of the backend can found in chapter 5.3.

The frontend mainly consists of four big parts (see: Figure 5.2):

- The logic of the backend
- The web interface (The GUI)
- The transfer database
- The internal database

The main focus of this chapter lies on the interconnection between these parts. To get more information about the details of implementation of these parts, please see chapter 7.

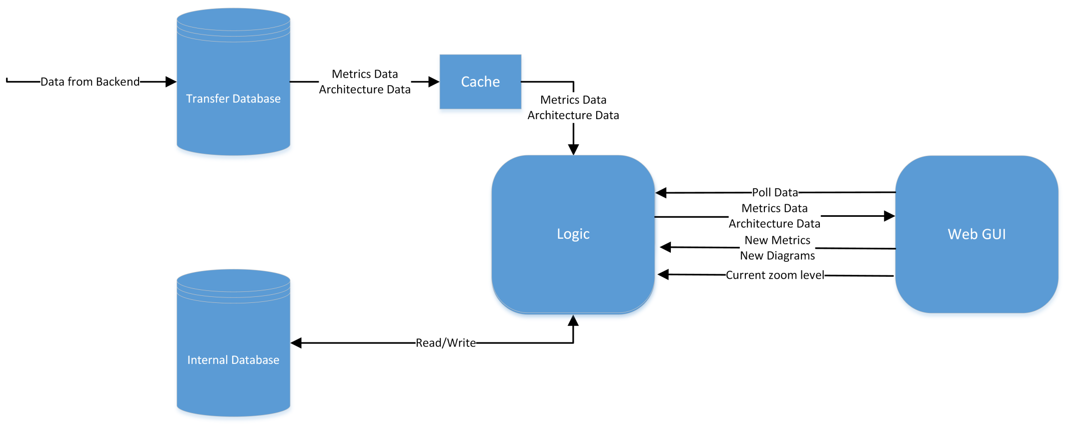


Figure 5.2. Overview of the architecture of the Web GUI

5.2.1 Connection Between Logic and GUI

The connection between the logic an the GUI mainly consists of transfer data about the architecture and the data of the metrics to the GUI. This connection also handles the management of metrics and diagrams. The first important fact is, that the GUI is the active part in this connection. It polls data from the logic, when it needs them. This applies to every module of the GUI: The architecture view, the diagrams as well as the information about exceptions at the bottom of the page. The GUI also has to send data back to the logic - but only in a few cases: When the user creates or deletes a metric, when the user creates, updates or deletes a diagram and when the zoom level of the architecture view changes - so the logic always knows the state the GUI is in. This is important for the cache (see 7).

5.2.2 Connection to the Transfer Database

The transfer database contains all the information, the backend produces about the monitored system. This includes the information about the architecture of the system and the data for each metric. From the view of the frontend, is only possible to read data from the database. The frontend can not write to the transfer database. There is one single class, which accesses this database directly and offers an interface for the queries - the class

`DatabasePlainConnection`. To reduce the load of the database, there is a cache between all the components, which needs data of the database and the `DatabasePlainConnection`. This cache holds all information, which were already queried in its memory for a certain amount of time. Components, which query data, get the data from the memory of the cache instead of directly from the database. Only if the data is outdated (which is defined differently for different types of data), the cache fetches the new data from database before returning them to the component, which queries for it.

5.2.3 Connection to the Internal Database

For the storage of diagrams and the metrics, the internal database is used, which was already used by Kieker to store user related data. Two new tables, one for the metrics and one for the diagrams were added to fit our needs. Each diagram is related to one metric and each metric is related to one user.

Martin Scholz

5.3 Backend

This section will explain the resulting architecture of the Kiekeriki backend in detail. As the backend of Kiekeriki is based on the Kieker 1.8 analysis module, it uses the same pipes and filter architecture.

The Kiekeriki backend consists of three main modules which are referred to as ΘPAD(x), RanCorr and System Discovery. Their inner structure will be explained in the respective subsections. Figure 5.3 shows the data flow between these modules.

The Kiekeriki backend reads the data from the monitored system from the message queue, which is then processed by ΘPAD(x) and the architecture discovery module. ΘPAD(x) uses the data to detect anomalies using time series analysis with the help of the forecasting database which holds the history of forecasted and actually measured data in the past. The architecture discovery module uses the incoming data to reconstruct the structure of the instrumented application. The results of the modules mentioned beforehand are forwarded to the RanCorr module, which correlates the anomalies with the structure of the application to give a more sophisticated estimation, where the source of the anomalies is. Finally the results of each module are written to the transfer database.

5.3.1 ΘPAD(x) Module

In Figure 5.4 the structure of the ΘPAD(x) module is depicted. In addition to the already established ΘPADx filter setup, which was developed by Frotscher [2013] (from `ExtractionFilter` to `SendAndStoreDetectionResultFilter`) we added native Kieker filters and developed filters ourselves to fit the ΘPADx functionality into the context of Kiekeriki. To

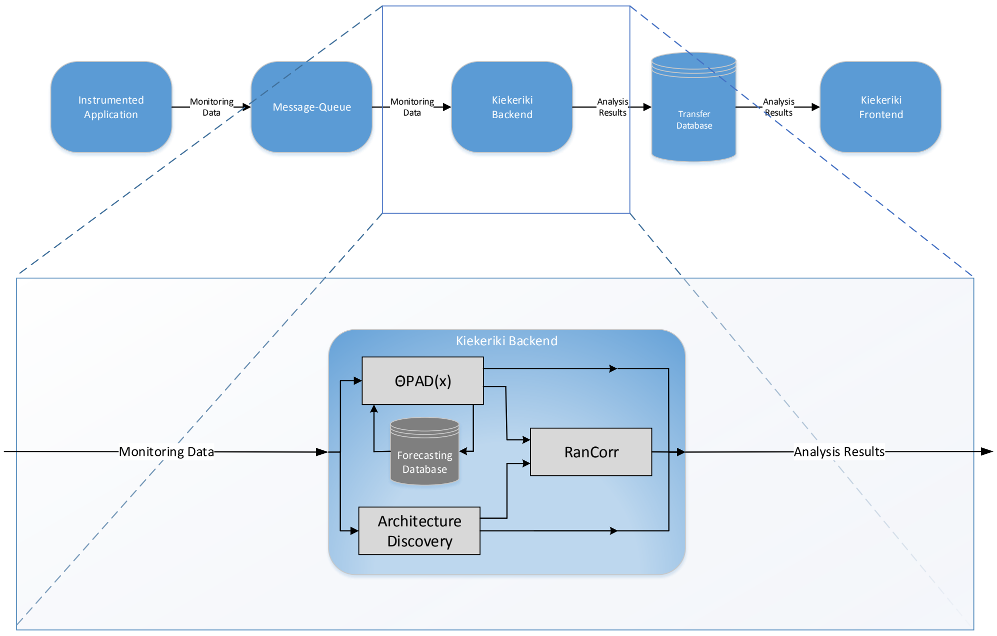


Figure 5.3. Overview of the architecture of Kiekeriki backend

easily use the ΘPADx functionality, we developed a composite filter plugin that wraps the original ΘPADx functionality in one single filter.

The following diagram (Figure 5.4) and list explains shortly what the filters that have been added to the original ΘPADx approach do. As the ΘPADx has been explained in detail by Frotscher [2013]. The corresponding filters are ommited in the listing.

- **JMSReader**
  Reads the monitored data from a Java Message Service (JMS) message queue.
- **RecordConverter**
  Converts the incoming OperationExecution information to an ΘPADx-compatible format.
- **StringBufferFilter**
  Buffers strings to optimize memory usage.
- **ΘPADx Filters**

- **OpadOutputCompositionFilter**
  Creates output objects that contain all information that ΘPADx can provide from the incoming data.

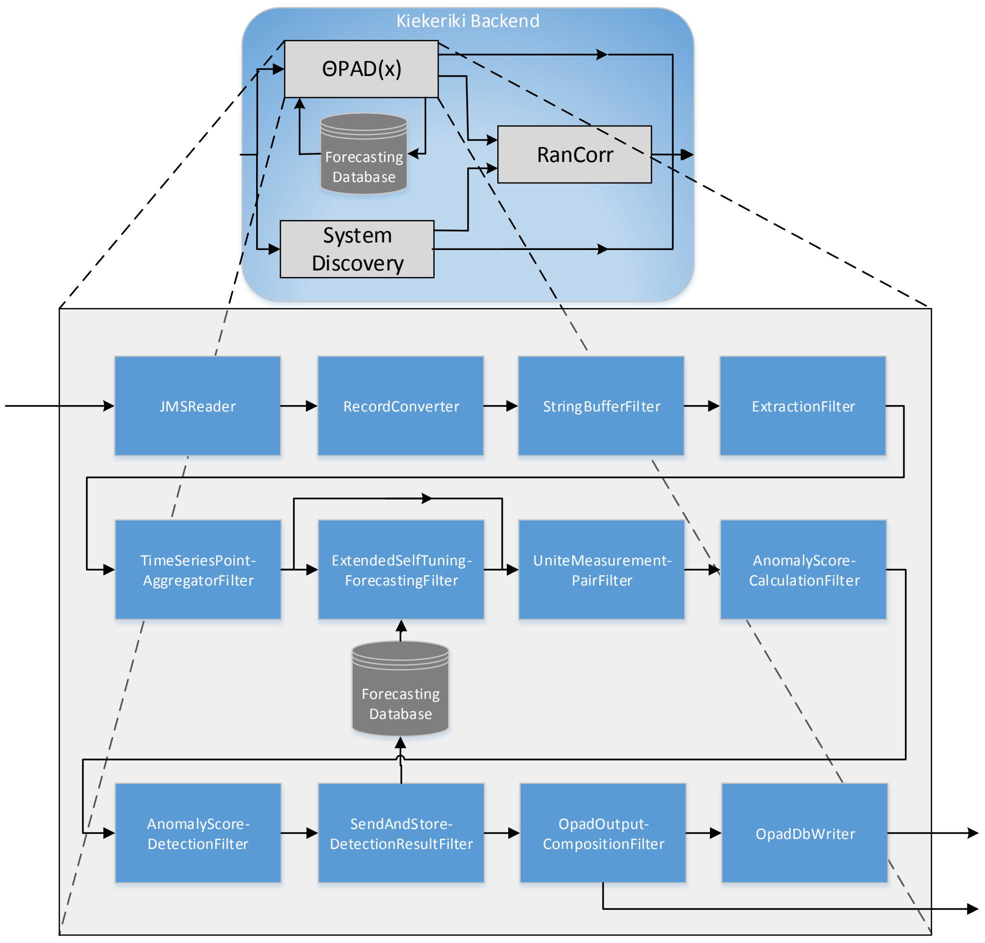

Kiekeriki Backend
ΘPAD(x)
Forecasting Database
RanCorr
System Discovery
JMSReader
RecordConverter
StringBufferFilter
ExtractionFilter
TimeSeriesPoint-AggregatorFilter
ExtendedSelfTuning-ForecastingFilter
UniteMeasurement-PairFilter
AnomalyScore-CalculationFilter
Forecasting Database
AnomalyScore-DetectionFilter
SendAndStore-DetectionResultFilter
OpadOutput-CompositionFilter
OpadDbWriter

Figure 5.4. Overview of the architecture of the ΘPADx part within the Kiekeriki backend

- **OpadDbWriter**
  Writes the composed data to a database.

5.3.2 System Discovery Module

The system discovery module shown in Figure 5.5 is a composition of filters, that mainly already existed in Kieker. The purpose of this module is to use the incoming monitoring data to reconstruct the method call traces and write them and additional system metrics to the database.

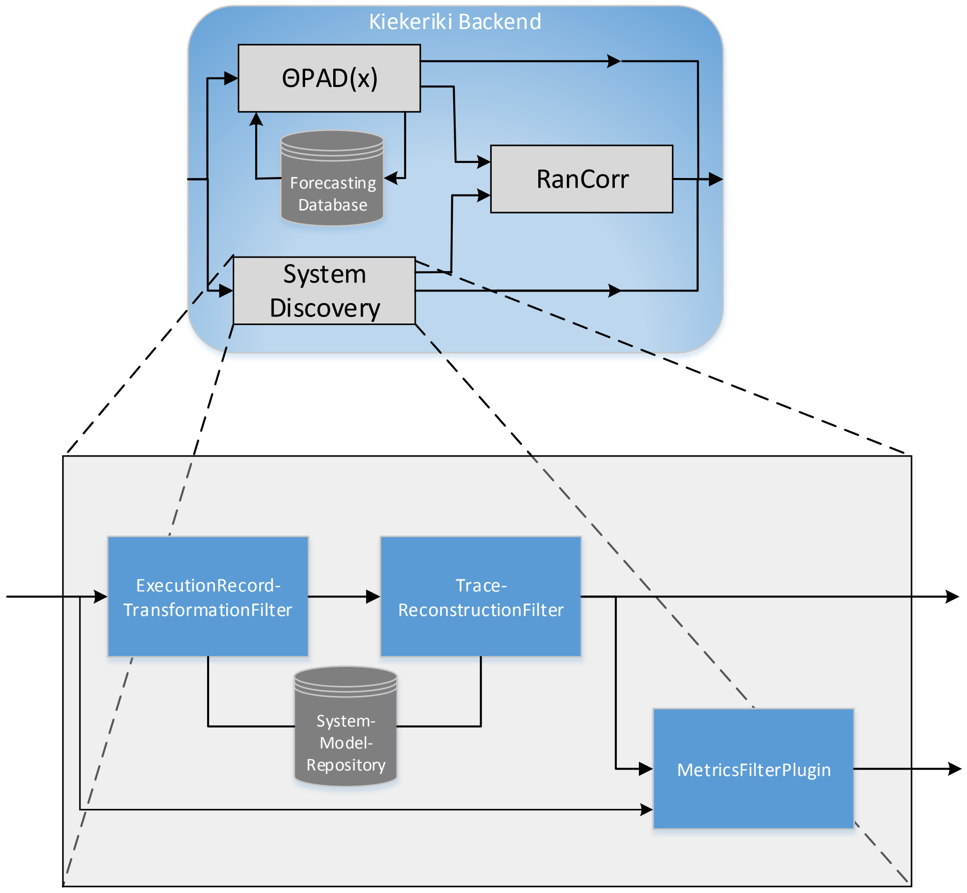


Figure 5.5. Overview of the architecture of the system discovery part within the Kiekeriki backend

The `ExecutionRecordTransformationFilter` and the `TraceReconstructionFilter` in combination with the `SystemModelRepository` already existed in Kieker. We added the `MetricsFilterPlugin` which writes system metrics (e. g., CPU usage) received from the module input and the discovered method call traces from the `TraceReconstructionFilter` to the transfer database.

5.3.3 RanCorr Module

The RanCorr module shown in Figure 5.6 uses the anomaly scores from the ΘPAD(x) module and the operation calls from the system discovery module to calculate RanCorr ranks.

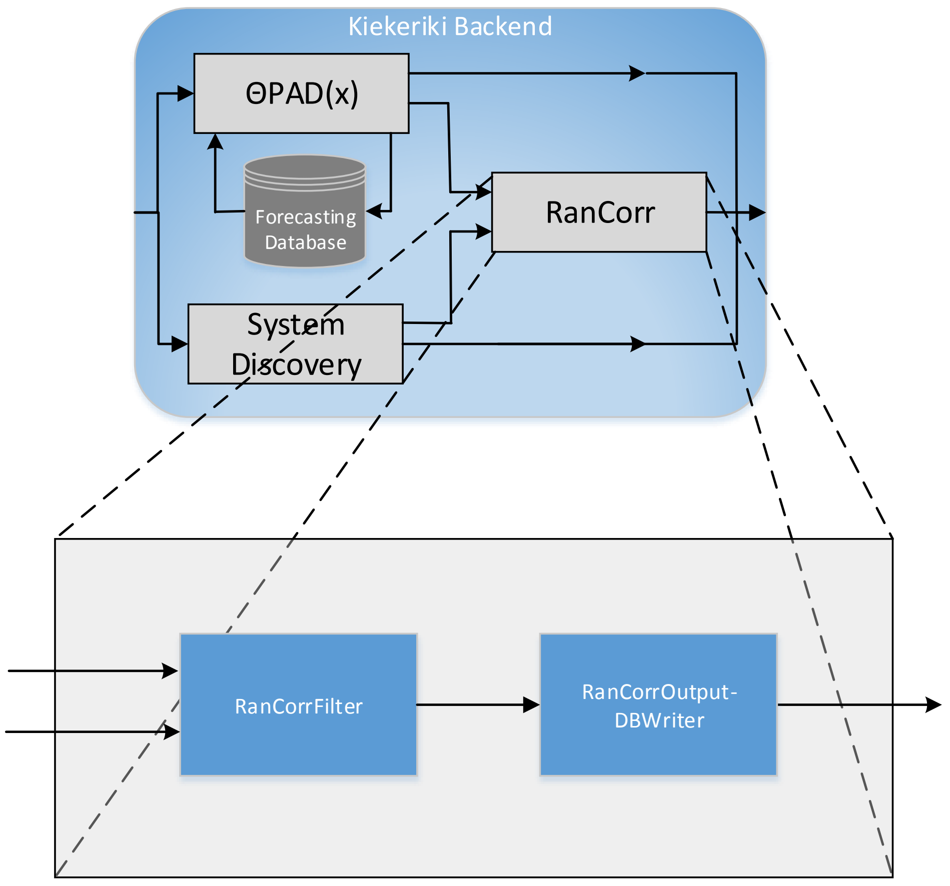


Figure 5.6. Overview of the architecture of the RanCorr part within the Kiekeriki backend

The `RanCorrFilter` implements the RanCorr approach developed by Marwede et al. [2009] and the `RanCorrOutputDBWriter` writes the results of the `RanCorrFilter` to the transfer database.

Thomas Düllmann

5.4 Raw Data, Data Aggregation and Metrics

This section describes the principles of the raw data, aggregated data and the metrics stored in the Transfer Database or aggregated of the data stored in the Transfer Database. The Transfer Database is described in 7.3 A single reported data entity is called raw data. Raw data which is aggregated by functions is called a metric. The Transfer Database is described in the chapter 7.3.

5.4.1 The architecture and its components

The Kiekeriki Backend monitors and analyses components of an instrumented application, service, host or any composition of these. The results of this analyses are belonging to a host, application or any subcomponent of an application. Thus each analysis result is dedicated to one of these components or components of a major system — the so-called architecture.

The ideal of the understanding of an architecture is an Java application running on a host. The breakdown of such an architecture is an instrumented component: the host, application, any structure of packages, a class and its implemented operation. As shown in Figure 5.7 all these components of an architecture form a graph which describes which component can be parent of which component.

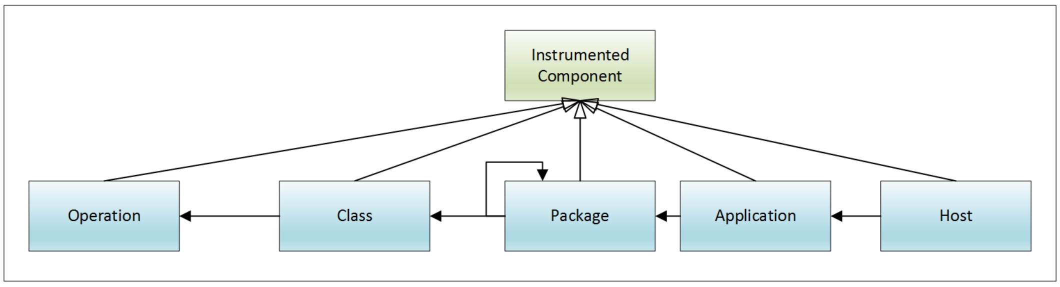


Figure 5.7. The architecture construct for the data and metrics

An instrumented system can be described as a tree and its nodes which are transitive descendants of the hosting root — the architecture tree. Only the package is characterized especially. A package can represent a whole component of an application as well as a logical or real subcomponent which is a descendant of another component, depicted by the cycle with itself in the architecture graph. Each instrumented component can be monitored on its own characteristics (the raw data) or the characteristics of its descendants (the aggregated data).

Christian Endres

5.4.2 Raw Data

Each instrumented component owns its monitoring data. The monitoring data has to be distinguished between the components own data and data which is aggregated of lower levels of the architecture tree. For example an operation can be invoked which results in a invocation latency. A package which represents a component can be also invoked but does not contain a latency itself. The latency of a package or logical component is the summation of the latencies of a trace of its operation descendants. The difference is that, for example, an operation provides raw data (the latency) and the package owns aggregated data (the aggregated latencies of the operation descendants) which is described in the next chapter. The Figure 5.8 visualizes which instrumented component owns which raw data.

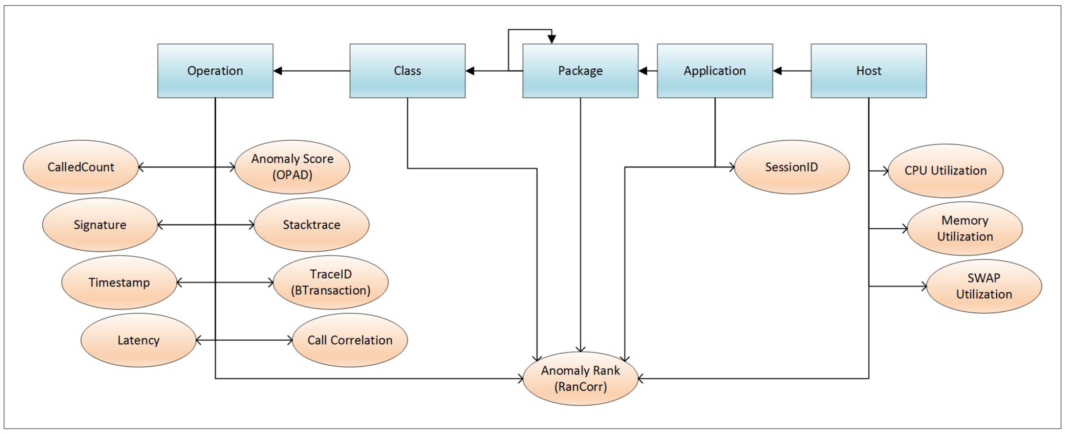


Figure 5.8. Instrumented compontents and fathered raw data

CalledCount

This raw data is in fact an aggregated metric which is very expensive to calculate. Thus as a call between two components is written to the database, the called count is increased and does not have to be calculated for each request of the Kiekeriki Frontend. The called count is treated as raw data.

Signature

The signature of each component is stored for human readable visualization.

Timestamp

For each report of raw data the corresponding timestamp in milliseconds is stored aswell.

Latency

The time in milliseconds it took to execute the operation. Additional the ID of the caller of the operation which was executed and the timestamp are stored.

ΘPAD anomaly score

The anomaly score of a operation calculated by ΘPAD.

Stacktraces (and other exceptions)

The content of a error which is thrown during an operation process. Additional the ID of the caller of the operation which threw the stacktrace and the timestamp are stored.

Call correlation

The correlation between the calling and called component. Additional the total amount of calls and timestamp are stored.

Anomaly rank

The rank of a component calculated by RanCorr.

Session (active users)

User session IDs which are active in a application. Additional the timestamp is stored.

CPU utilization

Utilization percentage of a single core of a host (value between 0 and 1). Additional the timestamp are stored.

Memory utilization

Percental utilization of the Memory of a host (value between 0 and 1). Additional the timestamp, the total memory size, the total free and total used memory size.

Swap utilization

Percental utilization of the swap of a host (value between 0 and 1). Additional the timestamp, the total swap size, the total free and total used swap size.

Christian Endres

5.4.3 Functions and Metrics

All functions process a set of raw data dependent on a bucket size. Each entity of raw data consists of the value and the timestamp on which the value was reported. The bucket size determines the intervals in which the set is divided. The function is processed on all entities in a single bucket and the result represents the bucket.

Example:

There are seven entities of latencies with the first three reported in minute 0, the second three in minute 1 and the last one in minute 2. The function average with a bucket size of one minute would return the average latencies of the first three, the second three and the latency of the last one in one bucket each minute.

The list of functions:

- Average
- Count
- Max
- Min
- Percentile
- Sum

The Table 5.1 visualizes which function on the x-axis can process raw data series on the y-axis.

Table 5.1. Functions and Raw Data

	Average	Count	Max	Min	Percentile	Sum
AnomalyRank (RanCorr)	x		x	x	x	
AnomalyScore (ΘPAD)	x		x	x	x	
CalledCount	x		x	x	x	x
CPU Util	x		x	x	x	
Latency	x		x	x	x	x
Memory Util	x		x	x	x	
SessionId		x				
Signature		x				
Stacktrace		x				
Timestamp		x				
TraceID (BT)		x				

The table 5.2 visualizes which function can be invoked on a result set of another function.

Christian Endres

5.4.4 Taxonomy for the architecture graph

The view names a "high level component" (e.g. host, application, etc.). The level describes the type of components which compose the "high level component". This classification serves

Table 5.2. Recursive Functions

	Average	Count	Max	Min	Percentile	Sum
Average			1	1	1	
Count	1		1	1	1	
Max					1	
Min					1	
Percentile	1		1	1		
Sum			1	1	1	

the deep dive approach.

Christian Endres

5.4.5 Possible raw data per view and level

Native metrics means metrics of components on this level. Aggregated metrics means metrics aggregated from lower levels and treated with functions.

Class View / operation level

native: RanCorr anomaly rank, ΘPAD anomaly score, call correlations between operations in this class, latency of an invocation, thrown stacktraces, business transactions which use/invoke the operation

Package View / class level

native: RanCorr anomaly rank
aggregated: call correlations between classes, latency of invocations, thrown stacktraces, a business transactions which use/invoke a operation in the classes

Application View / package level

native: RanCorr anomaly rank
aggregated: call correlations between packages (software components like "controller"), latency of invocations, thrown stacktraces, a business transactions which use/invoke a operation in the packages

Server View / application level

native: RanCorr anomaly rank, session IDs (active users), CPU utilization, CPU cores, Memory utilization, swap utilization
aggregated: call correlations between applications, latency of invocations, thrown stacktraces, a business transactions which use/invoke a operation in the applications

Architecture View / host level

native: RanCorr anomaly rank
aggregated: call correlations between hosts, latency of invocations, thrown stacktraces, business transactions which use/invoke an operation in the hosts, session IDs (active users)

Christian Endres

5.4.6 Default metrics

Architecture view: Error Rate, Active Users
Server View : CPU Load, Memory Load, SWAP, Error Rate, RanCorr Rank
Application View: RanCorr Rank, Error Rate
Package View: RanCorr Rank, Error Rate
Class View: RanCorr Rank, Error Rate
Operation View: Average Latency of invocation, Error Rate, Called Count

Panel: Thrown Stacktraces

Anton Scherer

Chapter 6

Current state

This chapter describes the state of Kieker and its WebGUI.

6.1 Kieker Backend

This section describes the state of Kieker before the project. The focus lies on properties of Kieker that are modified by Kiekeriki-Backend.

6.1.1 Kieker

The current stable version of Kieker is 1.8. It can be downloaded from the homepage of Kieker[1]. It is also available via Maven and Git.
Detailed information can be found in the documentation section of the Kieker homepage[2]. Some general information, about aspects of Kieker concerning this project, are provided in the following sections.

6.1.2 Configuration

Any implementation of *AbstractFilterPlugin* can be configured during the construction of the pipes and filter structure.
The *@Property* annotation within the *@Plugin* annotation allows to define plugin specific properties including a default value. A *Configuration* item is mandatory to construct a plugin. The logic to use the configuration is implemented with every AbstractFilterPlugin subclass.
After the controller is started with its run() method, there is no way to modify the current configuration of any plugin already constructed.

6.1.3 Analysis

Kiekers analysis component consists mainly of the possibility to construct a pipes-and-filter architecture composed of various readers and filters. Readers and filters are all registered to

[1]http://Kieker-monitoring.net/download
[2]http://Kieker-monitoring.net/documentation/

and connected by an instance of *IAnalysisController*. Readers mark the "beginning" of any pipes-and-filters structure. Every filter to be used needs to be associated with a controller. The connections within any structure are realized through the annotations. The *@InputPort* is set to a method that receives incoming messages. The *@OutputPort* annotation is set within *@Plugin* and defines Outputports to which other plugins can connect. The connection is ensured by the controller.
Every new plugin has to be instantiated seperately and then connected by the controller to form a pipes-and-filters-structure. There is no way of predefining "substructures".

6.1.4 ΘPAD

ΘPAD, introduced in 2.1, is provided by the *opad-tslib-integration* branch of the official Kieker git repository. ΘPAD is provided via its corresponding classes. How ΘPAD can be used is shown by the ΘPADx-Demo which can be downloaded from the Kieker wiki[3].

6.1.5 RanCorr

RanCorr, see 2.1.2, is currently not supported in any way by Kieker.

