Workload and Job Size Aware Performance Anomaly Detection

Bachelor's Thesis

Wolfgang Andreas Ramos Arhuis

March 29, 2017

Kiel University
Department of Computer Science
Software Engineering Group

Advised by: Prof. Dr. Wilhelm Hasselbring
Dipl. Inf. Armin Möbius

Eidesstattliche Erklärung

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

Kiel, 29. März 2017

Abstract

Varying server response times are often the result of varying workload and the varying computational demands of requests (*job size*). Although this link is well established, many monitoring approaches ignore it by relying exclusively on static response time thresholds for anomaly detection. We propose a combined approach where static response time thresholds are supplemented with techniques for detecting anomalies through comparing observed response times with predictions accounting for workload and job size. Empirical analysis of data collected from a real world route scheduling server application suggested, that workload and job size together can account for almost 50 % of the observed response time variability and can improve the prediction of respone times by almost 15 % for this application. We demonstrate how this method can be used in software operations by implementing a control center that helps operators to detect performance declines in this sofware.

Contents

List of Acronyms

AIC Akaike Information Criterion

ANOVA analysis of variance

ARIMA autoregressive integrated moving average

CRAN Comprehensive R Archive Network

CLR Common Language Runtime

CPU Central Processing Unit

CRM customer relationship management

CSS Cascading Style Sheets

GPL General Public License

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol

JDBC Java Database Connectivity

JIT just-in-time

JSON JavaScript Object Notation

JSR Java Specification Request

JVM Java Virtual Machine

MSE Mean Squared Error

MSSQL Microsoft SQL

OS Operating System

REST Representational State Transfer

RAM Random Access Memory

RSS Residual Sum of Squares

SOAP Simple Object Access Protocol

SQL Structured Query Language

UTC Coordinated Universal Time

Chapter 1

Introduction

Logistics, transportation, sales and on site service are growing business sectors in our modern economy. All of them are faced with a common challenge: the efficient allocation of mobile resources and workforce to jobs at different locations. The process of composing a set of routes from a set of geographically distributed jobs for a set of field service employees is called *route scheduling* or *route planning*. Good route scheduling can help minimize costs and waiting time, making it an important success factor for companies. Route scheduling software can make route scheduling more efficient, especially in complex scenarios with huge numbers of resources, employees, and jobs. The introduction of mobile devices into field operations and the extension of mobile networks brought new possibilities for automatic route scheduling. Mobile clients can be used to report on job process and fetch the most current schedules from a central server, constantly optimizing schedules. High availability and performance of the server are critical for the smooth operation of such a client-server setup. Thus creating the need to monitor route scheduling server applications for early signs of performance declines.

1.1 Motivation

The problem of finding ideal routes in advance for given sets of n jobs and m field staff employees is known as *vehicle routing problem* or *vehicle scheduling problem* [Lenstra and Kan 1981]. If field service employees do not share a common base location, the problem is also referred to as *l-depot vehicle scheduling problem*. In practice a composition of routes in advance is usually not sufficient. Changes to prearranged schedules often become necessary on short notice due to external influences, like traffic, sick leave, misjudged job duration and the emerging of new jobs. The problem of finding an ideal adaption of a schedule to such events is known as *vehicle rescheduling problem* [Li et al. 2007]. Like bad scheduling, bad rescheduling can lead to a waste of time and money. Thus, there is a need for efficient rescheduling and schedule optimization services available on short call to reduce those negative influences. The need for efficient route scheduling and rescheduling software is however countered by the complexity of the task as all three problems were proven to be NP-hard [Spliet et al. 2014; Lenstra and Kan 1981]. Thus, it can be considered unlikely that there exists an algorithm that can solve one of these problems in polynomial

time[1] [Sipser 2006]. To get an idea about how complex real-world route scheduling can be, consider that even a simple setting with only 10 jobs and one field service employee yields $10! = 3\,628\,800$ possible routes. Although in practice done heuristically, good route scheduling and rescheduling is still a computationally demanding task. Consequently, efficient route scheduling and rescheduling applications have high system requirements.

The increased use of mobile devices in field operations provides an ideal environment for client-server based route scheduling applications like FLS VISITOUR. Through a mobile client (FLS Mobile), field service employees can fetch the most current schedule from the server (VISITOUR Server) and report on job process. The server can use these updates on job process to perform real-time schedule optimization, a task to demanding for current mobile devices. The major advantage of centralized schedule handling is however the possibility to include all schedules of all field service employees in an optimization. So if one job falls out of schedule, or a new job has to be added, the server can utilize all field service employees to find a proper rescheduling. The optimized schedules can then be propagated to all field service employees almost in real time through the mobile devices. The overall usability of such a client-server setup depends heavily on the response time of the server. Smooth operation is only possible with short response times. Thus, high availability and performance of the route scheduling server need to be continuously ensured.