Markus Fischer

6.2 Kieker Frontend

The Kieker WebGUI which should be enhanced is the stable version 1.8 [4]. The enhancement does not need all functionality provided by the original implementation. Thus the following classes or packages are deleted:

- kieker.webgui.web.beans.view.CockpitBean.java
- kieker.webgui.web.beans.view.CockpitEditorBean.java
- kieker.webgui.web.utility.CockpitLayout.java
- java.main.webapp.pages.CockpitEditorPage.xhtml
- java.main.webapp.pages.CockpitPage.xhtml
- java.main.webapp.pages.ControllerPage.xhtml
- java.main.webapp.dialogs.CockpitEditorPageDialogs.xhtml
- java.main.webapp.dialogs.ControllerPageDialogs.xhtml
- java.main.webapp.css.CockpitEditorPage.css
- java.main.webapp.css.CockpitPage.css
- java.main.webapp.css.ControllerPage.css

The following classes are edited, because they use deleted classes:

[3]https://Kieker.uni-kiel.de/trac/wiki/opad
[4]`https://sourceforge.net/projects/kieker/files/kieker/kieker-1.8/kieker-1.8_sources.zip/download`

- kieker.webgui.common:
  removed unused annotations
- kieker.webgui.common:
  removed most of converters (except getSuitableAnnotation and function isProgrammaticOnly)
- kieker.webgui.service.IProjectService:
  removed everything which is related to cockpit
- kieker.webgui.service.impl.ProjectService:
  removed everything which is related to cockpit
- kieker.webgui.web.beans.applicaton.GlobalProperties.Bean.java:
  Removed everything related to deleted exceptions
- kieker.webgui.web.beans.view.ControllerBean.java:
  removed everything related to cockpit
- java.main.webapp.WEB-INFO.faces-config.xml:
  Removed all unused pages
- java.main.webapp.templates.PagesTemplate.xhtml:
  Removed all buttons and dialogs related to cockpit
- java.main.webapp.dialogs.SettingsDialog.xhtml:
  Removed all buttons and dialogues related to cockpit

Martin Scholz, Christian Endres

Chapter 7

Implementation

7.1 Backend

7.1.1 Extension of Kieker

This section describes new (abstract) classes and components added directly to the architecture of Kieker. This includes the `AbstractCompositeFilterPlugin` which allows the composition of filters to one plugin, the `AbstractUpdateableFilterPlugin` which allows configuration of a plugin during runtime and the `ConfigurationRegistry` which provides means to configure Kiekerplugins externally.

AbstractCompositeFilterPlugin

The class `AbstractCompositeFilterPlugin` is an abstract class that enables the predefinition of a pipes-and-filter structure. E. g., a predefined pipes-and-filter structure can be used by only connecting the implementation of the `AbstractCompositeFilterPlugin` to a `IAnalysisController`. The class `CompositeFilterPluginExample` is an exemplary implementation of `AbstractCompositeFilterPlugin` using one TeeFilter.

- **Instantiation**
  As any other subclass of `AbstractFilterPlugin`, the `AbstractCompositeFilterPlugin` has to be instantiated with a `Configuration` and a `IProjectContext` (AnalysisController).
- **Configuration of inner plugins**
  The `AbstractCompositeFilterPlugin` allows the user to write entries in its Configuration destined for use to update the configuration of inner plugins before they are instantiated. Listing 7.1 shows how a the config for a TeeFilter is created and updated. the update method `updateConfiguration(final Configuration config, final Class<? extends AbstractFilterPlugin> clazz)` scans the composites configuration for entries where the simple class name of TeeFilter is present. E. g., an entry with key ”TeeFilter.STDlog” will be used to update the ”STDlog” property of the given configuration.

```java
public CompositeFilterPluginExample(Configuration configuration,
    IProjectContext projectContext) {
  super(configuration, projectContext);

  // create Filters to be used in this composite filter
  Configuration teeConfig = new Configuration();

  // update the configuration with values from the composites
     configuration
  this.updateConfiguration(teeConfig, TeeFilter.class);
  TeeFilter tee = new TeeFilter(teeConfig, this.controller);
```

Listing 7.1. Updating the configuration of inner plugins.

- **Connecting the plugins**
  The `AbstractCompositeFilterPlugin` has two already integrated plugins: The `CompositeInputRelay` and the `CompositeOutputRelay`. Both are subclasses of `AbstractFilterPlugin`.

  The **CompositeInputRelay** is responsible to relay incoming messages to its outputport, which has to be connected to the receiving inner plugins. An example can be seen in listing 7.2.

  The **CompositeOutputRelay** is responsible for delivering messages to the outputports of the `AbstractCompositeFilterPlugin` dependent on their class. E. g., if three different outputports (of the AbstractCompositeFilterPlugin) support for example the `IMonitoringRecord`, and an inner plugin delivers to its `IMonitoringRecord` compatible outputport (which is connected to the inputport of the `CompositeOutputRelay`), the sent record will be delivered to all three of the `AbstractCompositeFilterPlugins` outputports.

```java
  @InputPort(name = CompositeFilterPluginExample.INPUT_PORT,
     eventTypes = { IMonitoringRecord.class })
  public void startAnalysis(final IMonitoringRecord
     monitoringRecord) {
    this.inputRelay.relayMessage(monitoringRecord);
  }
```

Listing 7.2. Relaying a message at the `AbstractCompositeFilterPlugins` inputport.

The inner plugins can be connected in the Kieker customary fashion. Important is that all plugins that need input from outside the `AbstractCompositeFilterPlugin` are connected to the `CompositeInputRelay` and that all plugins that want to deliver outside of the `AbstractCompositeFilterPlugin` are connected to the `CompositeOutputRelay`. Example 7.3 shows a connection from the `CompositeInputRelay` to a TeeFilter to the `CompositeOutputRelay`.

```
// connect InputRelay to TeeFilter
this.controller.connect(inputRelay, CompositeInputRelay.
    INPUTRELAY_OUTPUTPORT, tee, TeeFilter.INPUT_PORT_NAME_EVENTS)
    ;
// connect TeeFilter to OutputRelay
this.controller.connect(tee, TeeFilter.
    OUTPUT_PORT_NAME_RELAYED_EVENTS, outputRelay,
    CompositeOutputRelay.INPUT_PORT_NAME_EVENTS);
```

Listing 7.3. Example of connecting the plugins within the `AbstractCompositePluginFilter`.

- **Limitations**
  - Currently it is only possible to receive Messages of class `IMonitoringRecord` since the CompositeInputRelay only supports this inputport.
  - Inner plugins of the same class can not be configured with different values for the same properties when using the `updateConfiguration(...)` method

Markus Fischer

AbstractUpdateableFilterPlugin

The class `AbstractUpdateableFilterPLugin` is an abstract class to enable the runtime configuration of any plugin. The `AbstractUpdateableFilterPLugin` extends the `AbstractFilterPlugin`. The class `UpdateableFilterPluginExample` is an exemplary implementation.

- **Properties**
  Listing 7.4 show how properties can be declared `"updateable = true"`. The default value is false. The boolean marks if a property is generally updateable during runtime.
- **Method setCurrentConfiguration(...)**
  The abstract method `setCurrentConfiguration(...)` is designated to update a plugins configuration during runtime. The boolean ”update” marks whether all properties are updated (update = false) or only the properties marked as updateable are reconfigured. An exemplary implementation is shown in listing 7.5.
- **Usage** To reconfigure plugins during runtime they have to be registered to a `ConfigurationRegistry`, see 7.1.1

```
configuration = {
  // This property should not be updated during runtime (updateable-default=false)
  @Property(name = UpdateableFilterPluginExample.NOT_UPDATEABLE_PROPERTY, defaultValue = UpdateableFilterPluginExample.NOT_UPDATEABLE_PROPERTY_DEFAULT),
  // This property may be updated during runtime
  @Property(name = UpdateableFilterPluginExample.UPDATEABLE_PROPERTY, defaultValue = UpdateableFilterPluginExample.UPDATEABLE_PROPERTY_DEFAULT,
       updateable = true) }
```

Listing 7.4. Defining updateable properties.

```
public void setCurrentConfiguration(final Configuration config, final boolean update) {
  // exemplary implementation
  if (update) {
    // update only properties that are updateable given configuration items
    for (final Entry<?, ?> e : config.entrySet()) {
      if (config.containsKey(e.getKey()) && isPropertyUpdateable((String) e.getKey())) {
        this.configuration.setProperty((String) e.getKey(), (String) e.getValue());
      }
    }
  } else {
    // update all properties whether they are updateable or not.
    for (final Entry<?, ?> e : config.entrySet()) {
      this.configuration.setProperty((String) e.getKey(), (String) e.getValue());
    }
  }
}
```

Listing 7.5. Example implementation of setCurrentConfiguration().

Markus Fischer

ConfigurationRegistry

The `ConfigurationRegistry` provides functionality to register instances of `AbstractUpdateableFilterPlugin` and update their configuration during runtime.
The ConfigurationRegistry consists of the abstract class `AbstractConfigurationRegistry`, the interface `IConfigurationRegistry` and the `GlobalConfigurationRegistry` which is a singleton.

- **Register a plugin**
  Any AbstractUpdateableFilterPlugin can be registered to a `ConfigurationRegistry`. The `GlobalConfigurationRegistry` provides a globally visible and usable ConfigurationRegistry within Kieker. To register a plugin it needs a globally (within the scope of Kieker) unique String identifier, see example listing 7.6.

```
// register a plugin
GlobalConfigurationRegistry.getInstance().
    registerUpdateableFilterPlugin("uniqueID", updateablePlugin);

// update a plugin
Configuration conf = new Configuration();
conf.setProperty("propertyName", "newValue");
GlobalConfigurationRegistry.getInstance().updateConfiguration("
    uniqueID", conf, true);
```

Listing 7.6. Using the `GlobalCongigurationRegistry`.

- **Update a plugin**
  All registered plugins can be retrieved using the `GlobalCongigurationRegistry`. An example is shown in listing 7.6.
  If the given id is not present in the registry a `PluginNotFoundException` is thrown.

Markus Fischer

7.1.2 Implementation of RanCorr

This section describes the implementation of RanCorr (see chapter 2.1.2) in Kiekeriki. In the following sections this implementation is just called RanCorr.

Architecture Data Model

RanCorr needs its own data model because the data model of Kieker doesn't provide enough information about the monitored architecture and the calculated anomaly scores. Figure

7.1 shows the developed architecture data model. There is the abstract class `AbstractRanCorrItem` which represents any architecture component monitored by Kiekeriki. The implemented classes based on `AbstractRanCorrItem` are:

- `RanCorrHost` represents the host of the monitored system and decorates the class `ExecutionContainer` by Kieker. A host may contain several applications.
- `RanCorrApplication` represents the application which is monitored. This is a new component and was not considered by the original version of Kieker. An application may contain several packages.
- `RanCorrPackage` represents a Java package in the monitored application. Also the package is not in the original Kieker data model. Kieker knows packages only through the monitored operations which contain a package path. A package may contain several sub packages and classes.
- `RanCorrClass` represents a Java class and decorates the class `ComponentType` by Kieker. A class may contain several operations.
- `RanCorrOperation` represents a Java method and decorates the class `Operation` by Kieker. It may contain several anomaly scores and the calling dependencies to other operations.

Input/Output Model

The RanCorr algorithms, described detailed in the paragraphs below, need the class `RanCorrInput` as input, which contains the monitored operations with the according anomaly scores (see Figure 7.2). The input object also contains a reference to the class `DependencyManager` which contains the information about the calling dependencies of operations and the according weights. The RanCorr algorithm creates objects of the class `RanCorrOutput` which contain the mapping between architecture items and the calculated anomaly rankings. Additionally the output contains the information which algorithm was used to calculate the anomaly ranking, indicated by the enumeration `RanCorrAlgorithmEnum`. The `RanCorrInput` contains anomaly scores in range [-1.0, 1.0]. The `RanCorrOutput` contains anomaly rankings in range [0.0, 1.0].

Dependency Handling

The calling dependency handling is actually not considered in Kieker. Hence each `RanCorrOperation` contains direct references to operations which call this `RanCorrOperation` and are called by this `RanCorrOperation`. Additionally the class `DependencyManager` contains every dependency information as triple of two `RanCorrOperation` objects and the according weight. This information is used by the advanced RanCorr algorithm to prioritize the dependencies.

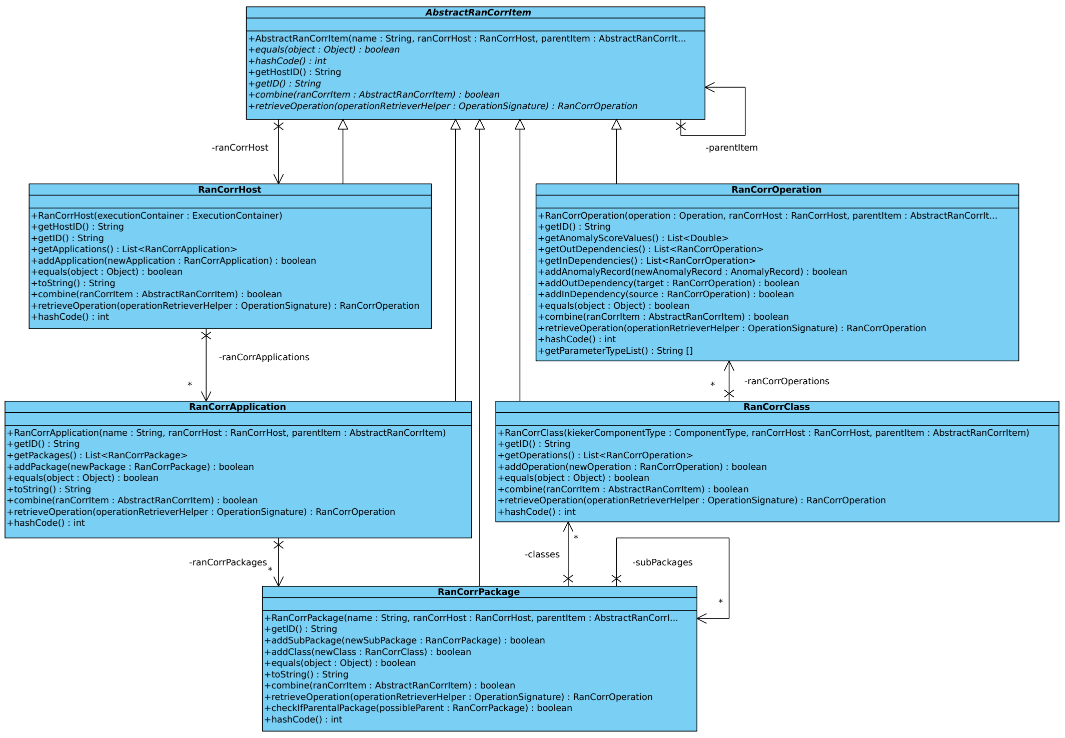

AbstractRanCorrItem
+AbstractRanCorrItem(name : String, ranCorrHost : RanCorrHost, parentItem : AbstractRanCorrIt...
+equals(object : Object) : boolean
+hashCode() : int
+getHostID() : String
+getID() : String
+combine(ranCorrItem : AbstractRanCorrItem) : boolean
+retrieveOperation(operationRetrieverHelper : OperationSignature) : RanCorrOperation
-ranCorrHost
-parentItem
RanCorrHost
+RanCorrHost(executionContainer : ExecutionContainer)
+getHostID() : String
+getID() : String
+getApplications() : List<RanCorrApplication>
+addApplication(newApplication : RanCorrApplication) : boolean
+equals(object : Object) : boolean
+toString() : String
+combine(ranCorrItem : AbstractRanCorrItem) : boolean
+retrieveOperation(operationRetrieverHelper : OperationSignature) : RanCorrOperation
+hashCode() : int
RanCorrOperation
+RanCorrOperation(operation : Operation, ranCorrHost : RanCorrHost, parentItem : AbstractRanCorrIt...
+getID() : String
+getAnomalyScoreValues() : List<Double>
+getOutDependencies() : List<RanCorrOperation>
+getInDependencies() : List<RanCorrOperation>
+addAnomalyRecord(newAnomalyRecord : AnomalyRecord) : boolean
+addOutDependency(target : RanCorrOperation) : boolean
+addInDependency(source : RanCorrOperation) : boolean
+equals(object : Object) : boolean
+combine(ranCorrItem : AbstractRanCorrItem) : boolean
+retrieveOperation(operationRetrieverHelper : OperationSignature) : RanCorrOperation
+hashCode() : int
+getParameterTypeList() : String []
-ranCorrApplications
*
*
-ranCorrOperations
RanCorrApplication
+RanCorrApplication(name : String, ranCorrHost : RanCorrHost, parentItem : AbstractRanCorrItem)
+getID() : String
+getPackages() : List<RanCorrPackage>
+addPackage(newPackage : RanCorrPackage) : boolean
+equals(object : Object) : boolean
+toString() : String
+combine(ranCorrItem : AbstractRanCorrItem) : boolean
+retrieveOperation(operationRetrieverHelper : OperationSignature) : RanCorrOperation
+hashCode() : int
RanCorrClass
+RanCorrClass(kiekerComponentType : ComponentType, ranCorrHost : RanCorrHost, parentItem : AbstractRanCorrItem)
+getID() : String
+getOperations() : List<RanCorrOperation>
+addOperation(newOperation : RanCorrOperation) : boolean
+equals(object : Object) : boolean
+combine(ranCorrItem : AbstractRanCorrItem) : boolean
+retrieveOperation(operationRetrieverHelper : OperationSignature) : RanCorrOperation
+hashCode() : int
-ranCorrPackages
*
-classes
*
-subPackages
*
RanCorrPackage
+RanCorrPackage(name : String, ranCorrHost : RanCorrHost, parentItem : AbstractRanCorrI...
+getID() : String
+addSubPackage(newSubPackage : RanCorrPackage) : boolean
+addClass(newClass : RanCorrClass) : boolean
+equals(object : Object) : boolean
+toString() : String
+combine(ranCorrItem : AbstractRanCorrItem) : boolean
+retrieveOperation(operationRetrieverHelper : OperationSignature) : RanCorrOperation
+checkIfParentalPackage(possibleParent : RanCorrPackage) : boolean
+hashCode() : int

Figure 7.1. Architecture data model of RanCorr implementation consisting of `RanCorrHost`, `RanCorrApplication`, `RanCorrPackage`, `RanCorrClass`, `RanCorrOperation`.

RanCorr Algorithms

RanCorr basically provides three different algorithms to calculate the anomaly ranking (see chapter 2.1.2). The general RanCorr algorithm is represented in the abstract class `AbstractRanCorrAlgorithm` (see Figure 7.3). The three different algorithms are implemented in the classes `TrivialAlgorithm`, `SimpleAlgorithm` and `AdvancedAlgorithm`. All of them implement the *calculate* method which starts the anomaly ranking calculation. First step is the calculation of anomaly rankings for all monitored operations. Then these values are aggregated for the different architecture levels: class, package, application and hosts. The three algorithms use different mean functions to aggregate the data. There are different power mean exponents for each architecture level defined as static attribute in the class `AdvancedAlgorithm`. The RanCorr algorithms work based on anomaly scores which must exist during calculation. We extended the algorithm by checks for empty lists of anomaly scores. In case of empty lists we return a `null` value for means and rankings.

There is an utility package for general, mathematical operations which contains the classes

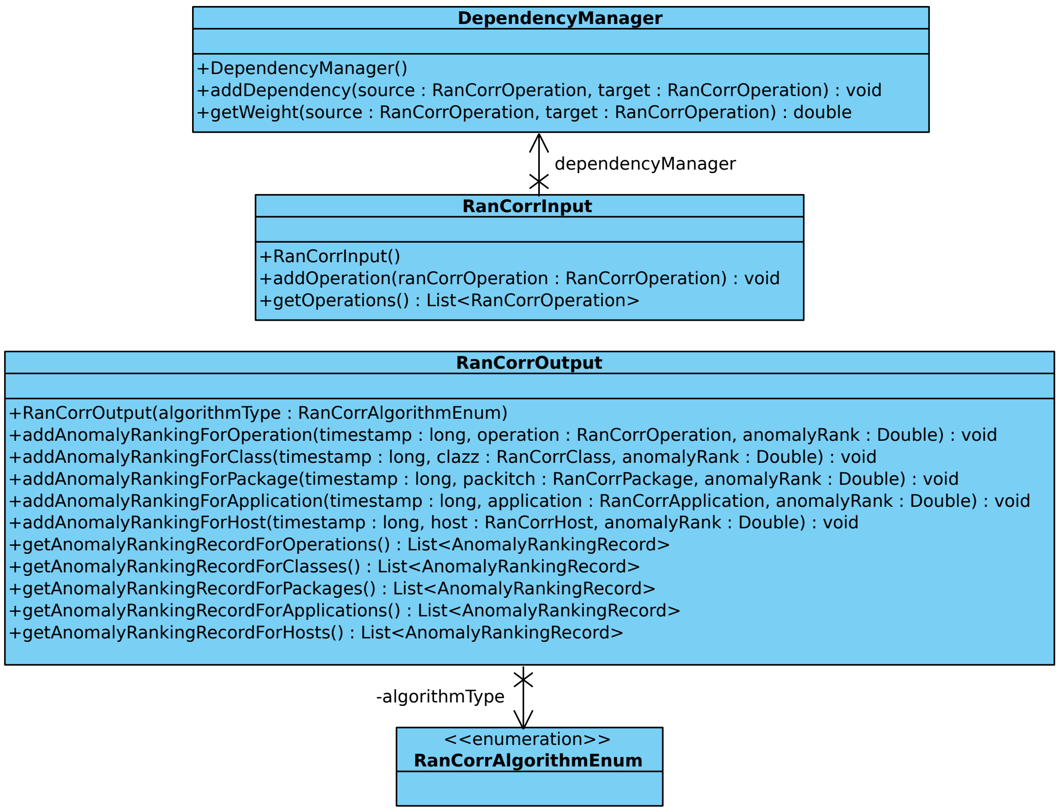

DependencyManager
+DependencyManager()
+addDependency(source : RanCorrOperation, target : RanCorrOperation) : void
+getWeight(source : RanCorrOperation, target : RanCorrOperation) : double
dependencyManager
RanCorrInput
+RanCorrInput()
+addOperation(ranCorrOperation : RanCorrOperation) : void
+getOperations() : List<RanCorrOperation>
RanCorrOutput
+RanCorrOutput(algorithmType : RanCorrAlgorithmEnum)
+addAnomalyRankingForOperation(timestamp : long, operation : RanCorrOperation, anomalyRank : Double) : void
+addAnomalyRankingForClass(timestamp : long, clazz : RanCorrClass, anomalyRank : Double) : void
+addAnomalyRankingForPackage(timestamp : long, packitch : RanCorrPackage, anomalyRank : Double) : void
+addAnomalyRankingForApplication(timestamp : long, application : RanCorrApplication, anomalyRank : Double) : void
+addAnomalyRankingForHost(timestamp : long, host : RanCorrHost, anomalyRank : Double) : void
+getAnomalyRankingRecordForOperations() : List<AnomalyRankingRecord>
+getAnomalyRankingRecordForClasses() : List<AnomalyRankingRecord>
+getAnomalyRankingRecordForPackages() : List<AnomalyRankingRecord>
+getAnomalyRankingRecordForApplications() : List<AnomalyRankingRecord>
+getAnomalyRankingRecordForHosts() : List<AnomalyRankingRecord>
-algorithmType
<<enumeration>>
RanCorrAlgorithmEnum

Figure 7.2. Data model for the input and output of RanCorr. The class `RanCorrInput` represents the input model. The class `DependencyManager` is used in the input to get the calling dependency information between operations. The class `RanCorrOutput` represents the output of RanCorr and contains the mapping between architecture item and the calculated anomaly ranking. The `RanCorrAlgorithmEnum` defines which type of algorithm was used.

`Maths` and `TransitiveClosureUtil` (see Figure 7.4). Each algorithm uses the class `Maths` to calculate different means of anomaly scores. The advanced algorithm uses additionally the class `TransitiveClosureUtil` to calculate the forward and backward transitive closure of calling dependencies. The transitive closure is calculated by a depth-first graph search.

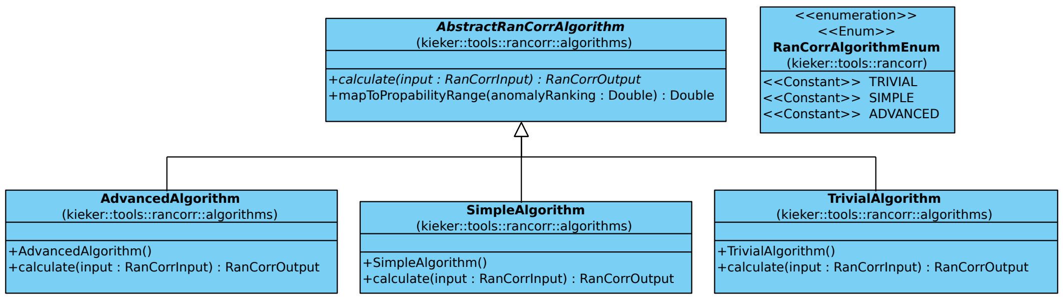

AbstractRanCorrAlgorithm
(kieker::tools::rancorr::algorithms)
+calculate(input : RanCorrInput) : RanCorrOutput
+mapToPropabilityRange(anomalyRanking : Double) : Double
<<enumeration>>
<<Enum>>
RanCorrAlgorithmEnum
(kieker::tools::rancorr)
<<Constant>> TRIVIAL
<<Constant>> SIMPLE
<<Constant>> ADVANCED
AdvancedAlgorithm
(kieker::tools::rancorr::algorithms)
+AdvancedAlgorithm()
+calculate(input : RanCorrInput) : RanCorrOutput
SimpleAlgorithm
(kieker::tools::rancorr::algorithms)
+SimpleAlgorithm()
+calculate(input : RanCorrInput) : RanCorrOutput
TrivialAlgorithm
(kieker::tools::rancorr::algorithms)
+TrivialAlgorithm()
+calculate(input : RanCorrInput) : RanCorrOutput

Figure 7.3. The three types of RanCorr algorithms are implemented in the classes `TrivialAlgorithm`, `SimpleAlgorithm` and `AdvancedAlgorithm` which all extend the abstract class `AbstractRanCorrAlgorithm`. The enumeration `RanCorrAlgorithmEnum` can be used to identify them.

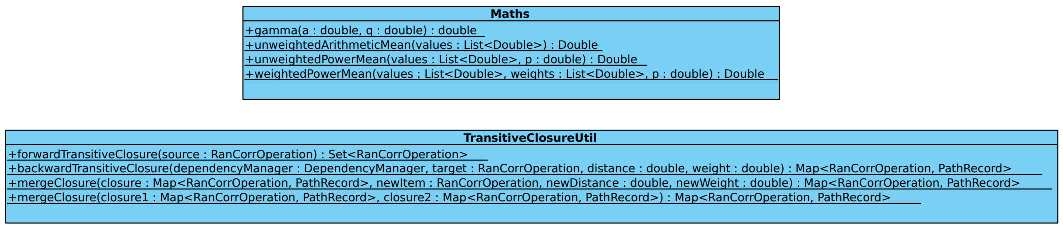

Maths
+gamma(a : double, q : double) : double
+unweightedArithmeticMean(values : List<Double>) : Double
+unweightedPowerMean(values : List<Double>, p : double) : Double
+weightedPowerMean(values : List<Double>, weights : List<Double>, p : double) : Double
TransitiveClosureUtil
+forwardTransitiveClosure(source : RanCorrOperation) : Set<RanCorrOperation>
+backwardTransitiveClosure(dependencyManager : DependencyManager, target : RanCorrOperation, distance : double, weight : double) : Map<RanCorrOperation, PathRecord>
+mergeClosure(closure : Map<RanCorrOperation, PathRecord>, newItem : RanCorrOperation, newDistance : double, newWeight : double) : Map<RanCorrOperation, PathRecord>
+mergeClosure(closure1 : Map<RanCorrOperation, PathRecord>, closure2 : Map<RanCorrOperation, PathRecord>) : Map<RanCorrOperation, PathRecord>

Figure 7.4. The RanCorr algorithms use the util classes to calculate the anomaly rankings. The class `Maths` contains math functions to calculate different means and the class `TransitiveClosureUtil` contains functions to calculate transitive closures of calling dependencies.

Integration into Kieker

In order to integrate RanCorr in the Kieker framework we built a new filter which starts the RanCorr calculation. The class `RanCorrFilter` represents that filter (see Figure 7.5). This filter has two input ports: one for the objects of class `MessageTrace` and one for objects of class `OperationAnomalyInfo`. The first port is used to read the message traces and to rebuild the architecture model of the monitored system. We achieve this by using the method *convert* of the class `RanCorrInputProvider`. This method analyze the messages in the given message trace and rebuilds the architecture with all elements. Finally it returns an object of the class `RanCorrInput` with which the RanCorr algorithm is started. The second port is used to get the anomaly score information for the several operations. This will be typically the output of ΘPADx. The range of that input values should be in range [0.0, 1.0]. The type of the used RanCorr algorithm can be configured via the filter configuration. There is also an additional filter represented by the class `RanCorrOutputDBWriter` which is used to write the results of RanCorr in the exchange database. This filter has one input port for objects of the class `RanCorrOutput`. The interface `IDatabaseConnection` is used to store the values in the database.

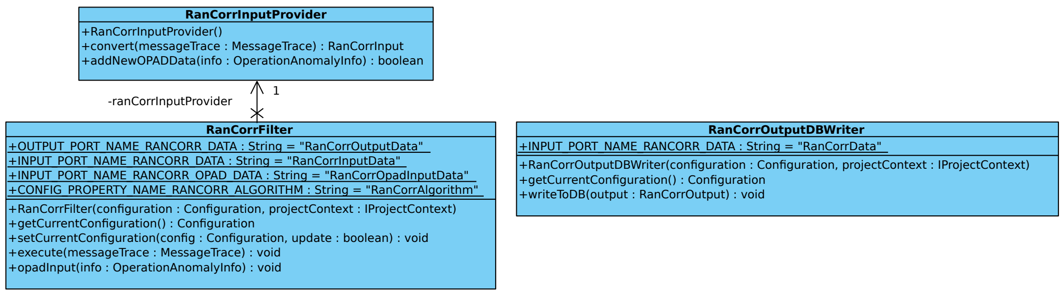

RanCorrInputProvider
+RanCorrInputProvider()
+convert(messageTrace : MessageTrace) : RanCorrInput
+addNewOPADData(info : OperationAnomalyInfo) : boolean
-ranCorrInputProvider
1
RanCorrFilter
+OUTPUT_PORT_NAME_RANCORR_DATA : String = "RanCorrOutputData"
+INPUT_PORT_NAME_RANCORR_DATA : String = "RanCorrInputData"
+INPUT_PORT_NAME_RANCORR_OPAD_DATA : String = "RanCorrOpadInputData"
+CONFIG_PROPERTY_NAME_RANCORR_ALGORITHM : String = "RanCorrAlgorithm"
+RanCorrFilter(configuration : Configuration, projectContext : IProjectContext)
+getCurrentConfiguration() : Configuration
+setCurrentConfiguration(config : Configuration, update : boolean) : void
+execute(messageTrace : MessageTrace) : void
+opadInput(info : OperationAnomalyInfo) : void
RanCorrOutputDBWriter
+INPUT_PORT_NAME_RANCORR_DATA : String = "RanCorrData"
+RanCorrOutputDBWriter(configuration : Configuration, projectContext : IProjectContext)
+getCurrentConfiguration() : Configuration
+writeToDB(output : RanCorrOutput) : void

Figure 7.5. The integration of RanCorr into Kieker is realized by a new filter implemented by the class `RanCorrFilter`. There is also the class `RanCorrInputProvider` which creates the input for the RanCorr algorithms. The output of RanCorr is written to the database using the class `RanCorrOutputDBWriter` which represents the writer filter.

Yannic Noller

7.1.3 Extended Self Tuning Forecasting Filter

In the project Kikeriki a new filter, the `ExtendedSelfTuningForecastingFilter`, was developed for the ΘPADx approach. The filter searches for the best forecasting method and calculates the forecasting value which should be the best forecasting value at that point of time. Basics for the development of the `ExtendedSelfTuningForecastingFilter` were given by the extended forecasting filter developed by Frotscher [2013], a filter created for ΘPADx and the WCF (Workload Classification and Forecasting) by Herbst [2012]. The foundations of ΘPADx are described in Section 2.1. In this chapter WCF (Workload Classification and Forecasting) is introduced, then the integration of different filters will be discussed and later the development of the new filter by merging WCF into the the `ExtendedForecastingFilter` will be described. The newly created filter is called `ExtendedSelfTuningForecastingFilter`.

WCF

WCF is the main component of a master's thesis written by Nikolas Herbst [Herbst, 2012]. In his thesis he describes an algorithm which switches between forecast algorithms automatically during run-time to get the best forecasting result. The WCF is based on a decision tree. After checking some decision parameters like overhead of an forecasting algorithm, WCF chooses two forecasting algorithms which might be the best algorithm at that point in time. After calculating forecasting values with the two forecast algorithms, the results are compared and the better result is chosen by WCF. For a deeper understanding of WCF it is recommended to read Herbst's thesis [Herbst, 2012].

An advantage of using WCF in Kikeriki is, there is an implementation which could be used in our project. The requirement of the project to be able to choose the best forecast algorithm during run-time could be solved with the approach of Herbst's thesis. Another advantage of using WCF is, that in the WCF implementation were some forecasting algorithms implemented already, which were not implemented in ΘPADx/Kieker yet. Because the algorithm methods were implemented similarly, they could be easily reused within our project. As a result the requirement to add more forecasting algorithms into ΘPADx could be solved.

Integration of forecasting methods

In ΘPADx there are different forecasting algorithms implemented. During the integration of WCF more forecasting methods were added. The new added forecasting methods include the ARIMA, Croston, SES, Naive and TBATS forecasting method. Two forecasting methods are not in the Extended Self Tuning Forecasting Filter, which were used in the Extended Forecasting Filter: The SESR and the ARIMA101 forecasting method. The SESR is replaced by the SES forecasting method because the SESR forecasting implementation was not written in pure R code. ARIMA101 was replaced by the ARIMA forecasting method because at some special time series ARIMA101 is not applicable. During testing, we found an error in the R implementation of ARIMA. So it may occur that the ARIMA forecasting method does not work as expected. Fortunately the WCF algorithm always uses two forecasting methods so if ARIMA does not work, the second forecasting method is used. Also

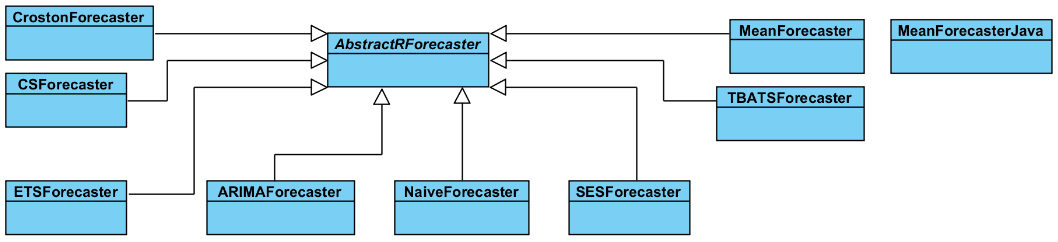


Figure 7.6. Forecasting filters

all forecasting methods not implemented in R were replaced by R forecasting methods. Only the Mean forecasting method is, besides the R method, written in Java, is part of the `ExtendedSelfTuningForecastingFilter` which can be used for forecasting. The java method should only be used for debugging purposes. The advantage of the Java method is, that the R environment is not needed.

Implementation of the ExtendedSelfTuningForecastingFilter

As mentioned before the `ExtendedSelfTuningForecastingFilter` is based on the implementation of WCF. The main difference between the original WCF implementation and the integration of WCF into the new `ExtendedSelfTuningForecastingFilter` is the data processing. In the approach of Herbst [2012] the input data is read form a xls fil. For the ΘPADx filter environment, the data is sent to the forecasting filter whenever data is read by the Kieker reader and send to the ΘPADx filter architecture. A condensed sequence diagram of the `ExtendedSelfTuningForecastingFilter` is shown in the diagram below. The sequence diagram shows the flow if self-tuning is activated. The mechanic of the `ExtendedSelfTuningForecastingFilter` is shown in the documentation of WCF [Herbst, 2012].

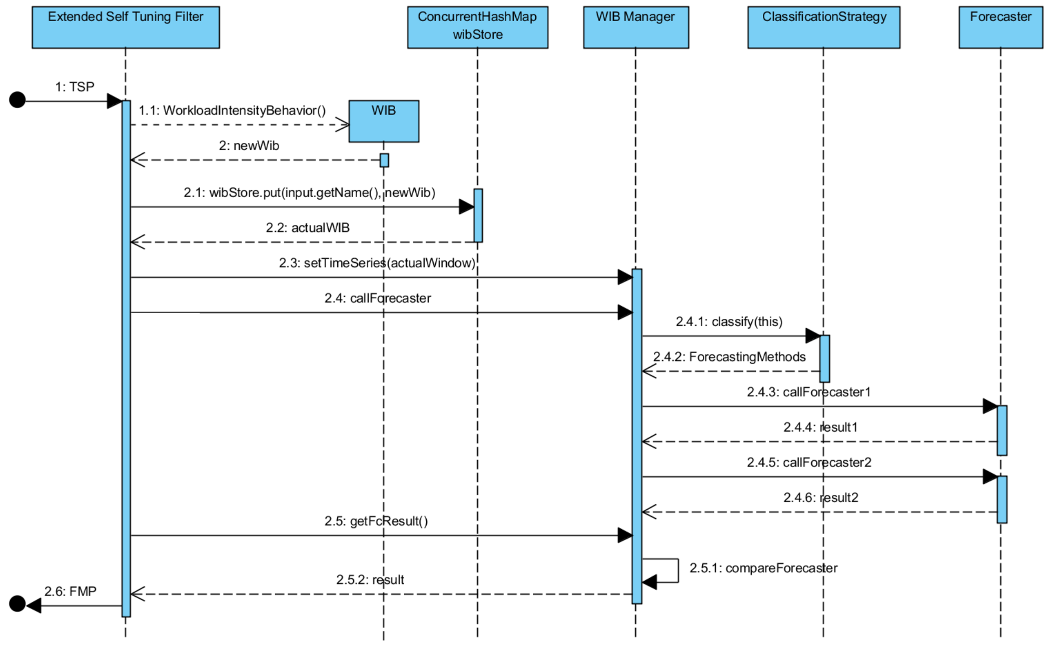


Figure 7.7. Sequence diagram of the extended self tuning forecasting filter.

As it can be seen in Figure 7.7 the `ExtendedSelfTuningForecastingFilter` receives a `Time Series Point` (TSP)from another filter as input data. For each TSP received on the input port of the `ExtendedSelfTuningForecastingFilter`, a new `WorkloadIntensityBehavior` (WIB) is created. The created WIB is stored in a ConcurrentHashMap. In the hash map the name of the called function is the key of the hash map. So every function is forecasted on its own. In the listing below the creation of the hash map and the storage of the object is shown.

```
    if (this.selftuningbool.get()) {
      ...
      final WorkloadIntensityBehavior newWib = new
          WorkloadIntensityBehavior(...);
      this.wibStore.put(input.getName(), newWib);
    }
```

Listing 7.7. Create WIB object and store the object into the wibStore hashmap.

Next the incoming TSP is added to the time series. With the function `callForecaster()` the forecasting is started. First of all the time series is classify into a category to find the best forecasting methods. The possible categories are initial, fast and complex. The procedure is based on some decision points like length of the time series. How the classification exactly works is described in [Herbst, 2012]. Compared to the implementation of Herbst, some decision points which do not fit into the ΘPADx/Kieker environment are not used in the implementation of the `ExtendedSelfTuningForecastingFilter`. Decision points like for example amount of skipped values are not used anymore. After the classification is finished, the two forecasting methods, which were chosen by the classification function, are called. These two forecasting methods calculate the forecasting value. After the two results have been compared, the better result is sent within a ForecastMeasurementPair (FSP) to the `ExtendedSelfTuningForecastingFilter` output port. For the possibility to change between self tuning mode and an forecasting method chosen by the user, there is the filter property `PROPERTY_SELFTUNING_ON` implemented. If the boolean property is set to true, the filter runs in self tuning mode. Otherwise the filter can be used as the `ExtendedForecastingFilter` developed by [Herbst, 2012].

Tobias Rudolph

7.2 Frontend

In this section we present the design and implementation details for the given frontend architecture. The components Logic, Web GUI, Cache and the internal database are described in separate subsection as well as the Transfer Database and the Data Generator which writes synthesized data into the Transfer Database and provides benchmarking functionality.

7.2.1 Web GUI

The Web GUI was developed based on a sketch that was created in the first sprint of the development project. Ideas for the sketch were based on the evaluated APM Tools currently on the market (See chapter 2.4). Figure 7.8 shows the main features the frontend should offer in the main page, the *Project View Page*. One requirement was to display the aggregated anomaly information from the RanCorr approach (See chapter 2.1). Therefore the sketch suggests an *Architecture Overview*, that shows the different components in an application system. Additionally, to show high level components like servers and databases, a deep-dive approach is intended. The approach should allow to inspect the internals of the high level components by showing instrumented applications, their classes and packages. For the requirement to be able to monitor different metrics on the instrumented components, the sketch contains a section for diagrams, the *Diagrams View*. The diagrams are configurable representations for metrics and components. Additionally to display diagrams was to add new diagrams which visualize the user-defined metrics, the suggestion was a dialog to enable this. Another integral part of the Project View Page is the presentation of important events, e.g. some instrumented component doesn't respond anymore, some anomaly was detected on a component. For this purpose the *Tab View* was designed to show the user all events that might happen with the application. The metrics are predefined by the user on the available monitored instrumentation records and applied on a specific component by a diagram. Components here can be Hosts, Applications, Packages, Classes and Operations. The definition of metrics are handled in the *Settings Page*. The sketch additionaly suggests a *List View* for all aspects of the monitored system, to e.g. look into packages and their hierachy.

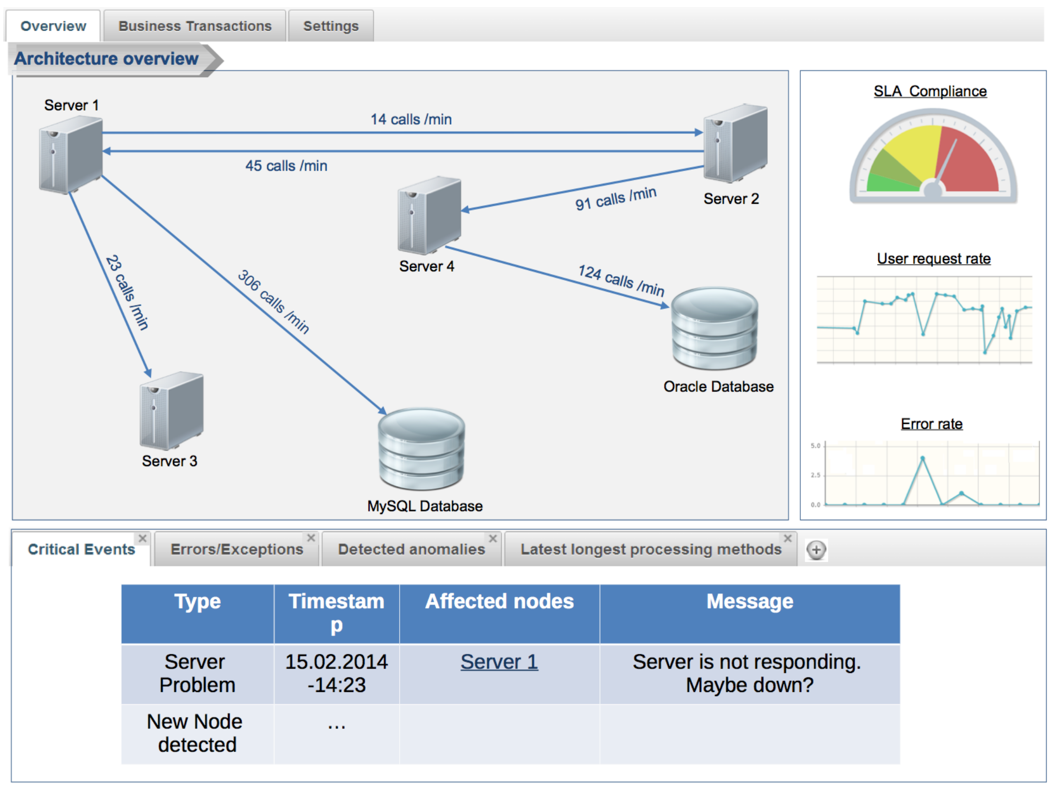
A screenshot of the final implementation is depicted in Figure 7.9. The Project View Page consists of implementations for the Architecture Overview, Tab View, Diagrams View and Dialog. The Settings Page was implemented as a single component.

The Architecture Overview (See 7.10) consists of 2 bean classes: `ArchitectureGraph-Bean` and `ArchitectureMenuBean`. The `ArchitectureGraphBean` manages the transition of the displayed graph in the UI, while the `ArchitectureMenuBean` allows the user to revert the transitions. The visible graph in the UI is rendered and layouted by the graph layout library Springy.js Spr. A user transition (a deep-dive into one component) is handled by the `ArchitectureGraphBean`, which loads the new components into the UI and reloads the Diagrams View.

The Diagrams View (See 7.11) is implemented with a bean class, the `DiagramDataBean`, and a JavaScript Component (`diagram.js`) in the Project View Page. The `DiagramDataBean` is responsible for adding new diagrams into the GUI, by sending diagram configurations to the `diagram.js` component. Another responsibility is to allow the component to poll live monitoring data. At last the component renders the diagrams with the JavaScript library Highcharts JS Hig and handles all incoming requests to delete, add or configure diagrams.

The Settings Page (See 7.12) was implemented in a single component that retrieves its data directly from the `DefinedMetricsBean`, which is responsible for getting the defined metrics of the current user. The `RawDataBean` is used to show the user the available data types on a component.

The Tab View (See 7.13), in the final state, was implemented as single component which displays exceptions of the monitored system. The `ExceptionEventsTab` uses the `ExceptionEventsBean` to get the latest exceptions. The bean gets its data trough the



Type	Timestamp	Affected nodes	Message
Server Problem	15.02.2014 -14:23	Server 1	Server is not responding. Maybe down?
New Node detected	...		

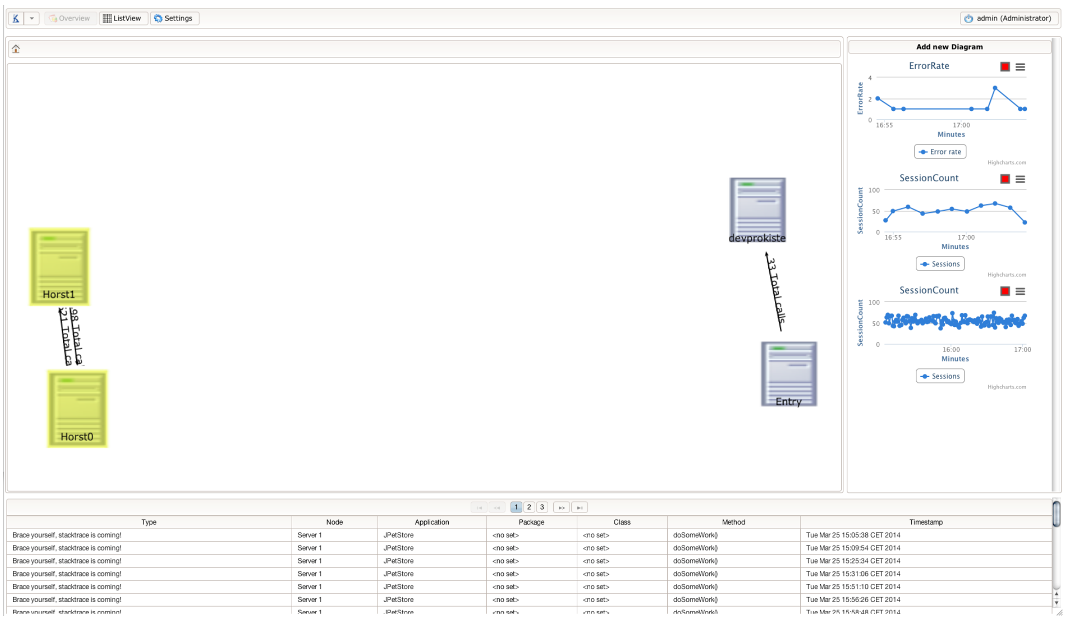
Figure 7.8. Design sketch of the Web GUI

`ArchitectureMetaStore`, which is the cache in the system.

The List View (See 7.16) was implemented as a single page and bean. The page displays the hierarchy of the monitored system in a tree, for that the bean is the interface to the data.

7.2.2 Logic

The main parts of the Logic are seperated into four classes: `ArchitectureGraphBean`, `DefinedMetricsBean`, `DiagramDataBean` and `MetricsDiagramBean` (See Figure 7.15). The `ArchitectureGraphBean` is responsible for the Architecture Overview, which consists of sending the nodes and edges of the current view with their associated RanCorr Anomaly Rank. Additionally the bean is responsible for keeping the state of the Project View Page, by setting the correct state in the `ArchitectureMetaStore`. `DefinedMetricsBean` is a thin logical layer above the IMetricService, it contains logic for the Settings Page and additionally to generate default metrics for the user. Logical parts to render appropiate diagrams for



Type	Node	Application	Package	Class	Method	Timestamp
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:05:38 CET 2014
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:09:54 CET 2014
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:25:34 CET 2014
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:31:06 CET 2014
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:51:10 CET 2014
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:56:26 CET 2014
Brace yourself, stacktrace is coming!	Server 1	JPetStore	<no set>	<no set>	doSomeWork()	Tue Mar 25 15:58:48 CET 2014

Figure 7.9. Design sketch of the Web GUI

the user is handled by the `DiagramDataBean`. The `DiagramDataBean` is also responsible for transforming the data inside the Exchange Database into Highcharts JS data series, for past and live data. `MetricsDiagramBean` is the interface for the dialog to add/configure diagrams. Its functionality consists of fetching available metrics for the current user, suggest a diagram type based on a selected metric, configure whether to load past or live data.

7.2.3 Cache

The Cache (See 7.16) for the Exchange Database is a class using the singleton pattern, the `ArchitectureMetaStore`. As the role as a cache implies, it caches the needed data series for the diagrams and the components in UI. The cache gets and stores the data from the database on request, the data is not kept persistent in the system, only temporarily for some fixed amount of time. This is needed as the size of all monitored records drastically decreases the performance of the Web Server.

7.2.4 Internal Database

The Internal Database (See 7.17) is a local Derby database which stores User, Metrics and Diagram data. It allows the user to store his own metrics in the system, which can be used for diagrams. Every user has his own metrics and diagrams. The User table originates

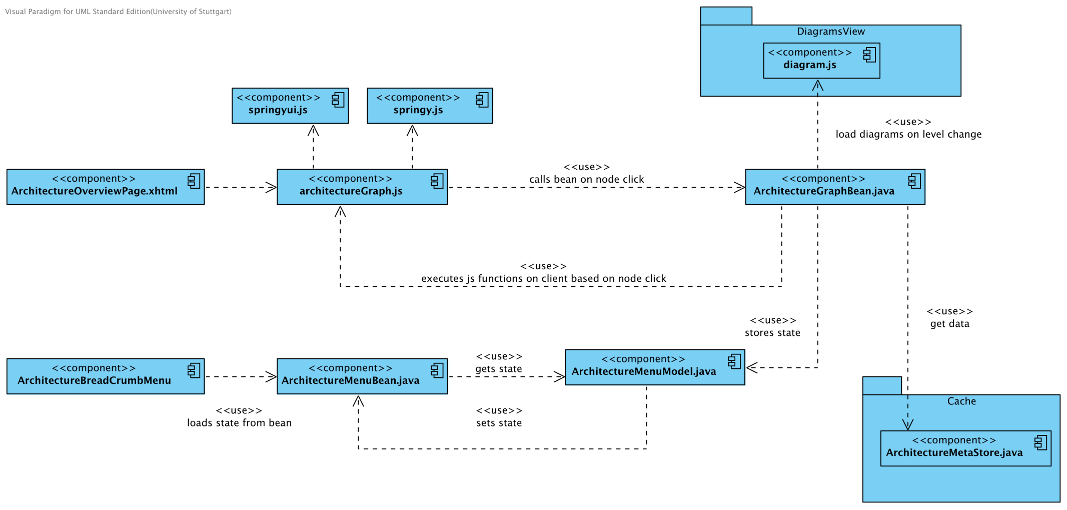

Visual Paradigm for UML Standard Edition(University of Stuttgart)
DiagramsView
<<component>> diagram.js
<<component>> springyui.js
<<component>> springy.js
<<use>> load diagrams on level change
<<component>> ArchitectureOverviewPage.xhtml
<<component>> architectureGraph.js
<<use>> calls bean on node click
<<component>> ArchitectureGraphBean.java
<<use>> executes js functions on client based on node click
<<use>> stores state
<<use>> get data
<<component>> ArchitectureBreadCrumbMenu
<<component>> ArchitectureMenuBean.java
<<use>> gets state
<<component>> ArchitectureMenuModel.java
<<use>> loads state from bean
<<use>> sets state
Cache
<<component>> ArchitectureMetaStore.java

Figure 7.10. Component Diagram of the Architecture Overview

from the original Kieker Web GUI. The new tables are Metrics and Diagrams, the first contains columns that specify which records (rawData) are used within the metric and what functions should be applied on the records from the exchange database. The Diagrams table contains simple information such as the title of the diagram and x/y-axis names. Additionally information about the component, the metric and whether to show past or live data is stored in the table.

Kálmán Képes

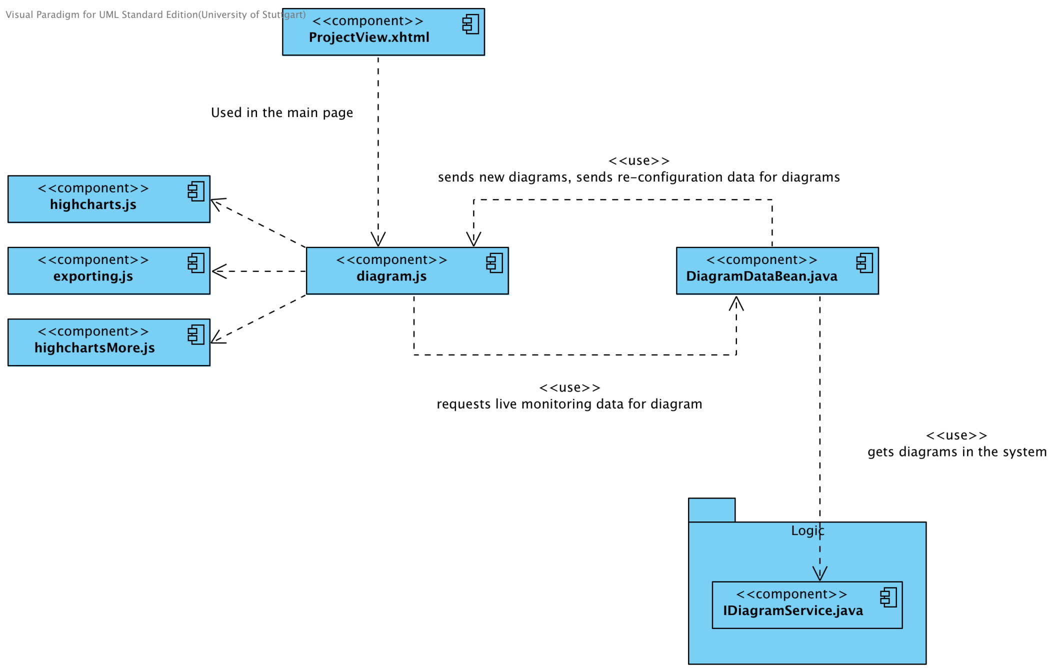


Figure 7.11. Component Diagram of the Diagrams View

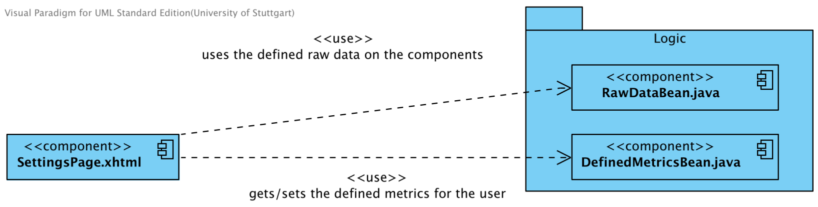

Visual Paradigm for UML Standard Edition(University of Stuttgart)
<<use>>
uses the defined raw data on the components
Logic
<<component>>
RawDataBean.java
<<component>>
SettingsPage.xhtml
<<component>>
DefinedMetricsBean.java
<<use>>
gets/sets the defined metrics for the user

Figure 7.12. Component Diagram of the Settings View

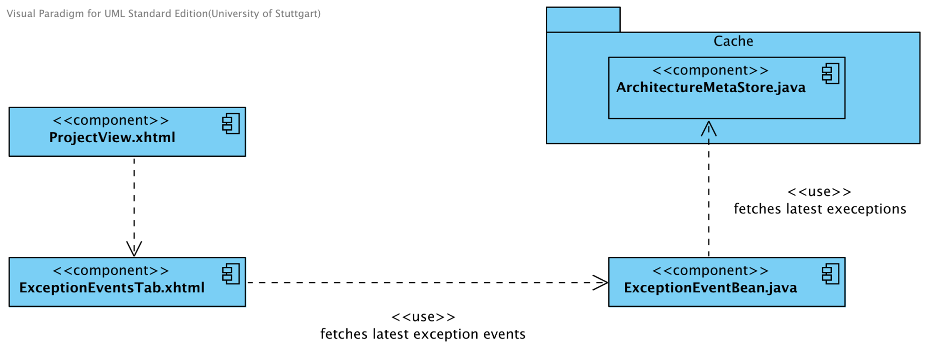

Visual Paradigm for UML Standard Edition(University of Stuttgart)
Cache
<<component>>
ArchitectureMetaStore.java
<<component>>
ProjectView.xhtml
<<use>>
fetches latest execeptions
<<component>>
ExceptionEventsTab.xhtml
<<component>>
ExceptionEventBean.java
<<use>>
fetches latest exception events

Figure 7.13. Component Diagram of the Tab View

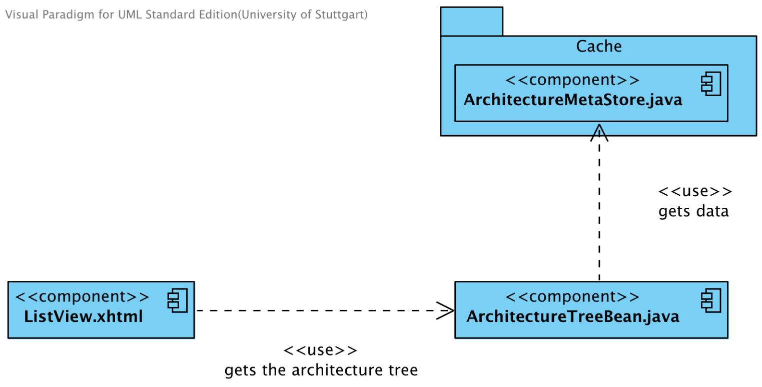

Visual Paradigm for UML Standard Edition(University of Stuttgart)
Cache
<<component>>
ArchitectureMetaStore.java
<<use>>
gets data
<<component>>
ListView.xhtml
<<component>>
ArchitectureTreeBean.java
<<use>>
gets the architecture tree

Figure 7.14. Component Diagram of the List View

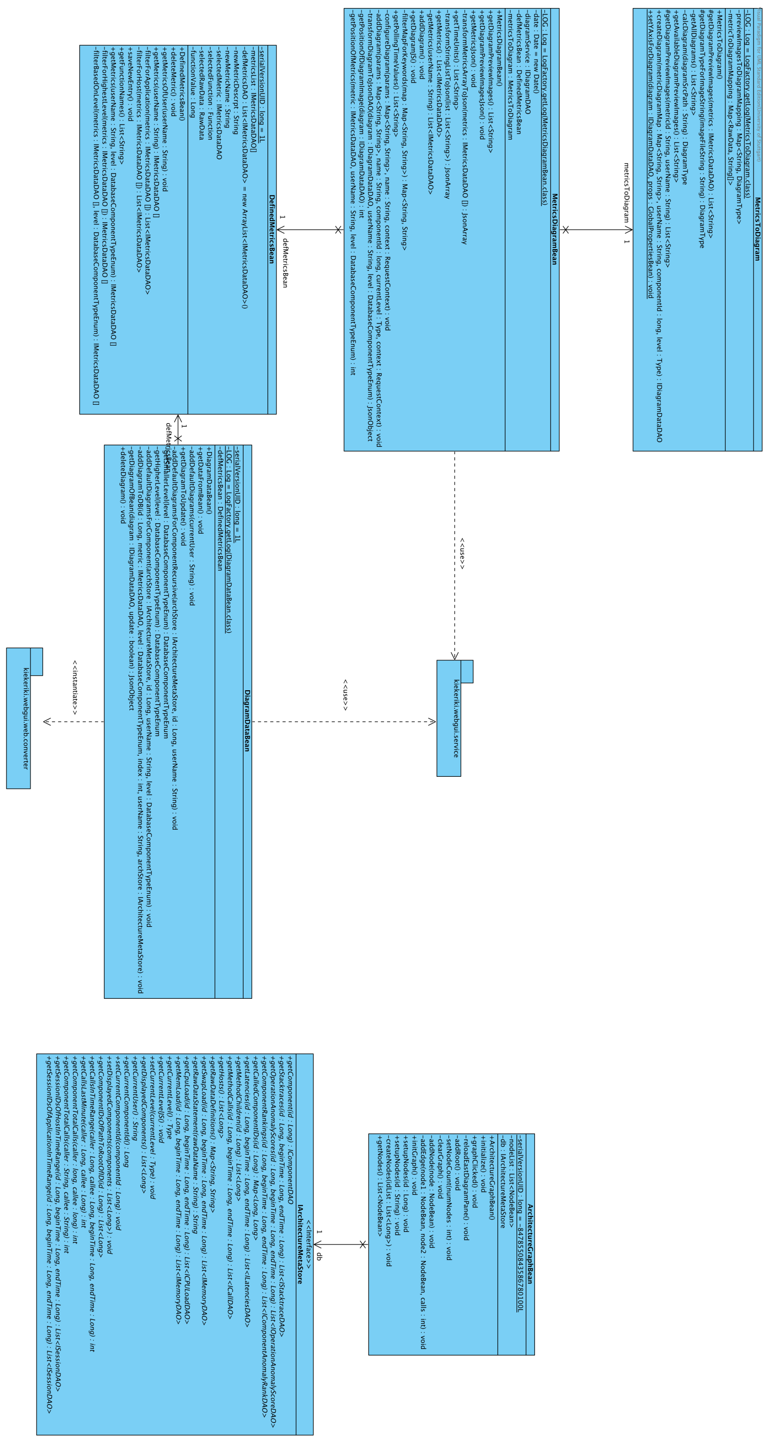
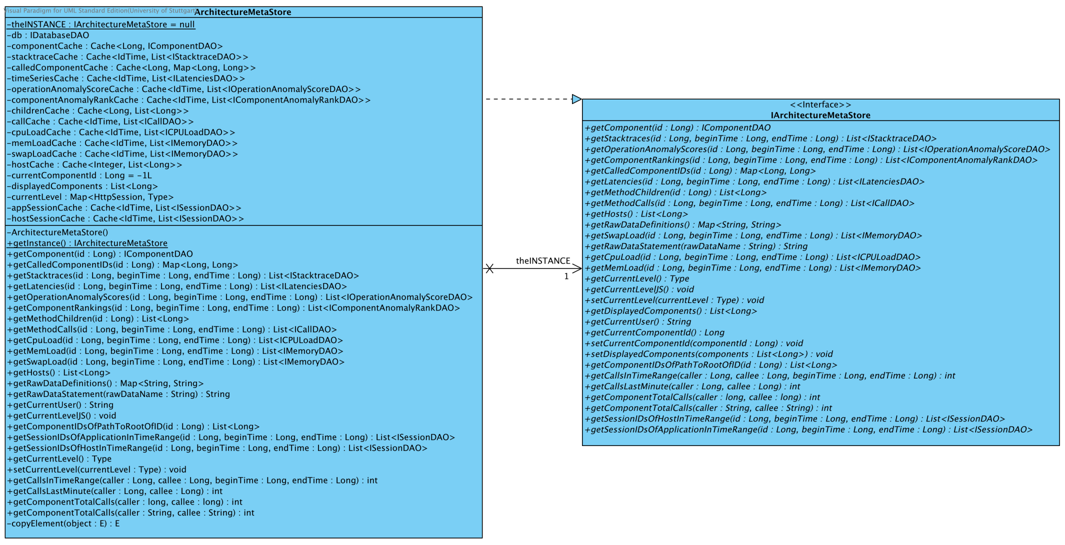


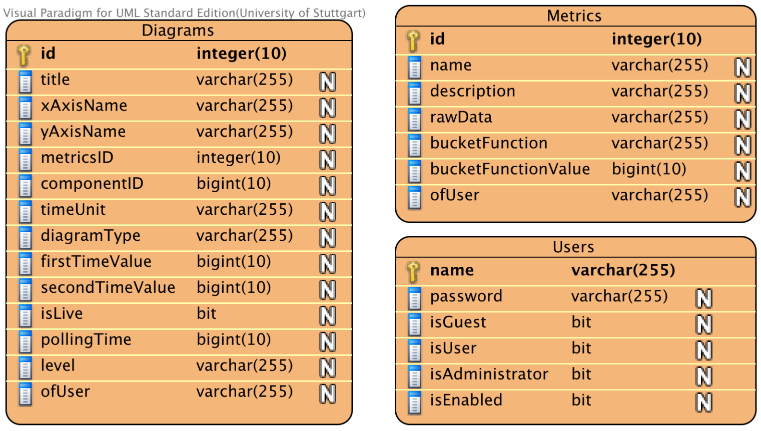
Figure 7.15. Class Diagram of the Web GUI Logic


Visual Paradigm for UML Standard Edition(University of Stuttgart)
theINSTANCE
1

```
ArchitectureMetaStore
-theINSTANCE : IArchitectureMetaStore = null
-db : IDatabaseDAO
-componentCache : Cache<Long, IComponentDAO>
-stacktraceCache : Cache<IdTime, List<IStacktraceDAO>>
-calledComponentCache : Cache<Long, Map<Long, Long>>
-timeSeriesCache : Cache<IdTime, List<ILatenciesDAO>>
-operationAnomalyScoreCache : Cache<IdTime, List<IOperationAnomalyScoreDAO>>
-componentAnomalyRankCache : Cache<IdTime, List<IComponentAnomalyRankDAO>>
-childrenCache : Cache<Long, List<Long>>
-callCache : Cache<IdTime, List<ICallDAO>>
-cpuLoadCache : Cache<IdTime, List<ICPULoadDAO>>
-memLoadCache : Cache<IdTime, List<IMemoryDAO>>
-swapLoadCache : Cache<IdTime, List<IMemoryDAO>>
-hostCache : Cache<Integer, List<Long>>
-currentComponentId : Long = -1L
-displayedComponents : List<Long>
-currentLevel : Map<HttpSession, Type>
-appSessionCache : Cache<IdTime, List<ISessionDAO>>
-hostSessionCache : Cache<IdTime, List<ISessionDAO>>
-ArchitectureMetaStore()
+getInstance() : IArchitectureMetaStore
+getComponent(id : Long) : IComponentDAO
+getCalledComponentIDs(id : Long) : Map<Long, Long>
+getStacktraces(id : Long, beginTime : Long, endTime : Long) : List<IStacktraceDAO>
+getLatencies(id : Long, beginTime : Long, endTime : Long) : List<ILatenciesDAO>
+getOperationAnomalyScores(id : Long, beginTime : Long, endTime : Long) : List<IOperationAnomalyScoreDAO>
+getComponentRankings(id : Long, beginTime : Long, endTime : Long) : List<IComponentAnomalyRankDAO>
+getMethodChildren(id : Long) : List<Long>
+getMethodCalls(id : Long, beginTime : Long, endTime : Long) : List<ICallDAO>
+getCpuLoad(id : Long, beginTime : Long, endTime : Long) : List<ICPULoadDAO>
+getMemLoad(id : Long, beginTime : Long, endTime : Long) : List<IMemoryDAO>
+getSwapLoad(id : Long, beginTime : Long, endTime : Long) : List<IMemoryDAO>
+getHosts() : List<Long>
+getRawDataDefinitions() : Map<String, String>
+getRawDataStatement(rawDataName : String) : String
+getCurrentUser() : String
+getCurrentLevelJS() : void
+getComponentIDsOfPathToRootOfID(id : Long) : List<Long>
+getSessionIDsOfApplicationInTimeRange(id : Long, beginTime : Long, endTime : Long) : List<ISessionDAO>
+getSessionIDsOfHostInTimeRange(id : Long, beginTime : Long, endTime : Long) : List<ISessionDAO>
+getCurrentLevel() : Type
+setCurrentLevel(currentLevel : Type) : void
+getCallsInTimeRange(caller : Long, callee : Long, beginTime : Long, endTime : Long) : int
+getCallsLastMinute(caller : Long, callee : Long) : int
+getComponentTotalCalls(caller : long, callee : long) : int
+getComponentTotalCalls(caller : String, callee : String) : int
-copyElement(object : E) : E
```

```
<<Interface>>
IArchitectureMetaStore
+getComponent(id : Long) : IComponentDAO
+getStacktraces(id : Long, beginTime : Long, endTime : Long) : List<IStacktraceDAO>
+getOperationAnomalyScores(id : Long, beginTime : Long, endTime : Long) : List<IOperationAnomalyScoreDAO>
+getComponentRankings(id : Long, beginTime : Long, endTime : Long) : List<IComponentAnomalyRankDAO>
+getCalledComponentIDs(id : Long) : Map<Long, Long>
+getLatencies(id : Long, beginTime : Long, endTime : Long) : List<ILatenciesDAO>
+getMethodChildren(id : Long) : List<Long>
+getMethodCalls(id : Long, beginTime : Long, endTime : Long) : List<ICallDAO>
+getHosts() : List<Long>
+getRawDataDefinitions() : Map<String, String>
+getSwapLoad(id : Long, beginTime : Long, endTime : Long) : List<IMemoryDAO>
+getRawDataStatement(rawDataName : String) : String
+getCpuLoad(id : Long, beginTime : Long, endTime : Long) : List<ICPULoadDAO>
+getMemLoad(id : Long, beginTime : Long, endTime : Long) : List<IMemoryDAO>
+getCurrentLevel() : Type
+getCurrentLevelJS() : void
+setCurrentLevel(currentLevel : Type) : void
+getDisplayedComponents() : List<Long>
+getCurrentUser() : String
+getCurrentComponentId() : Long
+setCurrentComponentId(componentId : Long) : void
+setDisplayedComponents(components : List<Long>) : void
+getComponentIDsOfPathToRootOfID(id : Long) : List<Long>
+getCallsInTimeRange(caller : Long, callee : Long, beginTime : Long, endTime : Long) : int
+getCallsLastMinute(caller : Long, callee : Long) : int
+getComponentTotalCalls(caller : long, callee : long) : int
+getComponentTotalCalls(caller : String, callee : String) : int
+getSessionIDsOfHostInTimeRange(id : Long, beginTime : Long, endTime : Long) : List<ISessionDAO>
+getSessionIDsOfApplicationInTimeRange(id : Long, beginTime : Long, endTime : Long) : List<ISessionDAO>
```

Figure 7.16. Class Diagram of the Web GUI Cache


Visual Paradigm for UML Standard Edition(University of Stuttgart)

Diagrams		
id	**integer(10)**	
title	varchar(255)	N
xAxisName	varchar(255)	N
yAxisName	varchar(255)	N
metricsID	integer(10)	N
componentID	bigint(10)	N
timeUnit	varchar(255)	N
diagramType	varchar(255)	N
firstTimeValue	bigint(10)	N
secondTimeValue	bigint(10)	N
isLive	bit	N
pollingTime	bigint(10)	N
level	varchar(255)	N
ofUser	varchar(255)	N

Metrics		
id	**integer(10)**	
name	varchar(255)	N
description	varchar(255)	N
rawData	varchar(255)	N
bucketFunction	varchar(255)	N
bucketFunctionValue	bigint(10)	N
ofUser	varchar(255)	N

Users		
name	**varchar(255)**	
password	varchar(255)	N
isGuest	bit	N
isUser	bit	N
isAdministrator	bit	N
isEnabled	bit	N

Figure 7.17. ER Diagram of the WebGUI Internal Database

7.3 Transfer Database

This chapter describes the functionality of the Transfer Database in the Kiekeriki project. The Figure 7.21 shows the architecture of the Kiekeriki project and the role of the Transfer Database in it. Chapter 7.3.1 describes the core principles of the Transfer Database and its logic implemented in its Stored Procedures.

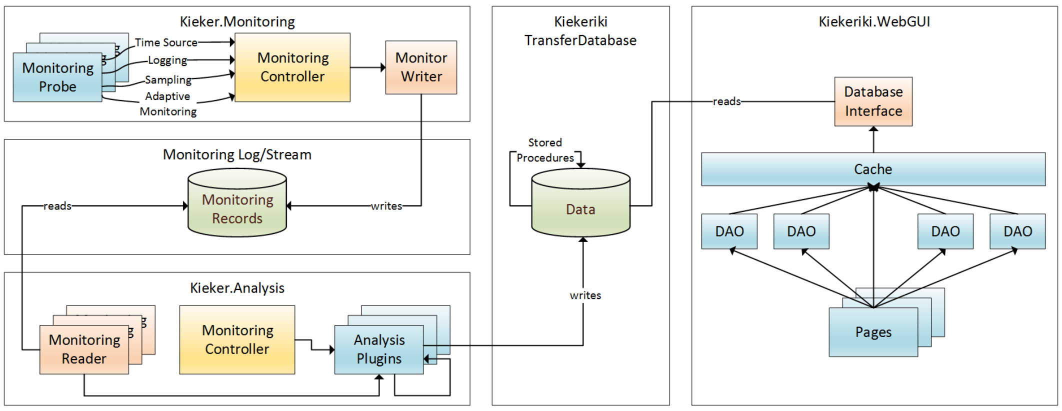


Figure 7.18. Kiekeriki Architecture[van Hoorn]

The Backend is connected to the Transfer Database due the writer plugins. The Backend writes due its database connection component processed monitoring record informations into the Transfer Database. The Frontend is also connected to Transfer Database which holds all the monitoring and analysis data which should be displayed in the UI.

Christian Endres

7.3.1 Database Design

Like the idea of the component tree described in section 5.4.1 the idea of the Transfer Database design is based on the component tree. The central table `InstrumentedComponent` hosts the component tree which points recursively to itself to represent the tree. In the Figure 7.19 you can see the `InstrumentedComponent` table and its references on the name and type of the component. The table is referenced twice by the table OperationDetails which stores the caller and callee correlation to a timestamp for each occurrence of a operation call and the corresponding latency measurement or a stacktrace occurrence. This example describes how the raw data is attached to one or two Instrumented Components.

All other tables are attached to the Instrumented Component, which display the raw data described in section 5.4.2. The figure 7.20 shows the whole table design.

Christian Endres

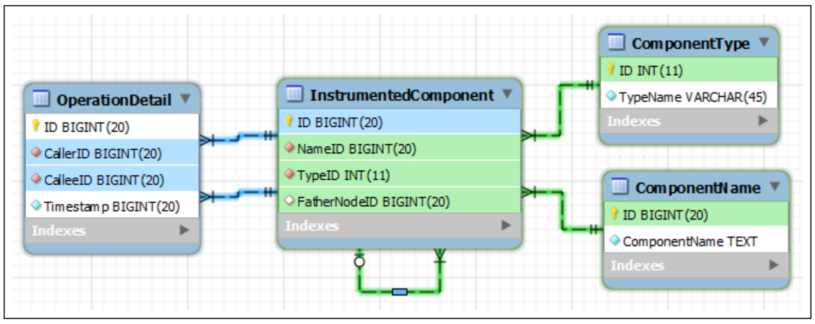


Figure 7.19. Central table of the Transfer Database

7.3.2 Database Connections and Stored Procedures

The MySQL-Server provides the functionality to call Stored Procedures which can encapsulate even complex data access logic. The goal to pursue by using this functionality is to separate the data access logic and the data processing logic. While the Kiekeriki Backend implements the data generation, it should not necessarily care about how to sort it into the dedicated tables. Simple insert methods are desirable. Just as the Kiekeriki Frontend should not care about the storage design of the data, it just wants to access the data in a convenient way.

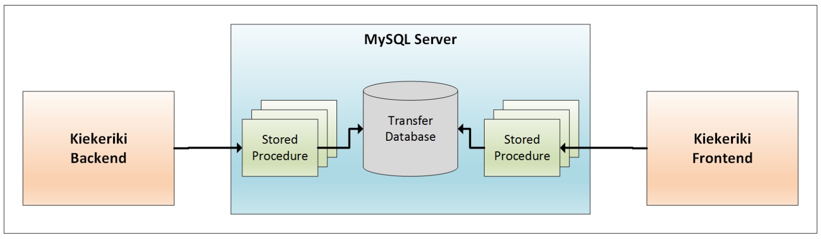


Figure 7.21. Transfer Database data access

Database connection of the Kiekeriki Backend

The database connection of the Kiekeriki Backend cares for the write access to the Transfer Database. Each method provided by the Interface of the database connection hides the logic of data manipulation and access to the database. There are three key concepts: First of all the database connection knows about the stored data in the Transfer Database – especially the keys of already known Instrumented Components. If a new signature of a component is inserted, the database connection cares for committing the new signature immediately. Secondly the database connection instance handles the data sets in batches, except new components which are committed instantly. Thus the writing efficiency is increased because

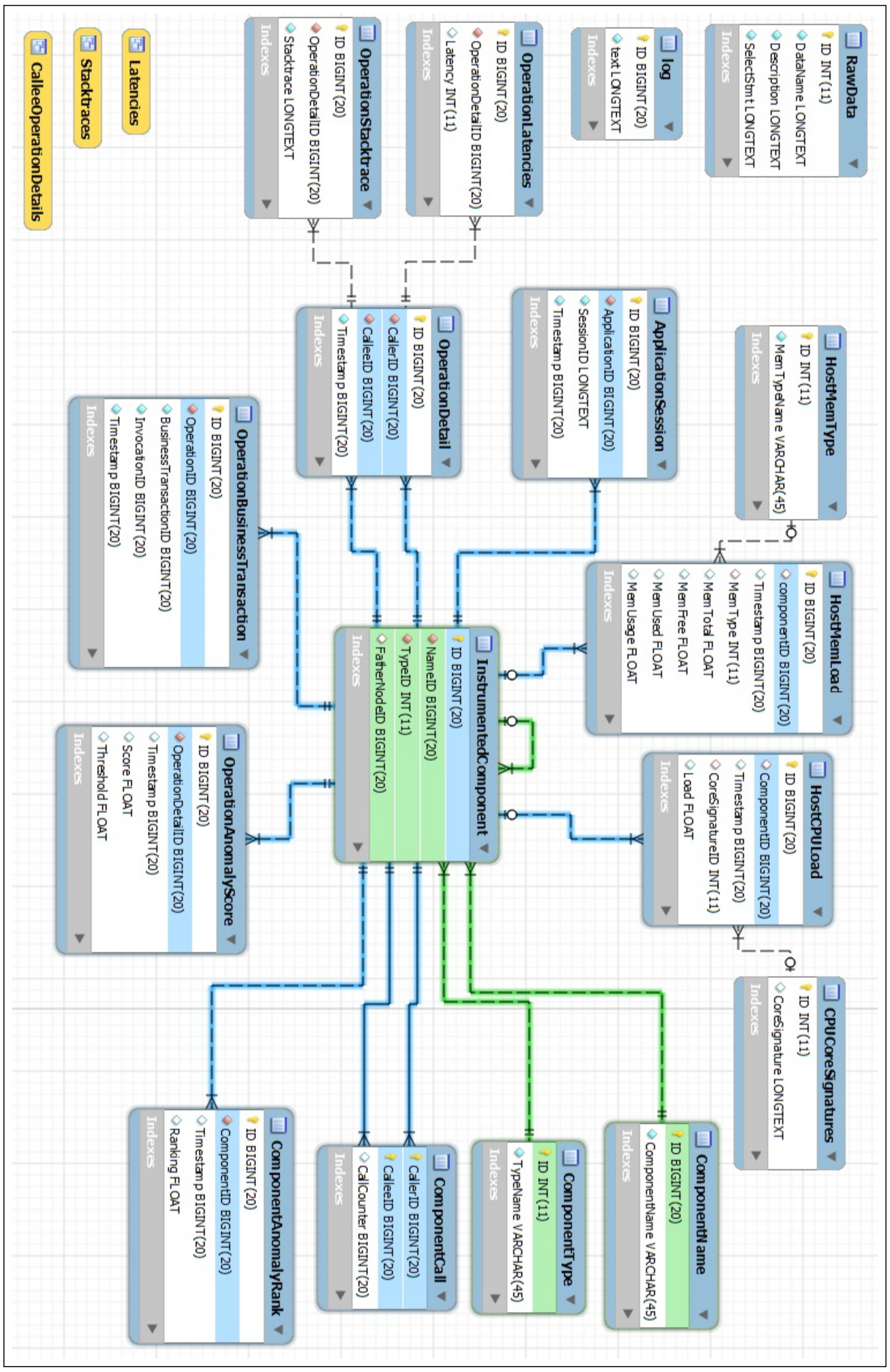


Figure 7.20. Transfer Database design

the connection overhead per data set is reduced. Last but not least each database connection instance has its own thread which commits all batched data after an elapsed time. Per default and in respect of live monitoring the thread commits the data sets each second. A batch is committed if it reaches a certain size or due the wait limit of the thread.

Additional aggregation of data is provided by the Stored Procedures of the MySQL Server.

Database connection of the Kiekeriki Frontend

The database connection of the Kiekeriki Frontend provides reading access to the Transfer Database for the visualization logic. Each method provided by the Interface of the database connection hides logic of data acquisition and aggregation. More complex logic like recursively traversing the component tree or selecting data just in a certain period of time is done by Stored Procedures. The database connection component processes with the retrieved data and returns the data processed into the internal data model.

Christian Endres

7.4 Transfer Database Data Generator and Benchmarking Tool

This section describes a tool which serves the Kiekeriki Frontend as generator of test data. During the second half of the project the tool was enhanced with functionality for benchmarking the Transfer Database. For the sake of readability the data generation functionality is called Data Generator and the benchmarking functionality is called Benchmark Tool. Both are the same program, however serving to achieve different goals. The Benchmark Tool uses the same data generation, write and read functionality as the Data Generator. Though the focus of this section is on the key features and not on the programmatically connections between both and the reused components. The Eclipse project is named `KiekerTransferDBDataGenerator`.

The database connection components are adapted copies of the Kiekeriki Backend and Frontend. The adaptions are mostly for visibility of objects and log output to gain measurement data.

7.4.1 Data Generator

The Data Generator uses the database connection of the Kiekeriki Backend. It writes randomly synthesised data to the Transfer Database. The synthesised data is modeled to fit the data written by the Kiekeriki Backend. Thus RanCorr ranks are depend to the simulated behavior, a low latency results in a low RanCorr rank and a stacktrace occurrence results in a high RanCorr Rank which stands for anomalous behavior.

Christian Endres

7.4.2 Benchmark Tool

The Benchmark tool uses a benchmark framework written by the IBM developer Brent Boyer [Boyer, 2008a,b,c]. The framework cares about the experiment conduction, starting with a heating phase to the end with a summary and statistical analysis. The aim of the framework is to eliminate non deterministic measurements caused by intervention of the Java VM like dead code analysis (DC), garbage collection and just-in-time (JIT) compiling to name some. After a heating phase there are at least 60 measurement rounds with at least one measurement of the benchmarked functionality. If for example the compiling time differs during the measurement rounds — the DC or JIT compiling was active — all is started over. Eventually there is a short or a detailed statistical analysis of the benchmark. Due the results differ with a cross test analysis done with R, the benchmark framework is used to generate the measurement data and the analysis in chapter 8.2.3 and 8.2.4 is done with R and RStudio.

The Benchmark Tool benchmarks the database connection of the Kiekeriki Frontend which retrieves data and processes it into the internal model. The benchmark of the Kiekeriki Backend writes data into the Transfer Database. The conduction and the results are analysed in the chapter 8.2.

Christian Endres

Chapter 8

Experiments and Validation

8.1 Experiments and validation of the Kiekeriki Backend

8.1.1 Evaluation goals

Described here are the goals by which the Kiekeriki backend should be evaluated.

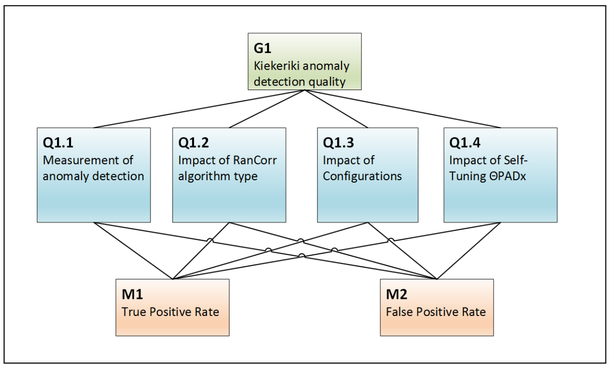

G1
Kiekeriki anomaly detection quality
Q1.1
Measurement of anomaly detection
Q1.2
Impact of RanCorr algorithm type
Q1.3
Impact of Configurations
Q1.4
Impact of Self-Tuning ΘPADx
M1
True Positive Rate
M2
False Positive Rate

Figure 8.1. Goal figure 1: High Accuracy

Table 8.2. Goal 1: Kiekeriki Accuracy

Goal	G1	Determining the quality of Kiekeriki anomaly detection.
	Purpose	Measurement of the system to allow comparisons.
	Issue	Accuracy
	Object	Backend pipes and filter structures in several variants.
	Viewpoint	Developer
Question	Q1.1	How well does Kiekeriki find injected anomalies?
	Q1.2	How does the RanCorr algorithm type affect the output?
	Q1.3	How do differing thresholds and configurations change our anomaly detection?
	Q1.4	What results does the new Self-Tuning algorithm deliver?
Metrics	M1	True Positive Rate
	M2	False Positive Rate

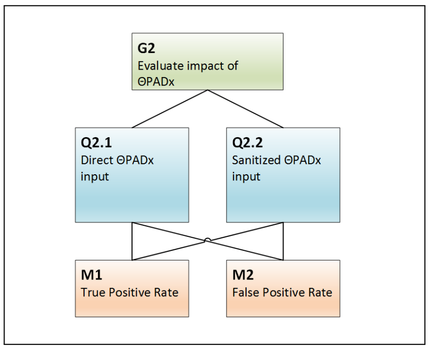


Figure 8.2. Goal figure 2: ΘPADx effect

Table 8.4. Goal 2: ΘPADx effects

Goal	G2	Determine impact of ΘPADxon system.
	Purpose	Analysis of effects that ΘPADx creates.
	Issue	Accuracy
	Object	ΘPADx filters in backend pipes and filters structure.
	Viewpoint	Developer
Question	Q2.1	What impact does using ΘPADx have as opposed to using self generated test data with anomaly scores?
	Q2.2	How does post-processing ΘPADx data for clarity change the outcome?
Metrics	M1	True Positive Rate
	M2	False Positive Rate

8.1.2 Experimental design

This section describes the experiments that allows finding answers to the questions noted in Section 8.1.1. To allow the generation and gathering of data, the Kieker pipes and filters architecture was extended with several filters to surround the ΘPADx and RanCorr filters.

Pipes and Filters setup

As can be seen in Figure 8.3,both the ΘPADx and RanCorr filters are surrounded by the test framework filters to allow controlling data in and outputs.

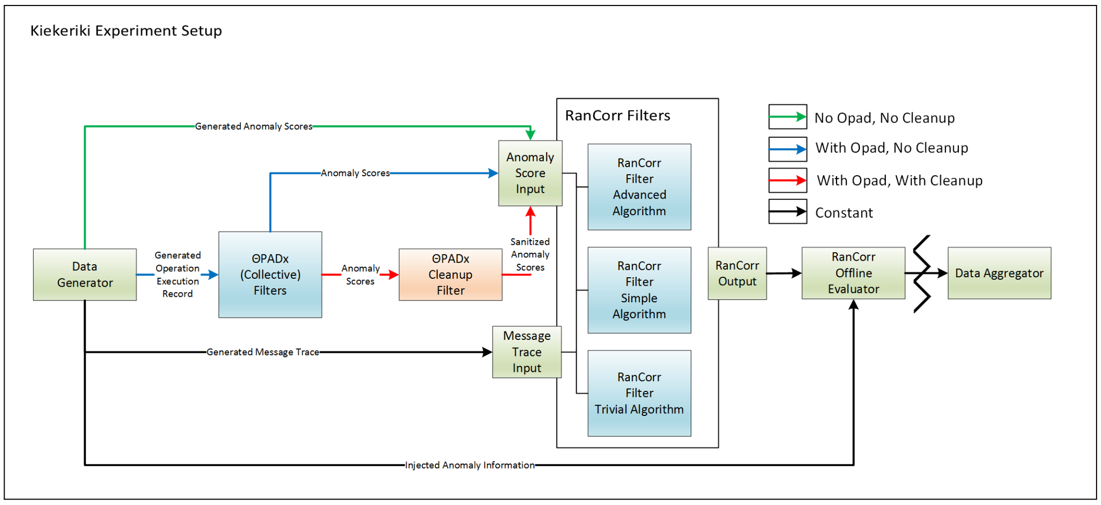


Figure 8.3. Experiment setup

The following list describes the additionally developed evaluation tools:

- Data Generator - A filter that creates data according to a static hierarchy. Will inject anomalies into the data for the system to find. See Section 8.1.2.
- RanCorr Offline Evaluator - designed to use the anomaly data from the Data Generator as well as the RanCorr output data. Will iterate evaluating the data with an internal threshold starting at 0.05 and incrementing by 0.05 until 1.00 and return the results for each evaluation.
- Data Aggregator - Takes the data from several runs and aggregates them into one dataset for ease of use. See Section 8.1.2 for details.

Paths through Pipes-and-Filters

On Display in Figure 8.3 are three different paths that the data takes through our pipes and filters structure. Constant connections that always exist are marked in black. These include the information about our injected anomaly and the message trace of generated data. The first is required for evaluation of RanCorr, the latter is needed by RanCorr itself to build the hierarchy of calls. See Section 7.1.2 on details of how RanCorr is implemented.

The green path is the most direct. Instead of using ΘPADx, anomaly scores are generated in accordance with patterns that are described in Section 8.1.2. This data is used by RanCorr to determine an anomaly ranking. This is done to allow studying the impact of the question from Table 8.4. We therefore have a baseline on how externally generated data compares to data ΘPADx generates.

The blue path is the most realistic one. In this version, the data created by the Input Generator is run through ΘPADx without post-processing. That data is then piped towards RanCorr.

The red path is a new path that was added after it was discovered that the ΘPADx implementation returns anomaly scores of 1.0 when no latency data is received for a method. While this is perfectly acceptable, it does cause a problem with the experiment setup. Control over anomalies is desired, therefore any value from ΘPADx with a measured latency of 0 is removed before the data is passed on to RanCorr. This is also done to answer the question postulated in Table 8.4. In our data generation, we do not consider these events, when a method is not detected an anomaly. We inject our own anomalies and we wish to detect only these. Fortunately it is quite simple to detect these, in the eyes of the testers at least, fallacious anomalies and remove them from the stream of data. Since these are not included in the anomalies to test for, their removal does not make the data unusable, as we can still test for the injected anomalies. An alternative would be to extend the data generation script to ensure that such downtimes do not exist. Unfortunately that was not possible within the timeframe of the project.

Also one should note that the three items named "Anomaly Score Input", "Message Trace Input" and "RanCorr Output" are simplifications to aid readability. In fact, each of the incoming pipes is connected to each instance of the RanCorr filter.

Data Generation

The data generator has four possible outputs.

- AnomalyTimeStampOut - The data telling the evaluator at what time an anomaly is injected, which method it affects and what kind of anomaly it is.
- OperationExecutionRecord - A Kieker class containing instrumentation data. The class contains latencies and anomaly are expressed in the form of longer than usual latencies.
- OperationAnomalyInfo - Required by RanCorr, it contains generated anomaly scores in keeping with the patterns described in Figure 8.4.
- MessageTrace - Also required by RanCorr, describes the call hierarchy.

The first step of generating data for the system is creating a hierarchy in which the objects we want to include are represented. These are:

- Hosts
- Applications
- Packages
- Classes
- Methods

Simply put, a host contains applications, an application contains packages. A package however can contain both classes and packages. Classes contain only methods. This data is required for RanCorr and is currently built in a static manner. It is technically possible to dynamically generate a new one at each runtime, but this would create difficulty further down the line at evaluating the data and also make it difficult to compare results from different runs. This data is used in each of the four outputs that the data generator has.
Once this hierarchy is created, a pool of methods is available for data generations. With this pool we can create a message hierarchy of calls and returns. This hierarchy is split into several patterns that were predetermined in order to allow for different scenarios.

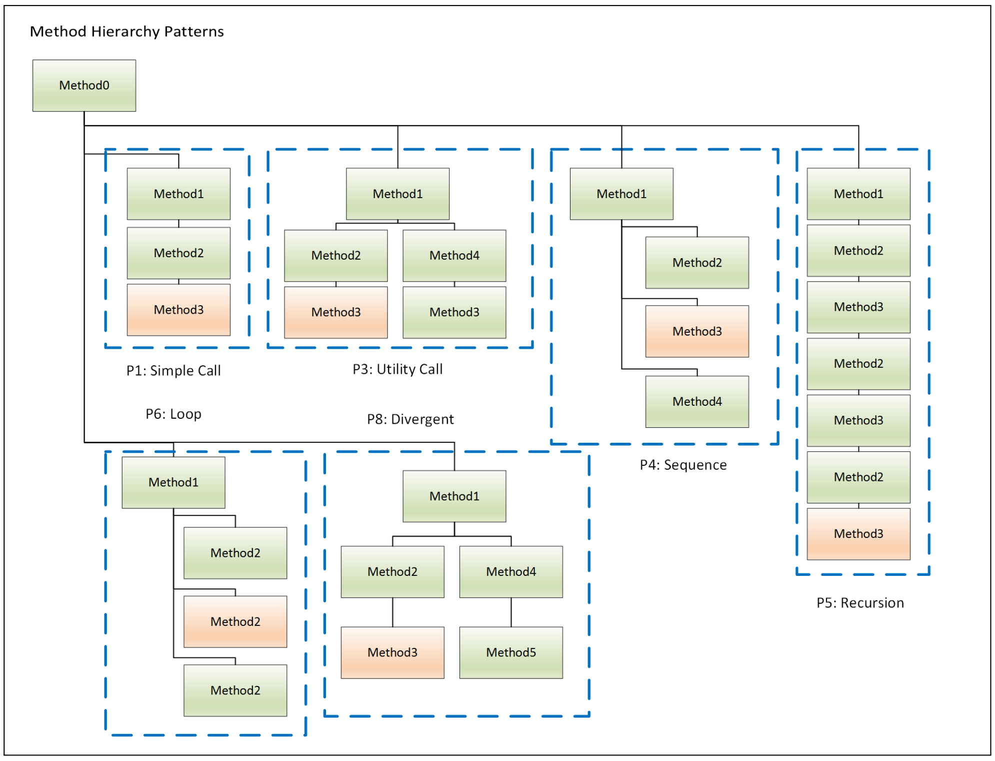


Figure 8.4. Call Hierarchy Patterns - Methods marked in red are possible anomaly positions.

These scenarios were picked to allow different types of calls that seemed basic and widespread. At the moment, not much is done with this pattern information. It would however, in the future, be possible to take a closer look at the RanCorr results. For that eventuality, each pattern uses only methods with specific names, to allow easy identification. Currently, accuracy in rancor is calculated by getting the False Positives, True Positives, False Negatives and True Negatives.

Each of the patterns in Figure 8.4 is represented in the final data and at data generation time.

Once a call hierarchy is created, we can export it to RanCorr as a MessageTrace. Iterating over this, it is now possible to generate the actual records (OperationExecutionRecords) for ΘPADx, the anomaly scores (OperationAnomalyInfo) for RanCorr and the anomaly data (AnomalyTimeStampOut) for the evaluation. The amount of times we iterate over is given as a parameter for our experimentation in Section 8.1.2. At this point we also inject anomalies. As can be seen in Figure 8.4, in each pattern there are possible anomaly injection sites and at generation time, the system will create anomalies there on a chance set by a constant (25% default chance). These anomalies are either:

- Point Anomaly - Only a single instance of this message being called is anomalous.
- Collective Anomaly - Several instances of this message call are created anomalous.
- Contextual Anomaly - While several instances are created anomalous, some in the middle are created with a normal value.

Important to know is that what constitutes an anomalous latency is currently set by constant at five times the default runtime of the method.

By having these anomalies and patterns, a matrix is created where any kind of anomaly is possible to be generated in any pattern. Due to the fact that each pattern is easily distinguishable and that each anomaly is recorded, even with more patterns added it should be possible to evaluate RanCorr or ΘPADx results in the future.

Experiment setup

After one successful run, there are 19 results created for each RanCorr Algorithm; trivial, simple and advanced. These 19 values represent the threshold at which we consider an anomaly rating from RanCorr anomalous. They start at 0.05 and increase by 0.05 until reaching 0.95.

This is iterated thrice, once for each path described in Section 8.1.2. This step (including the one above) is a single run generating one set of data. This in turn is iterated over several times, depending on the amount of data desired, creating a large base of data to calculate, for example, our True Positive and False Positive rates from. Once this is achieved, all the necessary data has been collected for one ΘPADx forecasting filter.

The entire procedure is repeated for another forecasting filter. In this experiment, the “JavaMean” forecaster will be compared to the “Extended Self-Tuning Forecasting Filter”.

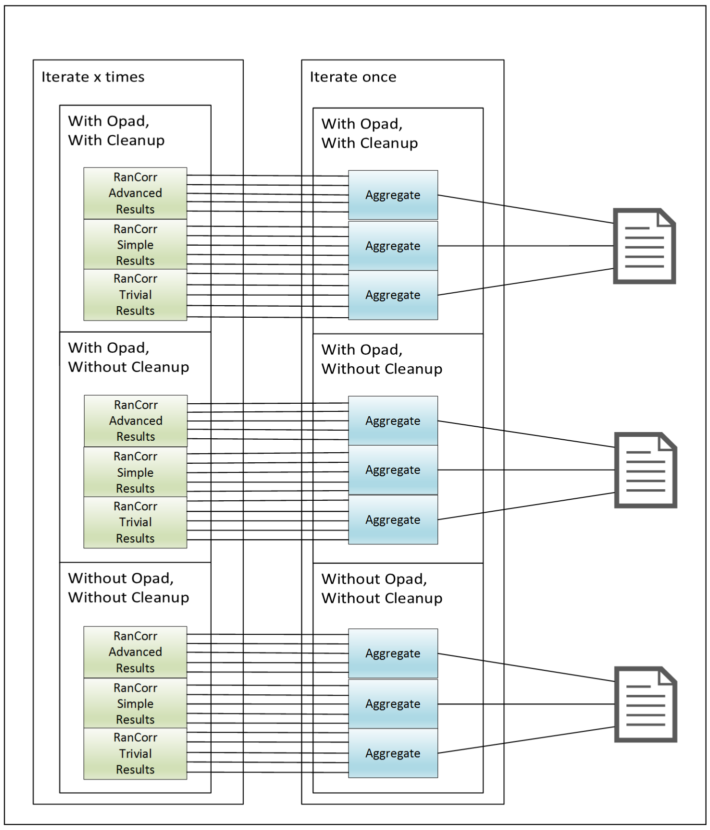


Figure 8.5. Experiment Setup and Data Aggregation
Generates data for one ΘPAD forecasting filter.

Variables

The following variables have been identified and are in order of magnitute of our testing.

- Threshhold - The value at which the RanCorrOffline evaluator considers an anomaly ranking anomalous. Is iterated over 19 times for each RanCorr Algorithm.
- RanCorr Algorithm - Which algorithm was chosen, trivial, simple or advanced? Is done thrice for each path chosen.
- Path chosen - In accordance with Figure 8.3 there are three possible paths through the pipes-and-filters structure: No ΘPADx No cleanup, with ΘPADx no cleanup and with ΘPADx with cleanup. Is done thrice for each forecasting algorithm
- ΘPADx Forecasting Algorithm - Either “JavaMean” or “Extended Self-Tuning Forecasting” filter.

Data Aggregation

The data aggregation takes data generated by a large amount of runs and condenses it into one set. This is accomplished by gathering the True Positives, True Negatives, False Positives and False Negatives of all corresponding data into one new set. Each time the average RanCorrAccuracy and RanCorrClearness are calculated and stored. At the very end, the Accuracy, Precision, F-Measure, True Positive Rate and False Positive Rate are calculated for the corresponding data set.
In this case, corresponding data refers to data sets that are run with the same setup (for example; With Opad, Without Cleanup), are measured using the same RanCorr algorithm (Trivial, Simple, Advanced) and are also calculated with an identical threshold.

Experiment execution

The execution of the experiment contains two phases as mentioned in Section 8.1.2, one for each ΘPAD forecasting filter. For each of these, a script is run which handles the iterations described. The default value for iterations for one forecasting filter is 100. Once all of these have been run the data aggregator is started, condensing our results to a three set of 19 results for each of the three RanCorr algorithms, one set for each path described in Section 8.1.2. This data is ready for analysis and will create an overview of the quality of RanCorr with a specific forecasting filter.

Christopher Gregorian

8.1.3 Experiment data and results

JavaMean Results

First, the aggregated results created using the “JavaMean” forecasting filter.

Table 8.5. "JavaMean" forecasting, No ΘPADx No Cleanup, Advanced Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	14	900	500	0	1500
0.1	5	14	900	500	0	1500
0.15	5	14	855	500	0	1545
0.2	5	13	790	500	0	1610
0.25	5	12	703	500	0	1697
0.3	5	12	699	500	0	1701
0.35	5	12	652	497	3	1748
0.4	5	10	503	441	59	1897
0.45	5	9	492	400	100	1908
0.5	5	3	0	157	343	2400
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.6. "JavaMean" forecasting, No ΘPADx No Cleanup, Simple Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	14	896	500	0	1504
0.1	5	12	616	499	1	1784
0.15	5	10	410	409	91	1990
0.2	5	7	165	400	100	2235
0.25	5	4	0	399	101	2400
0.3	5	4	0	358	142	2400
0.35	5	3	0	301	199	2400
0.4	5	3	0	300	200	2400
0.45	5	3	0	289	211	2400
0.5	5	3	0	157	343	2400
0.55	5	0	0	9	491	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.7. “JavaMean” forecasting, No ΘPADx No Cleanup, Trivial Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	14	896	500	0	1504
0.1	5	12	616	499	1	1784
0.15	5	10	410	409	91	1990
0.2	5	7	165	400	100	2235
0.25	5	4	0	399	101	2400
0.3	5	4	0	358	142	2400
0.35	5	3	0	301	199	2400
0.4	5	3	0	300	200	2400
0.45	5	3	0	289	211	2400
0.5	5	3	0	157	343	2400
0.55	5	0	0	9	491	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.8. “JavaMean” forecasting, ΘPADx No Cleanup, Advanced Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	18	1300	500	0	1100
0.1	5	18	1300	500	0	1100
0.15	5	18	1300	500	0	1100
0.2	5	18	1300	500	0	1100
0.25	5	18	1300	500	0	1100
0.3	5	18	1300	500	0	1100
0.35	5	18	1300	500	0	1100
0.4	5	18	1300	500	0	1100
0.45	5	18	1300	500	0	1100
0.5	5	18	1300	500	0	1100
0.55	5	7	619	48	452	1781
0.6	5	2	178	0	500	2222
0.65	5	1	60	0	500	2340
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.9. “JavaMean” forecasting, ΘPADx No Cleanup, Simple Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	18	1300	500	0	1100
0.1	5	18	1300	500	0	1100
0.15	5	18	1300	500	0	1100
0.2	5	18	1300	500	0	1100
0.25	5	18	1300	500	0	1100
0.3	5	18	1300	500	0	1100
0.35	5	18	1300	500	0	1100
0.4	5	18	1300	500	0	1100
0.45	5	18	1300	500	0	1100
0.5	5	18	1300	500	0	1100
0.55	5	18	1300	500	0	1100
0.6	5	18	1300	500	0	1100
0.65	5	18	1300	500	0	1100
0.7	5	18	1255	500	0	1145
0.75	5	16	1100	500	0	1300
0.8	5	12	1085	101	399	1315
0.85	5	2	197	0	500	2203
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.10. “JavaMean” forecasting, ΘPADx No Cleanup, Trivial Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	18	1300	500	0	1100
0.1	5	18	1300	500	0	1100
0.15	5	18	1300	500	0	1100
0.2	5	18	1300	500	0	1100
0.25	5	18	1300	500	0	1100
0.3	5	18	1300	500	0	1100
0.35	5	18	1300	500	0	1100
0.4	5	18	1300	500	0	1100
0.45	5	18	1300	500	0	1100
0.5	5	18	1300	500	0	1100
0.55	5	18	1300	500	0	1100
0.6	5	18	1300	500	0	1100
0.65	5	18	1300	500	0	1100
0.7	5	18	1255	500	0	1145
0.75	5	16	1100	500	0	1300
0.8	5	12	1085	101	399	1315
0.85	5	2	197	0	500	2203
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.11. “JavaMean” forecasting, ΘPADx Cleanup, Advanced Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	12	893	500	0	1507
0.1	5	11	736	441	59	1664
0.15	5	11	712	415	85	1688
0.2	5	11	700	400	100	1700
0.25	5	11	700	400	100	1700
0.3	5	11	700	400	100	1700
0.35	5	11	700	400	100	1700
0.4	5	11	700	400	100	1700
0.45	5	11	700	400	100	1700
0.5	5	2	151	87	413	2249
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.12. “JavaMean” forecasting, ΘPADx Cleanup, Simple Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	12	709	498	2	1691
0.1	5	11	700	400	100	1700
0.15	5	11	700	400	100	1700
0.2	5	11	700	400	100	1700
0.25	5	11	700	400	100	1700
0.3	5	11	700	400	100	1700
0.35	5	11	700	386	114	1700
0.4	5	8	635	111	389	1765
0.45	5	3	192	96	404	2208
0.5	5	1	121	39	461	2279
0.55	5	1	18	0	500	2382
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.13. “JavaMean” forecasting, ΘPADx Cleanup, Trivial Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	12	709	498	2	1691
0.1	5	11	700	400	100	1700
0.15	5	11	700	400	100	1700
0.2	5	11	700	400	100	1700
0.25	5	11	700	400	100	1700
0.3	5	11	700	400	100	1700
0.35	5	11	700	386	114	1700
0.4	5	8	635	111	389	1765
0.45	5	3	192	96	404	2208
0.5	5	1	121	39	461	2279
0.55	5	1	18	0	500	2382
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Extended Self-Tuning Forecast Results

The data measured using the Self-Tuning Forecast.

Table 8.14. “Extended Self-Tuning” forecasting, No ΘPADx No Cleanup, Advanced Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	14	900	500	0	1500
0.1	5	14	900	500	0	1500
0.15	5	14	843	500	0	1557
0.2	5	13	784	500	0	1616
0.25	5	12	702	500	0	1698
0.3	5	12	700	500	0	1700
0.35	5	12	633	495	5	1767
0.4	5	10	503	428	72	1897
0.45	5	9	492	400	100	1908
0.5	5	3	0	136	364	2400
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.15. “Extended Self-Tuning” forecasting, No ΘPADx No Cleanup, Simple Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	14	899	500	0	1501
0.1	5	12	592	498	2	1808
0.15	5	9	386	407	93	2014
0.2	5	7	135	400	100	2265
0.25	5	4	0	399	101	2400
0.3	5	4	0	352	148	2400
0.35	5	3	0	303	197	2400
0.4	5	3	0	300	200	2400
0.45	5	3	0	282	218	2400
0.5	5	3	0	136	364	2400
0.55	5	0	0	4	496	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.16. “Extended Self-Tuning” forecasting, No ΘPADx No Cleanup, Trivial Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	14	899	500	0	1501
0.1	5	12	592	498	2	1808
0.15	5	9	386	407	93	2014
0.2	5	7	135	400	100	2265
0.25	5	4	0	399	101	2400
0.3	5	4	0	352	148	2400
0.35	5	3	0	303	197	2400
0.4	5	3	0	300	200	2400
0.45	5	3	0	282	218	2400
0.5	5	3	0	136	364	2400
0.55	5	0	0	4	496	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.17. “Extended Self-Tuning” forecasting, ΘPADx No Cleanup, Advanced Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	11	700	400	100	1700
0.1	5	11	700	386	114	1700
0.15	5	10	700	331	169	1700
0.2	5	10	700	306	194	1700
0.25	5	10	699	297	203	1701
0.3	5	9	645	257	243	1755
0.35	5	2	422	123	377	1978
0.4	5	1	155	7	493	2245
0.45	5	0	38	0	500	2362
0.5	5	0	0	0	500	2400
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.18. “Extended Self-Tuning” forecasting, ΘPADx No Cleanup, Simple Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	10	700	315	185	1700
0.1	5	9	667	302	198	1733
0.15	5	0	87	0	500	2313
0.2	5	0	0	0	500	2400
0.25	5	0	0	0	500	2400
0.3	5	0	0	0	500	2400
0.35	5	0	0	0	500	2400
0.4	5	0	0	0	500	2400
0.45	5	0	0	0	500	2400
0.5	5	0	0	0	500	2400
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.19. “Extended Self-Tuning” forecasting, ΘPADx No Cleanup, Trivial Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	10	700	315	185	1700
0.1	5	9	667	302	198	1733
0.15	5	0	87	0	500	2313
0.2	5	0	0	0	500	2400
0.25	5	0	0	0	500	2400
0.3	5	0	0	0	500	2400
0.35	5	0	0	0	500	2400
0.4	5	0	0	0	500	2400
0.45	5	0	0	0	500	2400
0.5	5	0	0	0	500	2400
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.20. “Extended Self-Tuning” forecasting, ΘPADx Cleanup, Advanced Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	11	722	459	41	1678
0.1	5	11	701	410	90	1699
0.15	5	11	700	403	97	1700
0.2	5	11	700	382	118	1700
0.25	5	10	700	359	141	1700
0.3	5	10	700	324	176	1700
0.35	5	10	699	307	193	1701
0.4	5	10	697	301	199	1703
0.45	5	10	685	300	200	1715
0.5	5	1	121	0	500	2279
0.55	5	0	0	0	500	2400
0.6	5	0	0	0	500	2400
0.65	5	0	0	0	500	2400
0.7	5	0	0	0	500	2400
0.75	5	0	0	0	500	2400
0.8	5	0	0	0	500	2400
0.85	5	0	0	0	500	2400
0.9	5	0	0	0	500	2400
0.95	5	0	0	0	500	2400

Table 8.21. “Extended Self-Tuning” forecasting, ΘPADx Cleanup, Simple Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	11	698	411	89	1702
0.1	5	10	698	320	180	1702
0.15	5	10	697	303	197	1703
0.2	5	10	693	301	199	1707
0.25	5	10	666	301	199	1734
0.3	5	9	614	300	200	1786
0.35	5	6	596	232	268	1804
0.4	5	5	514	7	493	1886
0.45	5	1	158	1	499	2242
0.5	5	0	87	1	499	2313
0.55	5	0	22	1	499	2378
0.6	5	0	9	1	499	2391
0.65	5	0	7	1	499	2393
0.7	5	0	4	1	499	2396
0.75	5	0	3	0	500	2397
0.8	5	0	2	0	500	2398
0.85	5	0	2	0	500	2398
0.9	5	0	1	0	500	2399

Table 8.22. "Extended Self-Tuning" forecasting, ΘPADx Cleanup, Trivial Algorithm

Threshhold	Expected Anomalies	Found Anomalies	FP	TP	FN	TN
0.05	5	11	698	411	89	1702
0.1	5	10	698	320	180	1702
0.15	5	10	697	303	197	1703
0.2	5	10	693	301	199	1707
0.25	5	10	666	301	199	1734
0.3	5	9	614	300	200	1786
0.35	5	6	596	232	268	1804
0.4	5	5	514	7	493	1886
0.45	5	1	158	1	499	2242
0.5	5	0	87	1	499	2313
0.55	5	0	22	1	499	2378
0.6	5	0	9	1	499	2391
0.65	5	0	7	1	499	2393
0.7	5	0	4	1	499	2396
0.75	5	0	3	0	500	2397
0.8	5	0	2	0	500	2398
0.85	5	0	2	0	500	2398
0.9	5	0	1	0	500	2399
0.95	5	0	0	0	500	2400

Christopher Gregorian

RanCorr Evaluation

In order to evaluate the results of RanCorr we built two new filters (see Figure 8.6):

Offline Evaluator is implemented in the class `RanCorrOfflineEvaluator` which has two input ports to get the RanCorr results as an object of the class `RanCorrOutput` and to get the expected anomalies as objects of the class `AnomalyTimeStampOut`. These values are compared and the following metrics are calculated: number of found anomalies for the thresholds in range between +0.05 and +1.0 with step size of 0.05, number of false positives (FP), number of true positives (TP), number of false negatives (FN), number of true negatives (TN), statistical accuracy (8.1), statistical precision (8.2), statistical recall (8.3) and statistical F measure (8.4). Additionally we calculate the special accuracy (8.5) and clearness (8.6) of RanCorr based on Marwede et al. [2009]. The output of that filter is an object of the class `RanCorrOfflineEvaluationResult` which contains all calculated metrics.

Online Evaluator is implemented in the class `RanCorrOnlineEvaluator` which has one input port to get the RanCorr results as an object of the class `RanCorrOutput`. Since we assume during the online evaluation that we do not have any information about the expected anomaly values, not all metrics of the offline evaluation can be calculated in the online evaluation. Therefore we calculate in the online evaluation just the number of found anomalies for the thresholds in range between +0.05 and +1.0 with step size of 0.05.

Additionally we add the concrete anomaly rankings to the output. The output of that filter is an object of the class `RanCorrOnlineEvaluationResult`.

The offline evaluation is used during the experiments, in which the anomalies were injected automatically by our experiment framework and the information about the anomalies is known and accessible. Then it is possible to calculate metrics about how good the approach works. The online evaluation is used during live execution of Kiekeriki to get the information about the calculated anomaly ranking and how much anomalies were identified.

$$Accuracy := \frac{TP\ +\ TN}{TP\ +\ FP\ +\ FN\ +\ TN} \tag{8.1}$$

$$Precision := \frac{TP}{TP\ +\ FP} \tag{8.2}$$

$$Recall := \frac{TP}{FP\ +\ TN} \tag{8.3}$$

$$F\ Measure := 2\ *\ \frac{precision\ *\ recall}{precision\ +\ recall} \tag{8.4}$$

$\{r_1, ..., r_n\}$: anomaly rankings
r_i: anomaly ranking of root cause operation
$rank(r_i)$: position of ranking in ordered list

$$RanCorrAccuracy(\{r_1, ..., r_n\}) := \frac{n\ -\ rank(r_i)}{(n\ -\ 1)} \tag{8.5}$$

j: component with highest anomaly ranking

$$RanCorrClearness(\{r_1, ..., r_n\}) := \frac{r_j}{\sum_{k=1,k\neq j}^{n} \frac{r_k}{rank(r_k)}\ +\ 1} \tag{8.6}$$

Yannic Noller

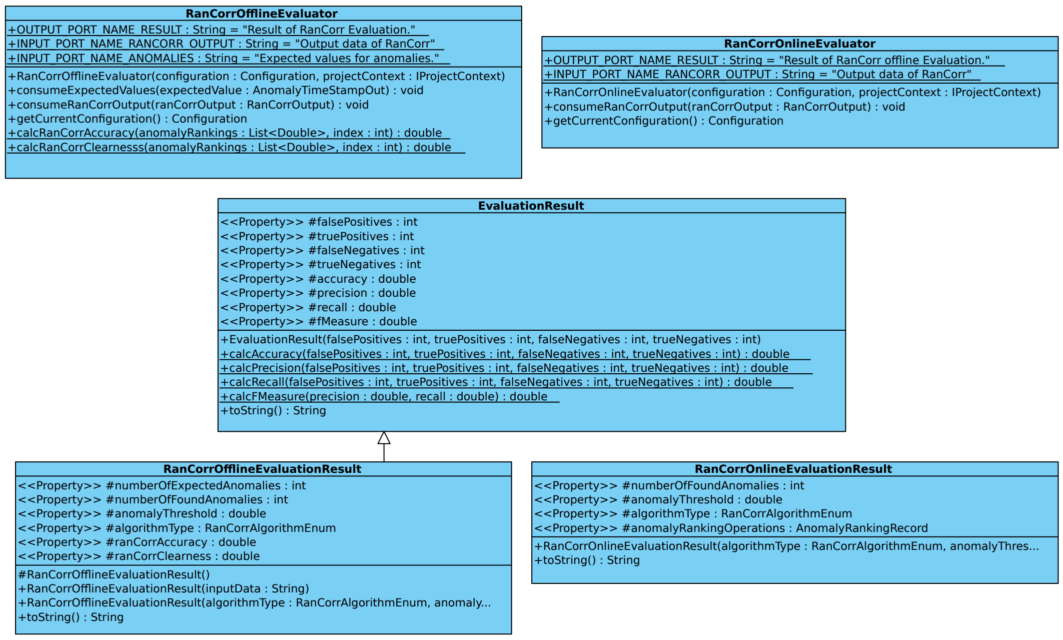

RanCorrOfflineEvaluator
+OUTPUT_PORT_NAME_RESULT : String = "Result of RanCorr Evaluation."
+INPUT_PORT_NAME_RANCORR_OUTPUT : String = "Output data of RanCorr"
+INPUT_PORT_NAME_ANOMALIES : String = "Expected values for anomalies."
+RanCorrOfflineEvaluator(configuration : Configuration, projectContext : IProjectContext)
+consumeExpectedValues(expectedValue : AnomalyTimeStampOut) : void
+consumeRanCorrOutput(ranCorrOutput : RanCorrOutput) : void
+getCurrentConfiguration() : Configuration
+calcRanCorrAccuracy(anomalyRankings : List<Double>, index : int) : double
+calcRanCorrClearnesss(anomalyRankings : List<Double>, index : int) : double
RanCorrOnlineEvaluator
+OUTPUT_PORT_NAME_RESULT : String = "Result of RanCorr offline Evaluation."
+INPUT_PORT_NAME_RANCORR_OUTPUT : String = "Output data of RanCorr"
+RanCorrOnlineEvaluator(configuration : Configuration, projectContext : IProjectContext)
+consumeRanCorrOutput(ranCorrOutput : RanCorrOutput) : void
+getCurrentConfiguration() : Configuration
EvaluationResult
<<Property>> #falsePositives : int
<<Property>> #truePositives : int
<<Property>> #falseNegatives : int
<<Property>> #trueNegatives : int
<<Property>> #accuracy : double
<<Property>> #precision : double
<<Property>> #recall : double
<<Property>> #fMeasure : double
+EvaluationResult(falsePositives : int, truePositives : int, falseNegatives : int, trueNegatives : int)
+calcAccuracy(falsePositives : int, truePositives : int, falseNegatives : int, trueNegatives : int) : double
+calcPrecision(falsePositives : int, truePositives : int, falseNegatives : int, trueNegatives : int) : double
+calcRecall(falsePositives : int, truePositives : int, falseNegatives : int, trueNegatives : int) : double
+calcFMeasure(precision : double, recall : double) : double
+toString() : String
RanCorrOfflineEvaluationResult
<<Property>> #numberOfExpectedAnomalies : int
<<Property>> #numberOfFoundAnomalies : int
<<Property>> #anomalyThreshold : double
<<Property>> #algorithmType : RanCorrAlgorithmEnum
<<Property>> #ranCorrAccuracy : double
<<Property>> #ranCorrClearness : double
#RanCorrOfflineEvaluationResult()
+RanCorrOfflineEvaluationResult(inputData : String)
+RanCorrOfflineEvaluationResult(algorithmType : RanCorrAlgorithmEnum, anomaly...
+toString() : String
RanCorrOnlineEvaluationResult
<<Property>> #numberOfFoundAnomalies : int
<<Property>> #anomalyThreshold : double
<<Property>> #algorithmType : RanCorrAlgorithmEnum
<<Property>> #anomalyRankingOperations : AnomalyRankingRecord
+RanCorrOnlineEvaluationResult(algorithmType : RanCorrAlgorithmEnum, anomalyThres...
+toString() : String

Figure 8.6. The two classes `RanCorrOfflineEvaluator` and `RanCorrOnlineEvaluator` implement the offline and online evaluation filters of the RanCorr approach. Both consume the RanCorr output, but only the offline filter assumes that there are information about the expected anomaly values. Additionally the online filter returns also the actual anomaly ranking information. The results are stored in objects of the classes `RanCorrOfflineEvaluationResult` and `RanCorrOnlineEvaluationResult`.

8.1.4 Analysis of the experiment results

Shown in 8.23 is an example of how the data is analyzed. Not all eighteen tables are on display here for brevity, but each table is calculated in the exact same way. The one table displayed here represents Table 8.11. The data is calculated in Section 8.1.3. The ranCorrClearness is the same for entire table and can only be compared between tables. Unfortunately, there appears to be an error in the RanCorr Accuracy calculation and the value has therefore not been included.

Table 8.23. Analysed data, "JavaMean", Opad And Cleanup, Advanced Algorithm

Precision	Accuracy	Recall	F-Measure	ranCorrClearness	FPR	TPR
0.3615329	0.69551724	0.3615329	0.361532899	0.244726433	0.367916667	1
0.3732993	0.72482759	0.37329932	0.37329932	0.244726433	0.307083333	0.878
0.36875	0.7262069	0.36875	0.36875	0.244726433	0.294583333	0.826
0.3642144	0.72448276	0.36421435	0.364214351	0.244726433	0.291666667	0.802
0.3636364	0.72413793	0.36363636	0.363636364	0.244726433	0.291666667	0.8
0.3636364	0.72413793	0.36363636	0.363636364	0.244726433	0.291666667	0.8
0.3636364	0.72413793	0.36363636	0.363636364	0.244726433	0.291666667	0.8
0.3636364	0.72413793	0.36363636	0.363636364	0.244726433	0.291666667	0.8
0.3636364	0.72413793	0.36363636	0.363636364	0.244726433	0.291666667	0.8
0.4	0.81172414	0.4	0.4	0.244726433	0.0575	0.184
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0
0	0.82758621	0	0	0.244726433	0	0

There are several ways that one can analyse this data. One of the most useful for this kind of data is the Receiver Operating Characteristic Curve, or the ROC-Curve, calculated by comparing the True Positive Rate and False Positive Rate. Each point on the ROC-Curve is the data measured with a different threshold for RanCorr.

JavaMean forecasting algorithm results

For the “JavaMean” forecasting algorithm, the following graphs; Figure 8.7, Figure 8.8 and Figure 8.7 represent the quality of each test.

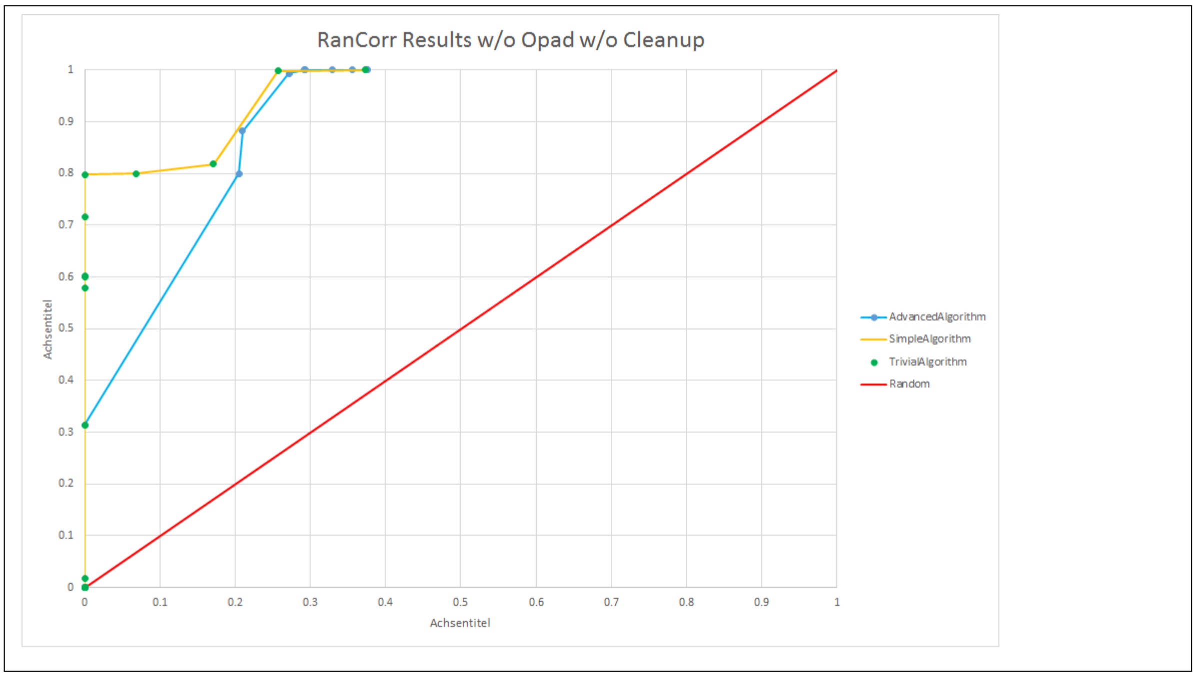


Figure 8.7. ROC Curve for Run with No Opad, No Cleanup, JavaMean Forecasting

Visible in Figure 8.7 is the clear high quality of the results. When the generated anomaly scores are given to RanCorr, it is quite clear that anomalies are easily detected. We have a very high true positive rate and a low false positive rate. It is important to note that the generated anomaly scores are more easily distinguished in general as an anomaly is marked with a high anomaly score. This diagram shows that if given clear values, RanCorr is capable of identifying anomalies and assigning proper anomaly rankings with a high degree of accuracy. All three RanCorr algorithms are on display here, however in all instances, the simple and trivial algorithms return the exact same data. In this particular setting, the trivial and simple algorithms actually outperform the advanced algorithm, but that was to be expected as anomalies are clearly marked by anomaly scores.

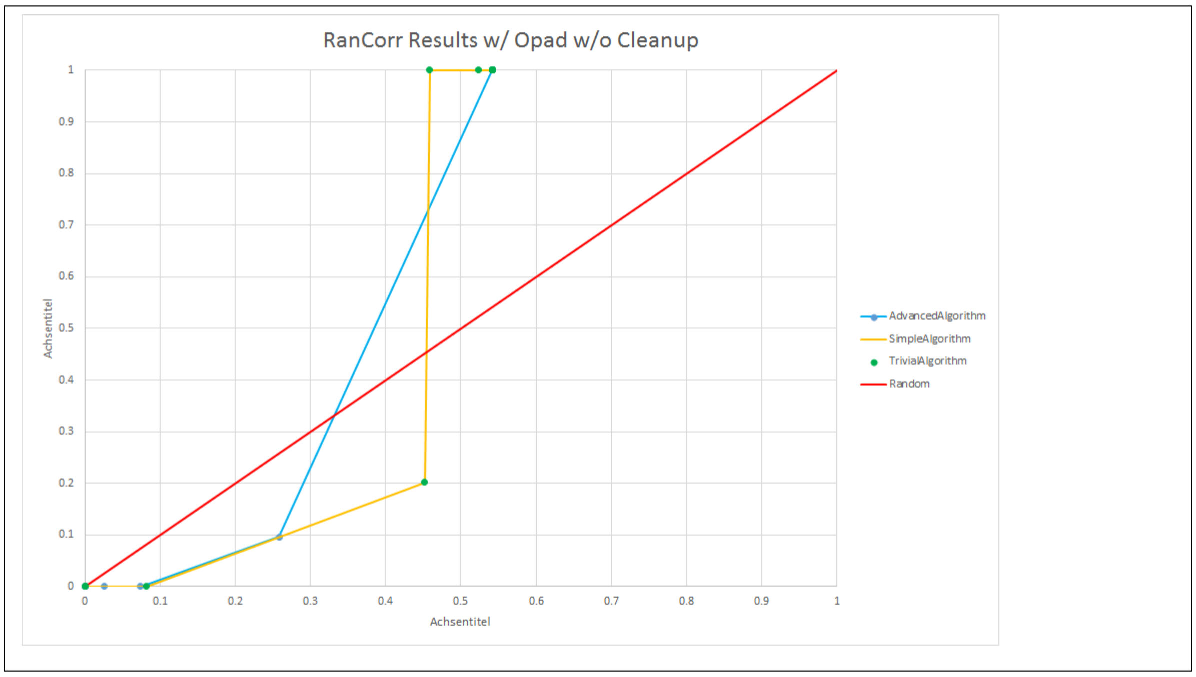


Figure 8.8. ROC Curve for Run with Opad, No Cleanup, JavaMean Forecasting

Figure 8.8 shows a clear drop in accuracy in comparison to the data that is generated without ΘPADx. Much of the data is actually on the 0 line for true positive rate, meaning that none of the injected anomalies were actually detected. Only after the threshold is set high enough does the true positive rate eventually rise over the line indicating random chance. This is attributed to the fact that without cleanup, ΘPADx returns a dataset with a latency of 0 and an anomaly score of 1 for each timeframe in which a method is not detected. While this is in fact desired to capture the anomaly of a program not running at all anymore, it does significantly skew our results. Therefore, for Figure 8.7 the data has been sanitized by removing these sets of data. Also of note here is that all three of the algorithms perform equally poor under this setup.

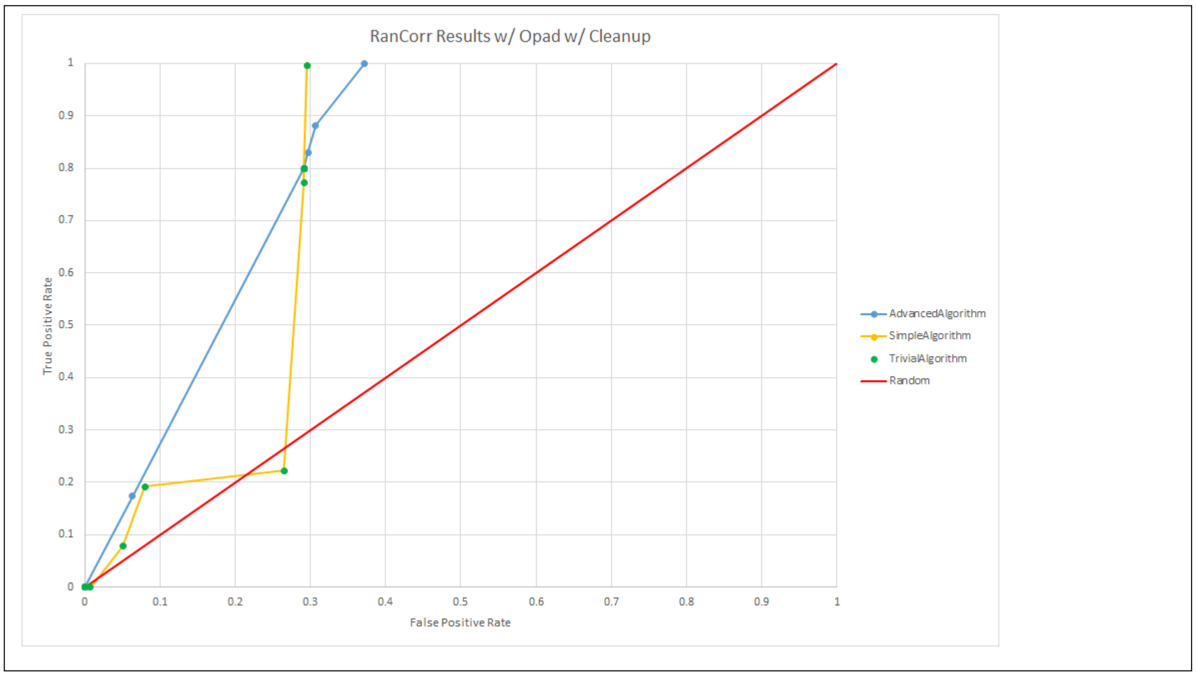


Figure 8.9. ROC Curve for Run with Opad, with Cleanup, JavaMean Forecasting

As already mentioned, this graph displays the sanitized data. The reasoning for this removal of data is explained in Section 8.1.2. It is immediatly clear that the accuracy of the system is increased as compared to the non sanitized data. Again it seems the trivial and simple RanCorr algorithms return the exact same results. It is also appears that the Advanced algorithm outperforms the other two for the most part, never dropping beneath the random chance line.

Also a relevant comparison is the RanCorrClearness as defined in Section 8.1.3.

Table 8.24. RanCorr Clearness

Path	Algorithm	RanCorr Clearness
No Opad No Cleanup	Advanced Algorithm	0.259349077
	Simple Algorithm	0.330827808
	Trivial Algorithm	0.330827808
Opad No Cleanup	Advanced Algorithm	0.265431363
	Simple Algorithm	0.277243307
	Trivial Algorithm	0.277243307
Opad with Cleanup	Advanced Algorithm	0.242408967
	Simple Algorithm	0.275840536
	Trivial Algorithm	0.275840536

As can be seen, in each case the RanCorr accuracy is lower for the advanced algorithm than for the others, and the other two are again identical. The clearness is a measure taken from Marwede et al. [2009].

Extended Self-Tuning forecasting algorithm results

ROC Curves are again useful indicators of how well a setup has performed.

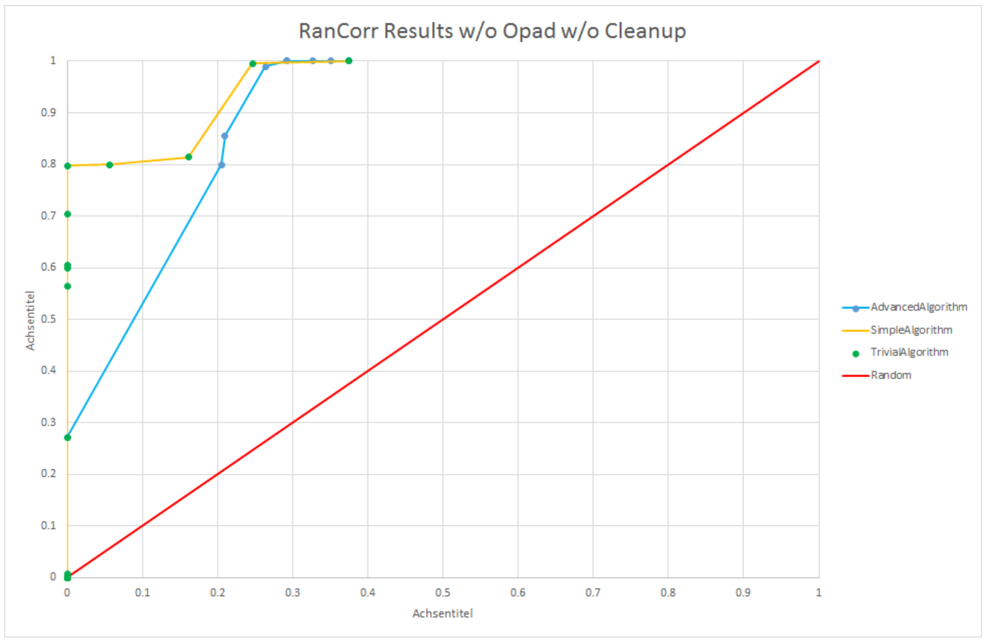


Figure 8.10. ROC Curve for Run with No Opad, No Cleanup, Self-Tuning Forecasting

Figure 8.10 can be compared to Figure 8.7. As can easily be seen, these two graphs are very similar. This is intended, as in the generation of data for both, no ΘPADx was involved. These two exist to allow comparison between using ΘPADx and not using it.

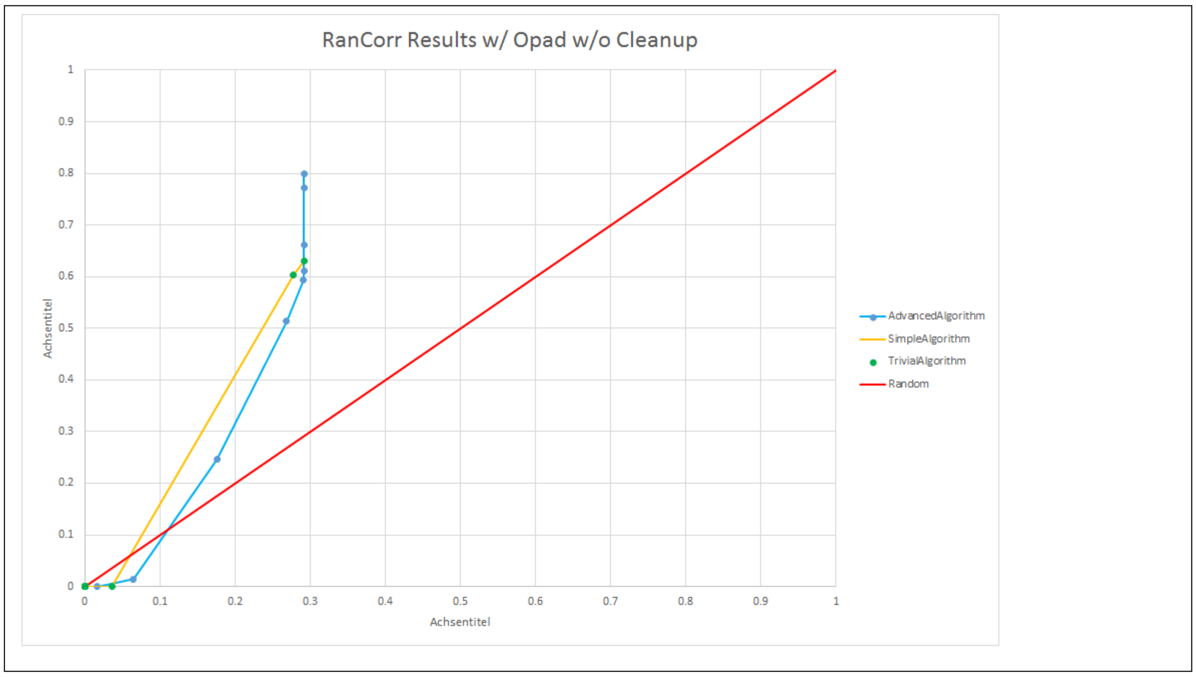


Figure 8.11. ROC Curve for Run with Opad, No Cleanup, Self-Tuning Forecasting

Figure 8.11 should be compared to both Figure 8.10 and Figure 8.8. When compared to the first, we see a clear degradation of accuracy. We have a overall decreased True Positive rate and an increased false positive rate. Similar to using the "JavaMean" forecasting filter, this is to be expected as self generated anomaly scores are higher and segregated clearer from the surrounding non-anomalies.
When we compare Figure 8.11 to Figure 8.8 we can see a clear difference between the two. Whereas the first has a lower general False Positive Rate, it also has a lower True Positive Rate. At this point, mere visual comparison is not sufficient, so an alternative is comparing the area under the curve for each RanCorr algorithm in each setup for each forecasting filter. This will be done in Section 8.1.4.

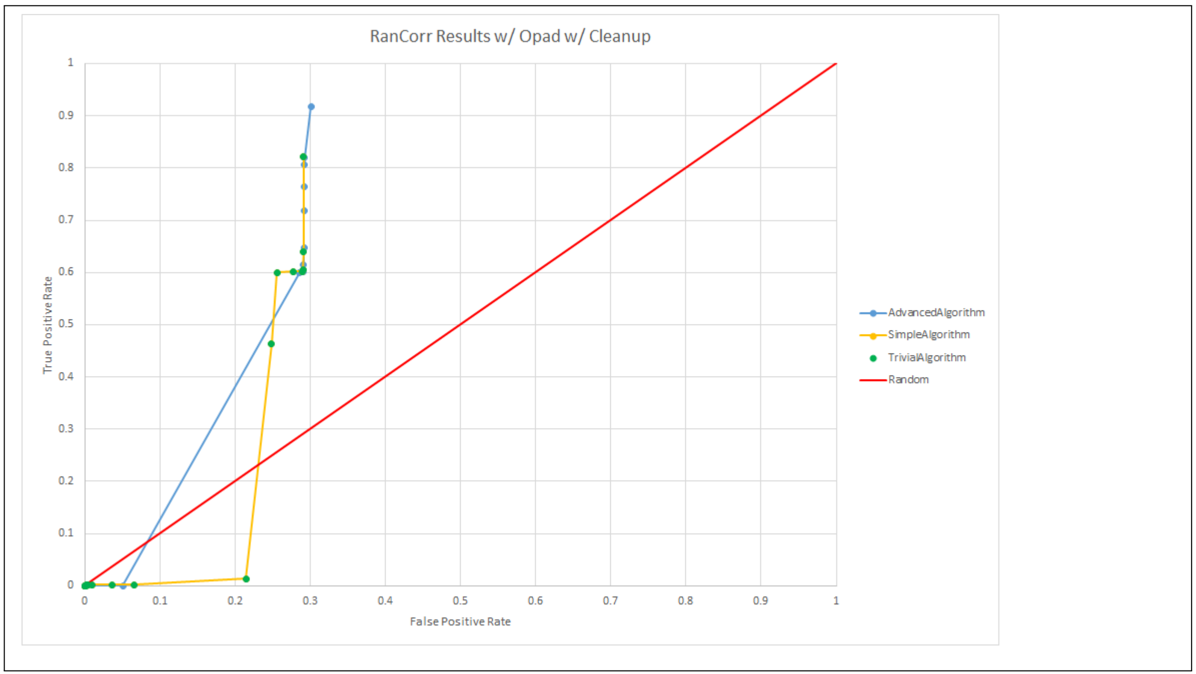


Figure 8.12. ROC Curve for Run with Opad, with Cleanup, Self-Tuning Forecasting

Comparing Figure 8.12 with Figure 8.11 shows a similarity that makes it difficult to distinguish which is performing better. Again, a direct comparison can be drawn using the area under curve in Section 8.1.4.
More definite is the difference between Figure 8.12 and Figure 8.9. It is immediatly visible that the Self-Tuning forecaster seems to do worse in this situation as opposed to the "JavaMean" forecaster. While an area under the curve comparison is necessary the graph speaks for itself.

Direct Comparison

For a numerical comparison of ROC Curves, the area under the curve is a useful tool. The area under the curve signifies the quality of the given values. A result of 1.0 would be a perfect result, 0.5 would be random chance and 0.0 would be completely terrible.
Unfortunately in this case, the plots created do behave slightly different than a regular rock curve. A simple example of this is that they do not end at 1.0 True Positive Rate and 1.0 False Positive Rate. Therefore, there is the question of what to do with the extra space at the end if we want to compare the different results received.
To counteract this, three area under the curves are calculated.

- Direct - Only the area exactly under the curve is considered. The starting point for the X axis is 0.0 and the ending point is the maximum value for X existing. See Figure 8.13
- Completed - An additional plot point is layed at coordinates 1.0, 1.0 for each curve. The area under the curve is then complete.
- Cut-Off. Again, the starting point for the X axis is 0.0 but the endpoint is 1.0. The starting point for the Y axis is 0.0 and the end point for the Y axis is the highest value of Y. The area is then calculated.

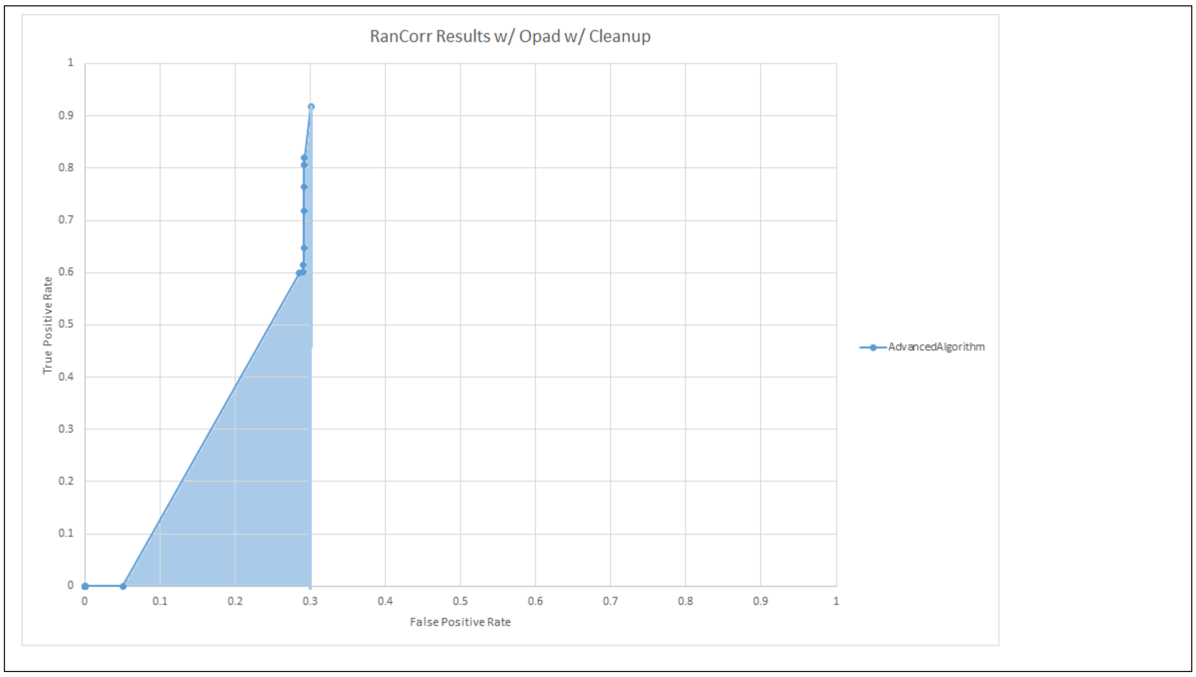


Figure 8.13. Example of direct area under curve.

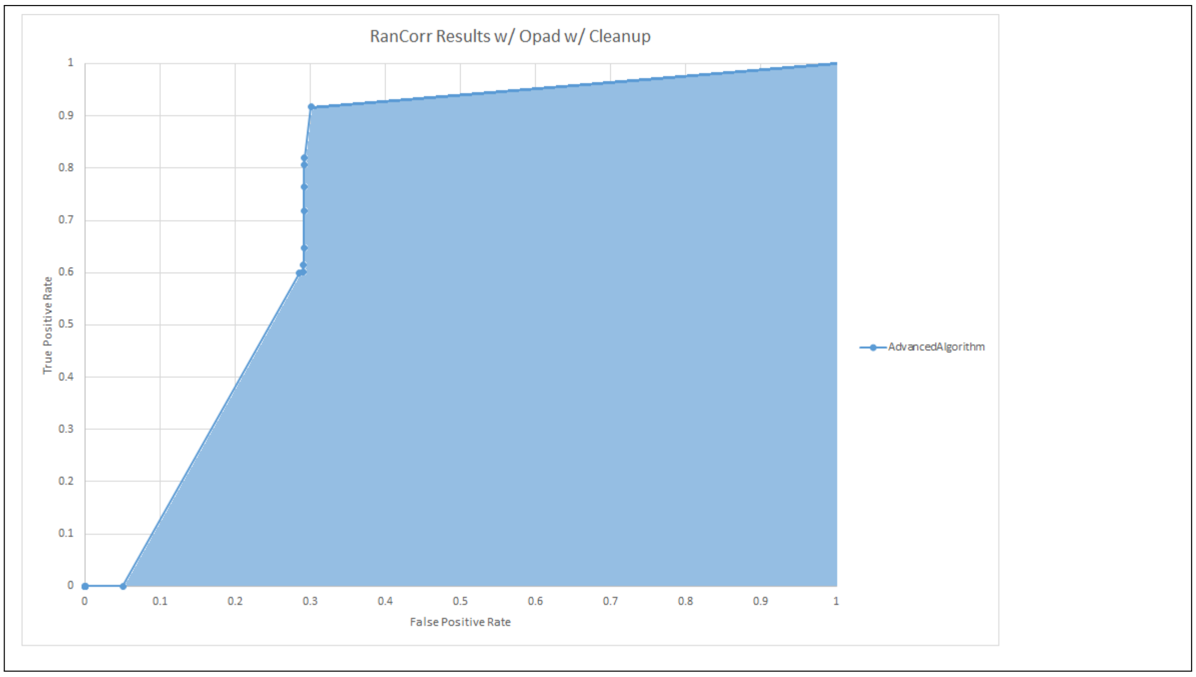


Figure 8.14. Example of completed area under curve.

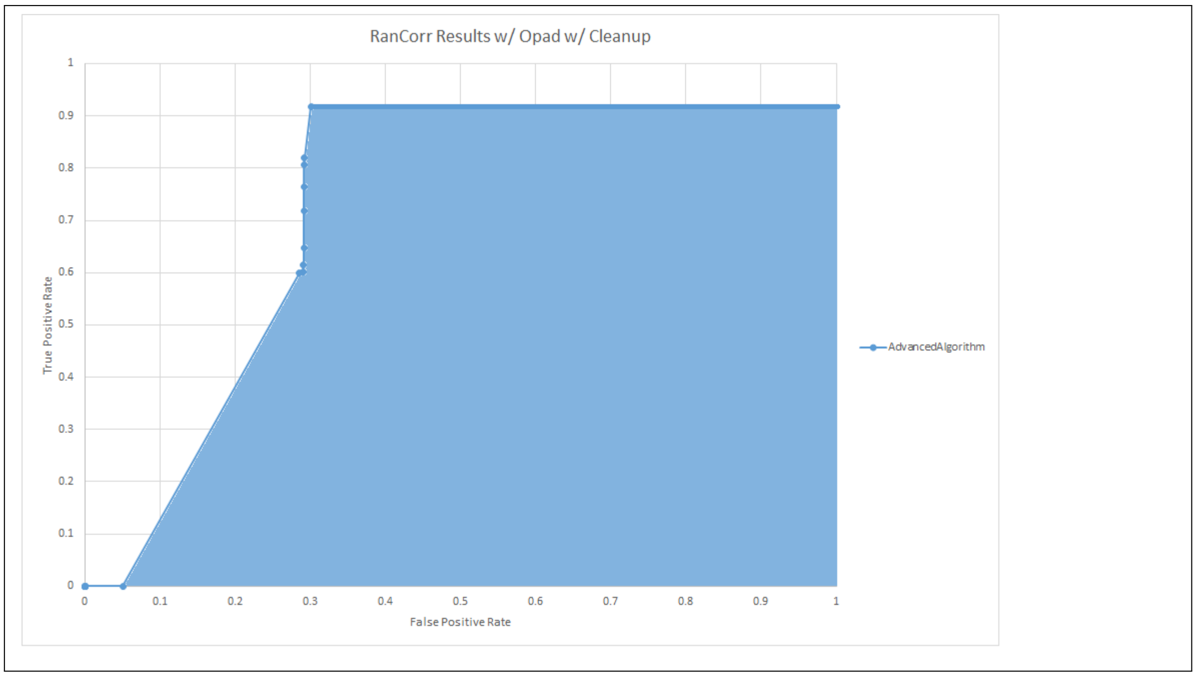


Figure 8.15. Example of cut-off area under curve.

Considering these different calculations for area under the curve, these are the results:

Table 8.25. Area under the curve, Direct

		No Opad No Cleanup	Opad No Cleanup	Opad Cleanup
Self Tuning				
	Advanced	0.2748	0.0628	0.0822
	Simple	0.3347	0.0815	0.0345
	Trivial	0.3347	0.0815	0.0345
Java Mean				
	Advanced	0.2795	0.1643	0.1911
	Simple	0.3320	0.1245	0.0607
	Trivial	0.3320	0.1245	0.0607

Table 8.26. Area under the curve, Completed

		No Opad No Cleanup	Opad No Cleanup	Opad Cleanup
Self Tuning				
	Advanced	0.8998	0.7712	0.7814
	Simple	0.9601	0.7898	0.7437
	Trivial	0.9601	0.7898	0.7437
Java Mean				
	Advanced	0.9045	0.6226	0.8190
	Simple	0.9587	0.5828	0.7653
	Trivial	0.9587	0.5828	0.7653

Table 8.27. Area under the curve, Cut-Off

		No Opad No Cleanup	Opad No Cleanup	Opad Cleanup
Self Tuning				
	Advanced	0.8998	0.6295	0.7241
	Simple	0.9601	0.5277	0.6175
	Trivial	0.9601	0.5277	0.6175
Java Mean				
	Advanced	0.9045	0.6226	0.8190
	Simple	0.9587	0.5828	0.7625
	Trivial	0.9587	0.5828	0.7625

Each of the tables Table 8.27, Table 8.26 and Table 8.25 should stand as a fair comparison between the parameters that have been chosen in Section 8.1.2 except for the threshold.

The threshold is contained within each of these results already and is iterated over in an identical fashion in each individual result. Represented are the path chosen (as described in Section 8.1.2), the forecasting algorithms and the RanCorr algorithms.
What should be noted is that the "Direct" approach to the area under the curve does not give us a useful result. In a ROC curve, an area under the curve of 0.5 is considered the be the score that a random choice algorithm would score. Yet, despite that in the diagrams it can cleary be seen that the results are significantly over the random line, the score in the "Direct" area under the curve measurement does not reflect this in Table 8.25. Therefore this is not a useful measurement.
The "Cut-Off" method delivers what can be considered the worst case scenario. If future measurements delivered results that would only increase the False Positive Rate with a static True Positive Rate. The alternative would be the best scenario, where the True Positive Rate would climb to one without the False Positive Rate increasing.
Therefore, the "Completed" method would deliver an average result, between best and worst case, where both the True Positive and the False Positive rates approached 1.0. We can consider both the "Completed" and the "Cut-Off" methods to return useful data.

Christopher Gregorian

8.1.5 Conclusion

The goal of the entire experiment setup was to answer the questions posed in Section 8.1.1. Therefore it is fitting to look how our results answer each individual question.

Question 1.1

How well does Kiekeriki find injected anomalies?
To discuss the quality of anomaly detection, one should turn to the area under the curve method. Table 8.26 and Table 8.27 display values for several areas under the curve.
When no ΘPADx is used at all, the area under the curve stands near or above 0.9, showing a very good result. When ΘPADx is generating anomaly scores, a marked drop in results is clearly visible. In the worst case (Cut-Off area under the curve Table 8.27) the result stands between 0.52 and 0.63 at a score much closer to random. This is a very poor result and stems mostly from the fact that, as discussed in Section 8.1.2, ΘPADx will generate high anomaly scores of 1.0 for times when a method is not detected at all.
However, when these anomalies are removed (under "Opad Cleanup") and only the intended anomalies remain, the detection rate climbs quite steadily. From 0.61 in the worst case to 0.81. Und what is considered an average case, the result is between 0.74 and 0.81 (Table 8.26).
By assessing these results, it is clear that Kiekeriki can detect anomalies quite well and the low scores under "Opad No Cleanup" stem not from Kiekeriki itself, but from a known problem with the data generation.

Question 1.2

How does the RanCorr algorithm type affect the output?
One of the most obvious conclusion that can be drawn is that apparently the Simple and Trivial algorithms are not in any way different. Whether this is a flaw in the algorithms themselves, the data generation or the implementation is unknown. The results however are identical for every single run.
As for the difference between the Advanced algorithm and the others it is again necessary to look at the worst case scenario Table 8.27. It is shown that when no ΘPADx is running, the Simple/Trivial algorithms outperform the Advanced algorithm by a margin of 0.06 or 6%. However, once ΘPADxis used in either its native or cleaned up form, the Advanced algorithm outperforms its counterpats by 10% or more. In the average case (Table 8.26) the difference becomes more subtle with only 3.7%. The conclusions that can be drawn are quite simple. The implementations of the Trivial and Simple algorithms will perform exactly identical each time and in the relevant scenario (ΘPADx creating data) the Advanced algorithm will outperform the others by a sizeable margin or be just about equal in performance.

Question 1.3

How do differing thresholds and configurations change our anomaly detection?
Looking further back at Section 8.1.3 it can be seen that depending on the situation, the different thresholds will return very different results. To evaluate these properly, it is necessary to find the best True Positive Rate to False Positive Rate Ratio, the larger the ratio the better.

Table 8.28. Best Thresholds for each setup

		No Opad No Cleanup	Opad No Cleanup	Opad Cleanup
Java Mean	Advanced	0.40	NONE	0.10
	Simple	0.20	0.75	0.05
	Trivial	0.20	0.75	0.05
Self-Tuning	Advanced	0.40	0.05	0.05
	Simple	0.20	0.10	0.05
	Trivial	0.20	0.10	0.05

While difficult to judge the best overall threshold in Table 8.28, it is most likely that the ones under the column "Opad Cleanup" the most useful ones, as theser are the most realistic conditions.

Question 1.4

What results does the new Self-Tuning algorithm deliver?
Determined by data in Table 8.26 and Table 8.27 (worst case scenario) it can be said that the implementation of the Self-Tuning algorithm still requires some work. The two relevant columns are "Opad No Cleanup" and "Opad Cleanup" since "No Opad No Cleanup" is irrelevant here. Mostly the Self-Tuning forecaster performs similar to the "JavaMean" filter, yet in one instance ("Opad Cleanup") it underperforms significantly by almost 9% with the Advanced Algorithm. The other negative is that in the situation "Opad No Cleanup" it actually outperformed the "JavaMean" forecaster. Unfortunately, it would be expected to performy poorly here since a large amount of unwanted anomalies are injected by the data generation (datasets with 0.0 latency and 1.0 anomaly score). These erronous anomalies should have skewed the result much further. Clearly, this implementation of the Self-Tuning forecaster is not complete.

Question 2.1

What impact does using ΘPADx have as opposed to using self generated test data with anomaly scores?
The answer to this question is quite simple. Looking at either Table 8.26 and Table 8.27 and comparing the "No Opad No Cleanup" column to either "Opad No Cleanup" or "Opad

Cleanup" will quickly reveal that the area under the curve is much larger for the first. The reasons for this are quite simply that any anomalies are very clearly offset from non-anomalous data. Therefore it is much easier for RanCorr to identify them.

Quesiton 2.2

How does post-processing ΘPADx data for clarity change the outcome?
The answer can be clearly seen by studying Table 8.27. By comparing the two columns "Opad No Cleanup" and "Opad Cleanup" the difference is quite clear. With the Self-Tuning forecaster an improvement of about 9% is made. Wheras with the "JavaMean" forecaster an astonishing 17% to 19% improvement is made. Clearly, removing the faulty anomalies has created a much better result.

Christopher Gregorian

8.2 Benchmarks of the Database Connections

This section describes the benchmarks of the database connections of the Kiekeriki Frontend and Backend, how to conduct it, the evaluation and which consequences can be derived from these results.

8.2.1 Evaluation Goals

The benchmarks of the database connections should validate the goals described in the Figures 8.16 and 8.17 and the Tables 8.29 and 8.30.

Table 8.29. Goal 1: write throughput

Goal	G1	High write throughput from the Kieker Backend to the Transfer Database
	Purpose	Prediction of the writing throughput from the Backend to the Transfer Database
	Issue	Performance
	Object	Database connection of the Frontend to the Transfer Database
	Viewpoint	Developer
Question	Q1.1	Which is the average latency of writing one measurement to the database?
	Q1.2	Which is the average latency of writing the most expensive measurement to the database?

Table 8.29. Goal 1: write throughput

Goal	G1	High write throughput from the Kieker Backend to the Transfer Database
	Q1.3	Which is the average latency depending on the batch size of the writing of multiple measurements to the database?
	Q1.4	Which is the average latency depending on the batch size of the writing of multiple of the most expensive measurement to the database?
Metrics	M1	Latency in milliseconds
Question	Q1.5	Which average writing throughput can the database handle with all inputs in one second?
	Q1.6	Which average writing throughput can the database handle with the most expensive input in one second?
Metrics	M2	Number of write executions per second

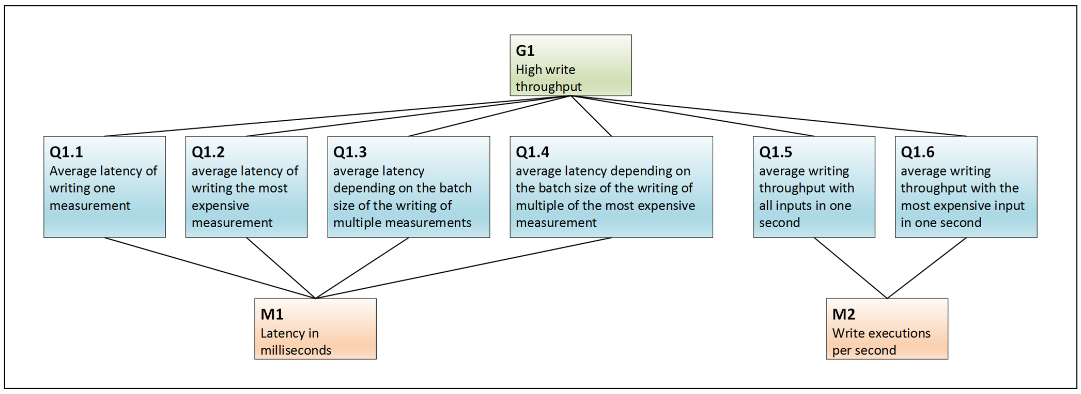

G1
High write throughput
Q1.1
Average latency of writing one measurement
Q1.2
average latency of writing the most expensive measurement
Q1.3
average latency depending on the batch size of the writing of multiple measurements
Q1.4
average latency depending on the batch size of the writing of multiple of the most expensive measurement
Q1.5
average writing throughput with all inputs in one second
Q1.6
average writing throughput with the most expensive input in one second
M1
Latency in milliseconds
M2
Write executions per second

Figure 8.16. Goal 1: write throughput

Table 8.30. Goal 2: read throughput

Goal	G2	High read throughput from the Kieker Frontend to the Transfer Database
	Purpose	Prediction of the reading throughput from the Frontend to the Transfer Database
	Issue	Performance
	Object	Database connection of the Frontend to the Transfer Database
	Viewpoint	Developer
Question	Q2.1	Which is the average latency of reading one information from the database?
	Q2.2	Which is the average latency of reading the most expensive information from the database?
Metrics	M1	Latency in milliseconds

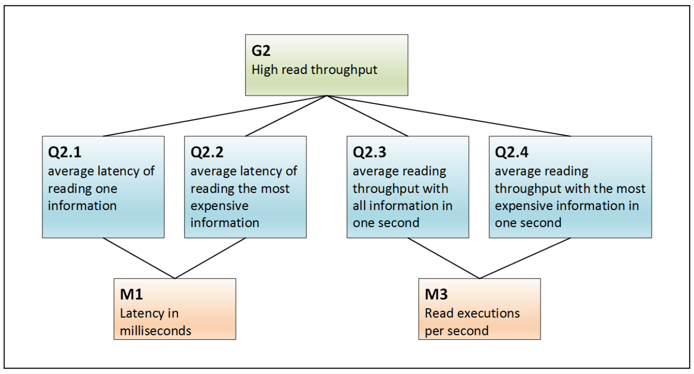

G2
High read throughput
Q2.1
average latency of reading one information
Q2.2
average latency of reading the most expensive information
Q2.3
average reading throughput with all information in one second
Q2.4
average reading throughput with the most expensive information in one second
M1
Latency in milliseconds
M3
Read executions per second

Figure 8.17. Goal 2: read throughput

Christian Endres

8.2.2 Experimental Design

This section describes how to build experiments which provide data to answer the questions defined in 8.3.1. To acquire the data on which the questions can be answered, there is an application which uses the Transfer Database in an automated, randomized way and writes the data into files which are analysed in section 8.2.3 and 8.2.4.

Hardware setup

The project uses a server at the data center of the Hetzner Online AG for the experiments. The Hetzner server has an Intel®Core i7-920 Quadcore processor, 48 GB DDR3 RAM and two 2 TB SATA 3Gb/s Enterprise HDDs in a raid 1 configuration.

Software setup

The Hetzner server runs a Debian 3.2.51-1 x86_64 GNU/Linux, Java version 1.7.0_51 and a MySQL-Server 5.5.35.

Experimental setup

To answer the questions of 8.3.1, there is a load driver (see 7.4) which stresses the database. The load driver uses one connection to the database and has the (slightly adjusted) methods of the Kieker Backend and the Kieker Frontend. The load driver is executed on the same Hetzner server as the Transfer database and the application server on which the Kieker Frontend is running.

Benchmark execution

For the execution of the benchmarks the KiekerTransferDBDataGenerator uses a benchmarking framework [Boyer, 2008a,b,c] which agrees with [Georges et al., 2007] about how to conduct statistically rigorous benchmarks of Java applications. The framework cares for everything like the heating phase before the measurements for example. A benchmark executes exactly 60 measurements rounds with multiple single measurements each round. If the compiling time differs, the benchmark framework restarts and starts another 60 measurement rounds until all runs without any cause for restart again.

Christian Endres

8.2.3 Analysis of the Benchmark Results for the Kiekeriki Backend

Figure 8.18 shows the latencies of each method which writes to the Transfer Database. Each method has many outliers which can be even 10 times higher than the median. This can be a result of many different system behaviors which the benchmark was still prone to. Thus the questions Q1.1 cannot be answered definitely. Question Q1.2 asks for the most expensive write operation, which cannot be determined, too. The medians of each writing method are nearly the same and the box plots show that the second and third percentiles are overlapping.

The question Q1.3 can be answered for higher batch sizes by the Figure 8.19 and Table 8.31, though the measurements are prone to the outliers. This is the result of the high standard deviation of the write executions as learned from the single write benchmarks. The values of the column set “Mean per data set” indicates a decreasing mean execution time per data set for increasing batch size. Though the measurements with a batch size of 50 contradicts. The

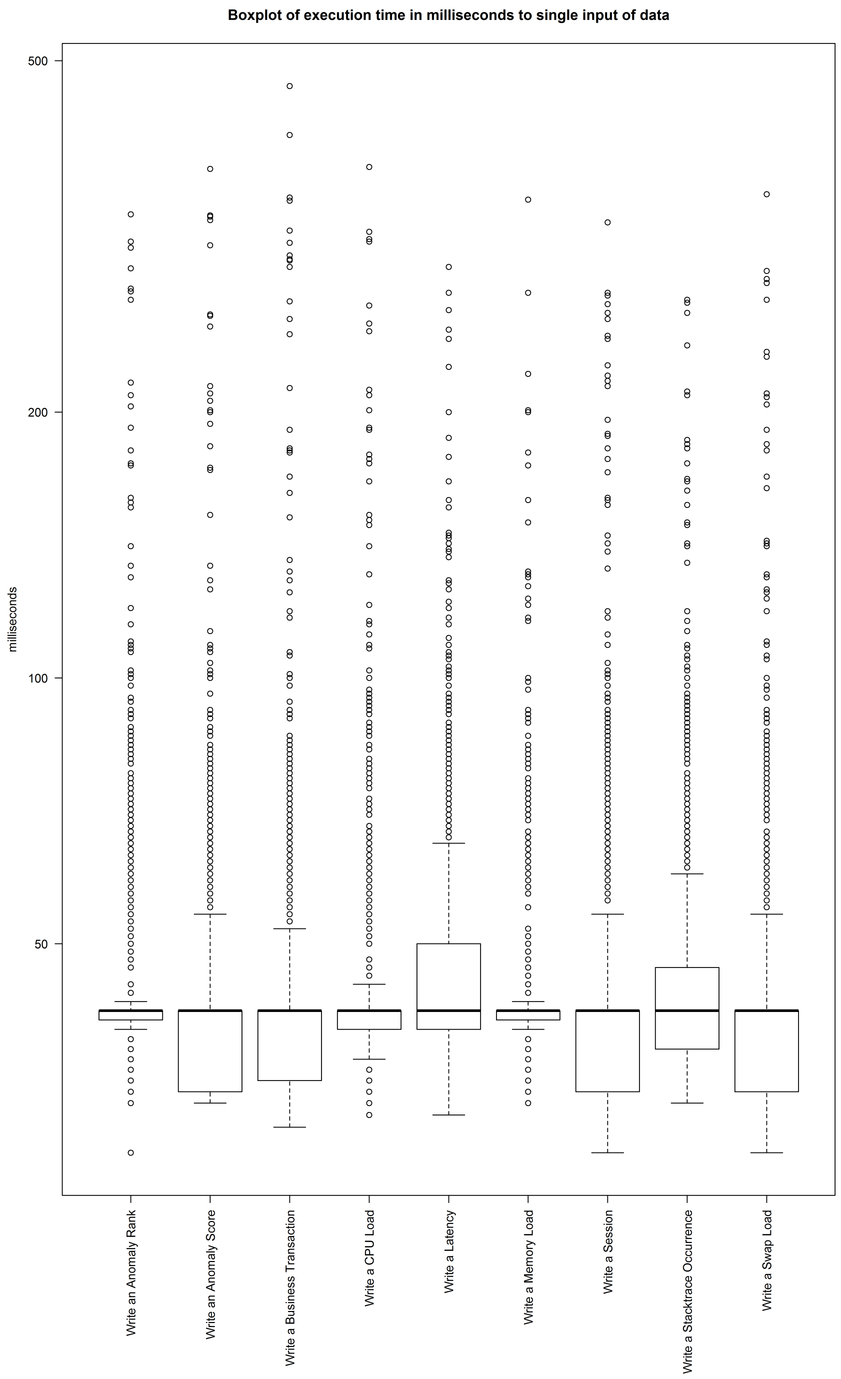


Figure 8.18. Write benchmark of each write methods with a single data set

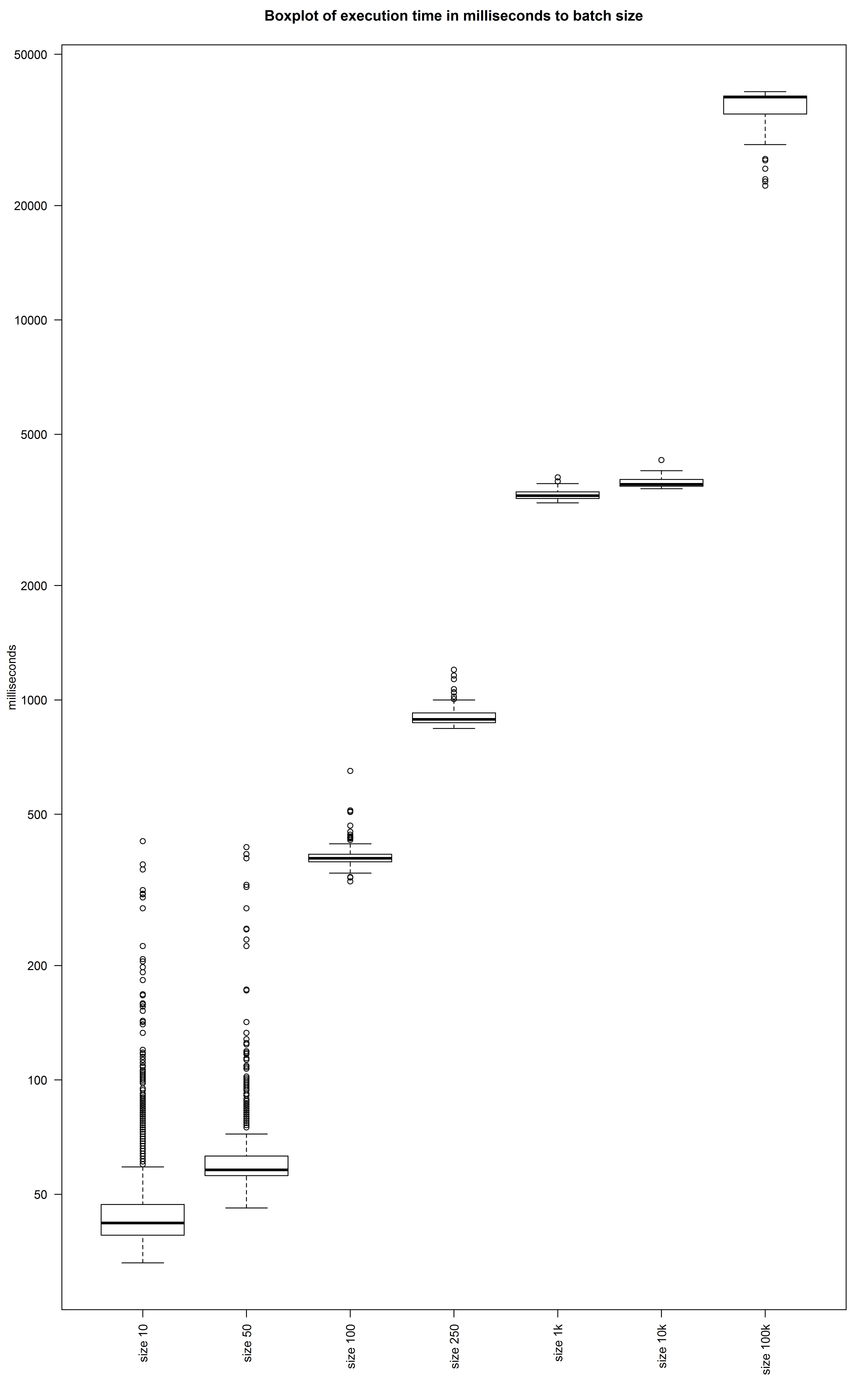


Figure 8.19. Write benchmark of all write methods with large data set sizes

measurements with the batch sizes of 50, 10,000 and 100,000 hypothesise different system behaviors like optimization for specific batch sizes than the other measurements.

The questions Q1.4 and Q1.6 cannot be answered because it could not be determined which is the most expensive write operation.

The question Q1.5 can be answered by the figure 8.19 which indicates a batch size of 250 data sets is very near the optimum. For further investigation of the best batch size there has to be specified if there are outliers with higher execution times are allowed and if yes how many.

Table 8.31. Write benchmark results of all write methods with large data set sizes, values are measured in milliseconds.

Data sets in batch	Median	Mean	Standard Deviation	Median per data set	Mean per data set
10	42	49.517	27.513	4.2	4.952
50	58	65.843	30.933	1.16	1.317
100	383	392.392	38.606	3.83	3.924
250	889	906.717	60.814	3.556	3.627
1,000	3446.5	3472.483	117.546	3.447	3.472
10,000	3693.5	3742.033	132.864	0.369	0.374
100,000	38577.5	36157.317	4647.078	0.386	0.362

Christian Endres

8.2.4 Analysis of the Benchmark Results for the Kiekeriki Frontend

The table 8.32 shows the results of the analysis of the benchmark data of the Kiekeriki Frontend. Each functionality represents a method in the database connection to the Transfer Database which retrieves the requested data and processes it into the internal data model. The data row shows how much data is stored in the database when the benchmark was run. Its number indicates a multiple of 1,000 data sets which is retrievable for each method except the branch to the root, raw data field description and raw data select statements functionality which always return the same number of results and could only be slowed down by the overall database size. The values are measured in milliseconds.

Table 8.32. Statistics of the read functionality of the Kiekeriki Frontend to the Transfer Database.

Functionality	Data	Median	Mean	SD
1. called component IDs	1	0	0.089	0.286
	10	0	0.242	0.428
	100	0	0.244	0.43
2. get calls in time range	1	3	3.380	0.536
	10	16	16.398	1.337
	100	192	192.808	2.462
3. get component	1	12	12.009	1.616
	10	124	124.560	2.25
	100	1364.5	1367.7	8.88
4. get component IDs of path to root	1	1	0.696	0.962
	10	1	0.595	0.814
	100	1	0.55	0.742
5. get component ranking	1	1	0.537	0.936
	10	1	0.513	0.713
	100	1	0.516	0.525
6. get component ranking in time range	1	3	3.240	0.49
	10	18	18.595	2.864
	100	176	177.523	7.33
7. get component total calls	1	0	0.417	0.629
	10	0	0.457	0.602
	100	0	0.282	0.45
8. get CPU load	1	5	5.536	3.328
	10	36	36.568	2.654
	100	334	334.567	7.58
9. get CPU load in time range	1	5	5.366	0.591
	10	33	33.430	4.122
	100	335	338.088	9.014
10. get hosts	1	1	0.556	0.531
	10	2	2.428	0.71
	100	24	25.501	2.744
11. get latencies	1	5	4.742	0.563
	10	59	58.931	1.346
	100	654.5	655.883	5.975

Table 8.32. Statistics of the read functionality of the Kiekeriki Frontend to the Transfer Database.

Functionality	Data	Median	Mean	SD
12. get latencies in time range	1	5	4.908	0.381
	10	59	59.565	2.289
	100	661.5	672.025	24.812
13. get memory load	1	6	5.8775	3.563
	10	38	38.776	3.769
	100	325	326.933	8.32
14. get memory load in time range	1	6	5.786	0.602
	10	37	38.453	3.652
	100	299	301.504	8.632
15. get operation anomaly scores	1	3	3.233	1.552
	10	24	24.786	1.987
	100	226	226.240	6.338
16. get operation anomaly scores in time range	1	3	3.249	0.533
	10	24	25.001	1.963
	100	225	225.469	6.263
17. get operation calls	1	1	0.977	0.299
	10	7	6.758	1.249
	100	44	44.65	3.515
18. get operation children	1	301	319.358	60.437
	10	2161	2171.733	27.315
	100	21008	21097.6	267.597
19. get raw data fields and description	1	0	0.18	0.388
	10	0	0.149	0.356
	100	0	0.174	0.379
20. get raw data select statement	1	0	0.176	0.381
	10	0	0.165	0.371
	100	0	0.091	0.287
21. get session IDs of application in time range	1	3	2.837	0.442
	10	24	23.700	1.869
	100	216	217.090	5.34
22. get session IDs of host in time range	1	4	4.153	2.45
	10	24	23.906	1.897
	100	216	217.256	5.168

Table 8.32. Statistics of the read functionality of the Kiekeriki Frontend to the Transfer Database.

Functionality	Data	Median	Mean	SD
23. get stacktraces	1	3	3.198	2.145
	10	26	26.666	2.434
	100	226	226.496	5.484
24. get stacktraces in time range	1	4	4.429	0.735
	10	30	29.768	2.189
	100	247	247.860	5.19
25. get swap load	1	5	5.090	1.255
	10	37	37.825	3.733
	100	314	316.479	8.566
26. get swap load in time range	1	6	5.737	0.67
	10	38	39.011	3.585
	100	298	299.4	8.059

The question Q2.1 is answerd with the table 8.33. The high standard deviations is due to the "get all operation children" method with the number 18 in the table 8.32 which operates recursively on the component tree. Question Q2.2 is answered aswell with the table 8.32 which states the average latencies of the 18th method. The measurements show a decreasing latency for multiple requests which can be due a caching of the MySQL-Server.

Table 8.33. Statistics about single read operations.

Median	Mean	Standard Deviation
3	14.846	58.425

The next questions to evaluate would be how many components the Kiekeriki Frontend can handle for a specific power and setup of the hosting server in respects of the user experience. Another question would be how much the performance can be increased due multiple concurrent database connections.

Christian Endres

8.2.5 Conclusion

Due the benchmarks we learned much about our system. The Kiekeriki Backend has limitations due the database connection if there are much data sets to be written critical in terms of time. If there is a need to display these data in a monitoring context and not in an analysis context, there is limitation of the batch size which can be written in one second for example. Thus one possible step to improve the system is to introduce database connection pooling.

The Kiekeriki Frontend or the Jetty which runs the WebGUI has limitations of handling all the data in respect of the user experience. To improve the performance, the data aggregation logic can be moved from Java classes to Stored Procedures to improve the reading performance.

Christian Endres

8.3 Usability Study of the WebGUI conducted with APM-Experts

As it is hard to measure soft goals such as usability or the degree of intuition computationally, the frontend-team conducted a short usability study in the end of the development phase. Two experts of the NovaTec GmbH (Stefan Siegl and Matthias Huber) with knowledge of many current APM software solutions on the market kindly agreed to be participants in our study. This section describes our intended goals for this study, how it was conducted and its results.

Anton Scherer

8.3.1 Evaluation goals

The expert feedback should answer the questions of table 8.34.

Table 8.34. Evaluation goals of the expert feedback

Goal	G1	High usability of the Kiekeriki Frontend
	Purpose	Identify enhancement potential of the Kiekeriki Frontend
	Issue	Usability
	Object	Kiekeriki WebGUI
	Viewpoint	User

General		
Question	Q1.1	The application is visually appealing
	Q1.2	The overall organization of the GUI is easy to understand
	Q1.3	Individual pages are well designed
	Q1.4	The terminology of the website met my expectations
	Q1.5	The overall performance (load times) of the GUI is appropriate

ArchView		
Question	Q1.6	The ArchView shows hosts well arranged offering a good overview
	Q1.7	Deep Diving into components is easy and intuitive
	Q1.8	Switching between component levels (e.g. going to previous components)

Diagrams		
Question	Q1.9	The visualization of the data through diagrams is pleasant
	Q1.10	The diagrams show meaningful data, so the user gets important information about the monitored application
	Q1.11	Default diagrams are appropriate on each level
	Q1.12	The configuration dialog for adding new diagrams is intuitive providing the configuration of all parameters I had expected
	Q1.13	Overall, adding and deleting diagrams is easy

Misc		
Question	Q1.14	The List View is helpful for getting an overall picture of the application's architecture
	Q1.15	Defining new user metrics is easy and intuitive

Closing questions		
Question	Q1.16	I was able to complete my tasks in a reasonable amount of time
	Q1.17	My overall impression of the Web GUI is
	Q1.18	The degree of completeness of the GUI is
Metrics	M1	Expert feedback

Christian Endres

8.3.2 Conducting the usability test

According to a lecture of the University of Washington ([**?**]), a typical usability test consists of the four main steps we complied with in our study:

1. **Find representative users**

   Since the WebGUI is not designed to be used by standard end users but system administrators, experts are necessary for our study. Because APM experts are expected to have a great knowledge about how a monitoring WebGUI is structured, they are able to give a qualified feedback. Furthermore, if they have problems using specific features of the GUI, it is highly probable that the majority will not get this feature to work correctly.

2. **Ask the users to perform representative tasks**

   For this step, the frontend-team (represented by Christian Endres and Anton Scherer) prepared some use cases the participants had to go through. For this purpose, the underlying database was prepared to contain synthetical data, where the use cases can be conducted on. However, before the test persons started to interact with the GUI, the frontend team gave a short introduction about the context, the generated data and the used metrics. After that, the following use cases were given as tasks to be performed:

   - Deep dive into any method level
   - Add an arbitrary diagram showing historical data
   - Define a new user metric
   - Create a new diagram that displays the newly defined metric with live data
   - Create a new diagram which shows the anomaly score of ΘPAD
   - Change options of an existing diagram
   - Find an anomaly by showing the matching anomaly rank

   After the use cases were executed, the test persons had time to use the GUI in an explorative way with no restrictions about what they were doing.

3. **Oberserve what the users do**

   While the two experts were using the web application, their behavoir was carefully analyzed by the examiners. If problems arose during interaction with the GUI, they were written down. Also communication between the two test persons was recorded in order to know which sections in the GUI caused unclarity.

4. **Get feedback and summarize the results**

   Most feedback was given while the test persons were interacting with the WebGUI. If questions arose, they were asked immediately. Often, the experts made improvement suggestions for the detected weakness points. In addition to the consecutive feedback, the frontend team handed a questionnaire out to the experts containing 18 questions for rating the GUI. We emphasized that this judgement will not affect the project's rating in order to get honest assessments. The questionnaire's results are illustrated in 8.3.3. Finally, a last feedback round finished the usability study.

Anton Scherer

8.3.3 Usability Study Results

This section reveals the results of the usability study. The experts' feedback was constructive as they did not just criticize but also suggest how to do it better. The following table structures the feedback the experts gave to optimize the GUI. The frontend team evaluated the tasks and decided which of these suggestions can be implemented within a reasonable amount of time. Due to time pressure, we had to reject some tasks which are considered complex. These are referenced in the future work (Chapter 9). All other optimizations were implemented in the last sprint.

Category	Task	Should be fixed?
Configure Dialog	Delete the metric ID out of the dialog	Yes
	Provide more information in the metric selection	Yes
	Find another solution for the metric caroussell (not intuitive)	No
	Line break for description field	No
	Cancel button does not always work	No
	Resizing the window does not work	Yes
	Position and size of the dialog must not oversize the screen	Yes
	The time unit selection is not intuitive	No
	Provide default values in every selectable field	No
	Explicitly show the level on which the dialog should be added	No
Metrics definition	Place "Add new user metric"-button on top of the table	Yes
	The raw data table must be placed below	Yes
	Delete the percentile option as it does not work correctly	Yes
	Define which function makes sense on which metric	No
	Close the dialog after a new metric was added	Yes

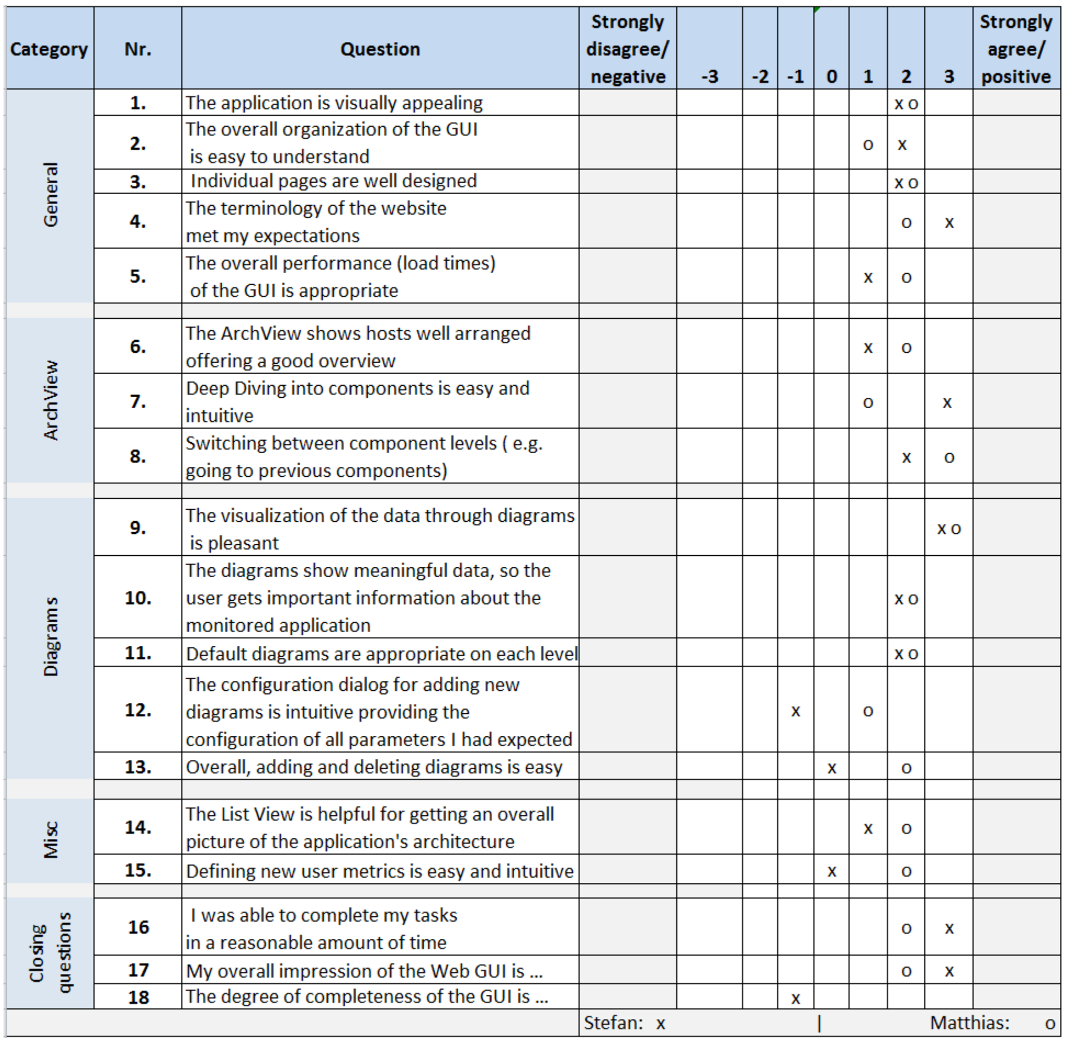
	Show the level in which the metric is able to be selected	No
Architecture view	Provide the component level in an extra label above the architecture view	No
	Tthe graph must be more static (no jumping)	No
	Design a button which renders optimally	No
	The visualization time of an anomaly must be extended appropriately	No
	Visualization is only interesting on host, app and package level	No
Diagrams view	Highlight the "add new diagram"-button	Yes
	The export function does currently only work online	No
	The line area chart of ΘPADxoverlays the score	No
	Create a new option for enlarging diagrams	No
	Implement a drag-and-drop solution for diagrams	No
	The close button of the diagrams is not intuitive	Yes
	The session metric has a bug on application level	Yes
	Do not show empty diagrams but the value 0	Yes
General	Provide an option for defining a global time range for all diagrams	No
	In addition to the standard view a new highly configurable user-view should be provided	No
	Create an anomaly table with all occurring or occurred anomalies	No
	Create a new option for enlarging diagrams	No
	Implement a business transaction and traces view	No
	Implement an end user experience view	Yes
	After marking something (e.g. multiple diagrams) it does not unmark after a click	No

Table 8.35. Experts suggestions for improvement

Furthermore, we were interested in a valuable judgement from experts. For this purpose we mixed fundamental usability questions (e.g. is the application visually appealing?) with specific questions regarding different components of the GUI. Finally, we formulated closing questions for getting an overall impression. The rating range spans from -3 (strongly disagree/negative) to +3 (strongly agree/positive) in order to give the experts the possibility to rate granularly. Table Figure 8.20 illustrates the results.

Basically, both experts rated the various aspects of the GUI consistenly positive. Overall, their average rating value is +1,55 (Stefan) and +1,66 (Matthias). According to the test persons, especially the configuration dialog for adding diagrams has potential for improvement regarding usability. That is why question number 12 has the lowest rating points (we fixed some issues afterwards). In contrast, the visualization of the data through diagrams (question number 9) received an outstanding rating. According to the experts, the overall impression of the GUI is highly positive without any claim for completeness. Matthias abstained from voting the last question.

Anton Scherer



Category	Nr.	Question	Strongly disagree/ negative	-3	-2	-1	0	1	2	3	Strongly agree/ positive
General	1.	The application is visually appealing							x o		
	2.	The overall organization of the GUI is easy to understand						o	x		
	3.	Individual pages are well designed							x o		
	4.	The terminology of the website met my expectations							o	x	
	5.	The overall performance (load times) of the GUI is appropriate						x	o		
ArchView	6.	The ArchView shows hosts well arranged offering a good overview						x	o		
	7.	Deep Diving into components is easy and intuitive						o		x	
	8.	Switching between component levels (e.g. going to previous components)							x	o	
Diagrams	9.	The visualization of the data through diagrams is pleasant								x o	
	10.	The diagrams show meaningful data, so the user gets important information about the monitored application							x o		
	11.	Default diagrams are appropriate on each level							x o		
	12.	The configuration dialog for adding new diagrams is intuitive providing the configuration of all parameters I had expected				x		o			
	13.	Overall, adding and deleting diagrams is easy					x		o		
Misc	14.	The List View is helpful for getting an overall picture of the application's architecture						x	o		
	15.	Defining new user metrics is easy and intuitive					x		o		
Closing questions	16	I was able to complete my tasks in a reasonable amount of time							o	x	
	17	My overall impression of the Web GUI is ...							o	x	
	18	The degree of completeness of the GUI is ...				x					
			Stefan: x			|			Matthias:		o

Figure 8.20. Results of the feedback questionnaire

Chapter 9

Future Work

This chapter considers possible topics for future extension and development of Kieker.

9.1 Backend

- **ARIMA and GARCH**
  Extend ΘPAD with a new forecasting algorithm based on ARIMA and GARCH introduced in Amin et al. [2012].

- **Business Transaction Detection**
  Business Transaction Detection tries to detect continuously occuring sequences of methods that may represent a business transaction. E. g., a login process.

- **Correlation of Measures**
  Correlation of measures tries to identify correlation patterns between two or more independet measures taken by Kieker. E. g., increased CPU-load correlates with increased anomaly scores.

- **Software Aging Detection**
  Software aging describes a progressive degredation of performance caused by errors, memory leaks, fragmentation or exhaustion of other resources. Kieker may be extended to detect symptomps of software aging like continuously increasing latencies. This would basically introduce a new anomaly detection algorithm.

- **Exception Monitoring**

  Exception occuring in the instrumented and observed application can not be recognized as such by Kieker. Exception could be registered by Kieker and offered to the GUI.

- **Experiments**
  The newly implemented experiment framework features collective, point and contextual anomalies. Not yet implemented are system-wide anomalies, or anomalies caused by a complete failure of individual components. Provide more complex input generation

patterns and pattern recognition within the output evaluation.

- **Online Configuration** Kiekeriki is currently not controlled by the GUI. Future work contains offering online configuration of filters and other functionality to the GUI.

- **ExtendedSelfTuningForecastingFilter - WCF**
  The newly introduced forecasting filter utilizes a decision tree which is optimized for accuracy. To further improve the filter it also needs to be optimized in performance to be less dependent on the current workload, of the system on which the forecasting filter is executed on, and to require less resources for the forecasting itself.

Markus Fischer

9.2 Frontend

- **Business Transactions**
  Add a new view to the Web GUI to see business transactions separately. This makes it possible to see one business transaction as a whole without the need to use the deep dive mechanism. A list of business transactions can give an overview, which transaction uses the most resources. In the search for problems in the system this could be essential.
- **Control the backend**
  Add the possibility to start and stop the monitoring of the Kiekeriki backend. In addition some parameters for the ΘPAD or RanCorr configuration could be transfered to the backend.
- **Project management**
  Add the possibility to work with different projects.
- **Dynamic Metrics and Diagrams**
  Add the possibility to make the metrics more dynamic and to show different metrics in one diagram. At the moment, everything related to the metrics and diagrams is statically programmed. A refactoring could make it possible to easily add new metrics without requiring to change the code.

During the evaluation, the participants had the chance to give feedback about the WebGUI. The feedback contains some ideas, which are worth adding to this section.

- **Refine Dialog to add Diagrams**
  The dialog for adding metrics should be refined. It is not optimally intuitive. A list for example would probably be more suitable than the "carousel"-selection. In addition it should be more clear for which components the diagram will show data for.
- **Improve Diagrams**

The diagrams could offer more possibilities. For example a fullscreen view of a diagram could be helpful to examine the graphs in detail.

- **Improve Architecture View**
  The architecture view could be improved by removing the flickering of the components. This is caused by the library we are using for the force directed layout (see [Spr]). For the future, another library could be used.
- **Modular Views**
  The participants wished that the view would be more modular. So for example the modules architecture view, diagram view and the events section at the bottom should be resizable and detachable so they can be reordered dynamically.

Martin Scholz

Bibliography

[Apa] Apache continuum website. https://continuum.apache.org/. URL https://continuum.apache.org/.

[CoC a] Cocome 2. http://sourceforge.net/apps/trac/cocome/, a. URL http://sourceforge.net/apps/trac/cocome/.

[CoC b] Cocome website. http://cocome.org/, b. URL http://cocome.org/.

[Git] Git website. http://git-scm.com/. URL http://git-scm.com/.

[Het a] Hetzner ex60. http://www.hetzner.de/en/hosting/produkte_rootserver/ex60, a. URL http://www.hetzner.de/en/hosting/produkte_rootserver/ex60/.

[Het b] Hetzner website. http://www.hetzner.de/en/, b. URL http://subversion.apache.org/.

[Hig] Highcharts js website. http://www.highcharts.com/. URL http://www.highcharts.com/.

[Jen] Jenkins website. http://jenkins-ci.org/. URL http://jenkins-ci.org/.

[Kie] Kieker website. http://kieker-monitoring.net/. URL http://kieker-monitoring.net/.

[Mav a] Checkstyle plugin website. https://maven.apache.org/plugins/maven-checkstyle-plugin/, a. URL https://maven.apache.org/plugins/maven-checkstyle-plugin/.

[Mav b] Findbugs plugin website. https://code.google.com/p/findbugs/, b. URL https://code.google.com/p/findbugs/.

[Mav c] Pmd plugin website. https://maven.apache.org/plugins/maven-pmd-plugin/, c. URL https://maven.apache.org/plugins/maven-pmd-plugin/.

[Mer] Mercurial website. http://mercurial.selenic.com/. URL http://mercurial.selenic.com/.

[RPr] R project website. http://www.r-project.org/. URL http://www.r-project.org/.

[Red] Redmine website. http://www.redmine.org/. URL http://www.redmine.org/.

[Sof] Sofa svn repository. svn://svn.forge.objectweb.org/svnroot/sofa/trunk/sofa-j/trunk/demos/sofashop/. URL svn://svn.forge.objectweb.org/svnroot/sofa/trunk/sofa-j/trunk/demos/sofashop/.

[Spr] Springy.js website. http://getspringy.com/. URL http://getspringy.com/.

[Sub] Subversion website. http://subversion.apache.org/. URL http://subversion.apache.org/.

[Tra] Trac website. http://trac.edgewall.org/. URL http://trac.edgewall.org/.

[ant] Apache ant website. http://ant.apache.org/. URL http://ant.apache.org/.

[bui] Apache buildr website. https://buildr.apache.org/. URL https://buildr.apache.org/.

[gra] Gradle website. http://www.gradle.org/. URL http://www.gradle.org/.

[mav] Apache maven website. http://maven.apache.org/. URL http://maven.apache.org/.

[mys] mysqltuner website. http://mysqltuner.pl. URL http://mysqltuner.pl/.

[Amin et al. 2012] A. Amin, A. Colman, and L. Grunske. An approach to forecasting qos attributes of web services based on arima and garch models. In *Web Services (ICWS), 2012 IEEE 19th International Conference on*, pages 74–81, June 2012. doi: 10.1109/ICWS.2012.37.

[Bielefeld 2012] T. C. Bielefeld. Online performance anomaly detection for large-scale software systems, Mar. 2012. Diploma Thesis, Kiel University.

[Boyer 2008a] B. Boyer. Robust java benchmarking, part 1: Issues, June 2008a. URL https://www.ibm.com/developerworks/java/library/j-benchmark1/.

[Boyer 2008b] B. Boyer. Robust java benchmarking, part 2: Statistics and solutions, June 2008b. URL https://www.ibm.com/developerworks/java/library/j-benchmark2/.

[Boyer 2008c] B. Boyer. Robust java benchmarking, supplements, June 2008c. URL http://www.ellipticgroup.com/html/benchmarkingArticle.html.

[Frotscher 2013] T. Frotscher. Architecture-based multivariate anomaly detection for software systems, Oct. 2013. Master's Thesis, Kiel University.

[Georges et al. 2007] A. Georges, D. Buytaert, and L. Eeckhout. Statistically rigorous java performance evaluation. In *Proceedings of the 22Nd Annual ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications*, OOPSLA '07, page 57–76, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. doi: 10.1145/1297027.1297033. URL http://doi.acm.org/10.1145/1297027.1297033.

[Hasselbring et al. 2013] W. Hasselbring, R. Heinrich, R. Jung, A. Metzger, K. Pohl, R. Reussner, and E. Schmieders. iobserve: Integrated observation and modeling techniques to support adaptation and evolution of software systems. 2013.

[Herbst 2012] N. R. Herbst. Workload classification and forecasting. Master's thesis, Karsruhe Institute of Technology, 2012.

[Herold et al. 2008] S. Herold, H. Klus, Y. Welsch, C. Deiters, A. Rausch, R. Reussner, K. Krogmann, H. Koziolek, R. Mirandola, B. Hummel, et al. Cocome-the common component modeling example. In *The Common Component Modeling Example*, pages 16–53. Springer, 2008.

[Jung et al. 2013] R. Jung, R. Heinrich, and E. Schmieders. Model-driven instrumentation with kieker and palladio to forecast dynamic applications. In *Symposium on Software Performance: Joint Kieker/Palladio Days 2013*, volume 1083, pages 99–108. CEUR, 2013.

[Marwede et al. 2009] N. S. Marwede, M. Rohr, A. van Hoorn, and W. Hasselbring. Automatic failure diagnosis in distributed large-scale software systems based on timing behavior anomaly correlation. In A. Winter, R. Ferenc, and J. Knodel, editors, *Proceedings of the 13th European Conference on Software Maintenance and Reengineering (CSMR'09)*, pages 47–57. IEEE, Mar. 2009. ISBN 978-0-7695-3589-0. doi: 10.1109/CSMR.2009.15.

[Project 2013] K. Project. Kieker user guide. Forschungsbericht, April 2013. URL `http://eprints.uni-kiel.de/16537/`.

[Salfner et al. 2010] F. Salfner, M. Lenk, and M. Malek. A survey of online failure prediction methods. *ACM Comput. Surv.*, 42(3):10:1–10:42, Mar. 2010. ISSN 0360-0300. doi: 10.1145/1670679.1670680. URL `http://doi.acm.org/10.1145/1670679.1670680`.

[Shields 2010] G. Shields. The definitive guide to application performance management. *International Journal on Advances in Software*, 2010. URL `https://www.gartner.com/doc/2639025/magic-quadrant-application-performance-monitoring`.

[van Hoorn] A. van Hoorn. Dissertation - unpublished.