Monitoring and *performance anomaly detection* are two established techniques to detect early signs of performance declines and ensure high availability. Monitoring is the supervision of the operation of a system [IEEE Standards Board 1990]. Anomalies are patterns that do not conform to expected behavior [Chandola et al. 2009]. A popular indicator used to monitor the performance of server systems is the *response time* [Avritzer et al. 2006]. Rohr et al. [2010] define the *response time* of an operation as the time between the start and the end of the execution of the operation, including the time needed for executing nested suboperations. Anomalies in response times can be detected by comparing observed response times with statically set thresholds or expected response times (see Chandola et al. [2009] for an overview over techniques for anomaly detection). Expectations for response times can e. g., be generated using statistical models to predict response times based on past observations [see e. g., Henning 2016; Bielefeld 2012; Frotscher 2013].

Similar to the definition of the response time of an operation given by Rohr et al. [2010], the response time of a web service can be defined as the time between the completion of the reception of a request and the completion of the generation of the response. In the case of productively used systems, detecting anomalies in response times is complicated by varying *workload*. Menasce and Almeida [2001] define the workload of a system as the set of all inputs received from the environment. They differentiate between *workload intensity* and *individual request characteristics*. The number of concurrently processed requests is a typical measure of workload intensity in a server system. Individual request characteristics refer to the parameterization of a request that affect the computational complexity of processing the request. To avoid confusion between workload caused by workload intensity and workload

[1] This would only be possible if $NP = P$.

caused by individual request characteristics, we will refer to workload intensity as *workload* and to individual request characteristics as (request) *job size*. The term *job size* was chosen, because the individual request characteristics affect the duration of the processing of a request independent of the processing of other requests, like the size of a job does in a batch system.

Because of varying workload and job size, response times of productively used systerms often show high statistical variance, thus, complicating the detection of anomalies. Rohr et al. [2010] developed a method to reduce response time variance by accounting for workload. The authors grouped response times by defined workload classes. They were able to show that, the resulting average response time variance of the groups was lower than the globally computed response time variance. Thus, within each group response times were more stable, allowing better detection of performance anomalies.

In this thesis, we will develop a method to detect anomalies that accounts for varying workload and request job sizes in VISITOUR Server. Similarly to Rohr et al. [2010], we will use a workload measure to account for the effect of workload on response times. In addition, we will use request parameters to account for the effect of request job size on response times. We will investigate in how far the variability of VISITOUR's response times could be reduced by accounting for these parameters. Additionally, we will demonstrate how this method could be used in software operations by implementing a *control center* that utilizes this method to automatically detect performance anomalies in running VISITOUR Server instances.

1.2 Goals

The ultimate goal of this thesis is the implementation of a system that makes it possible to detect anomalies in the performance of VISITOUR Server. Therefore, we will first provide means to monitor performance, workload and request job sizes in a running server (G1). We will then use this monitoring setup to conduct an empirical analysis of the influence of workload and request job size on server response times in order to identify a model for response time prediction that accounts for workload and job size (G2). Finally, we will implement a control center that uses this method to support the detection of performance anomalies in VISITOUR Server (G3).

1.2.1 G1: Monitoring

We will implement a system to monitor and store one performance and several workload indicators for a single running VISITOUR Server. Monitored data is most useful when monitoring takes places in real-time, we will thus keep the monitoring delay as small as possible. We will target a monitoring solution that will allow future work to expand monitoring by adding additional performance indicators or increasing the number of simultaneously monitored server instances.

1.2.2 G2: Anomaly Detection

We will develop a prediction based method to detect anomalies in the response times of VISITOUR Server that accounts for workload and request job size. To identify such a prediction model tailored to predict application-specific response times, we will conduct an empirical examination of the influences of workload and job size parameters on VISITOUR's response times. To account for the influence of varying workload on server response times, we will adapt the approach of Rohr et al. [2010] to reduce the variability of response times through including workload measures into the prediction model. In addition, we will extend this approach, by including request job size measures in the prediction model to account for the influence of varying request job sizes on server response times.

1.2.3 G3: Control Center

We will implement a control center that visualizes changes in server performance, workload and request job sizes over time and utilizes the method envisioned in goal G2 to support the detection of performance anomalies in running VISITOUR Server instances.

1.3 Document Structure

We begin with an introduction on the foundations of anomaly detection, VISITOUR Server, software for processing log data, control centers for software operations, load generation for empirical assessment of server performance, and statistical data analysis in Chapter 2. Thereafter, in Chapter 3 we detail on the conducted empirical examination of the influences of workload and request job size on the response times of VISITOUR Server's appointment proposal service. In Chapter 4 we describe the implementation of the control center envisioned in goal G2. Next, we discuss the results from the empirical examination and the implementation of the control center in Chapter 5. We close with recommendations for future work in Chapter 6, and a comprehensive summary of our results in Chapter 7.

Chapter 2

Foundations

This chapter presents the foundations of response time analysis, anomaly detection, and related technologies. Section 2.1 details on approach used by Rohr et al. [2010] to analyze response times with respect to workload. Thereafter, Section 2.2 describes methods to detect anomalies in response time data. Next, Section 2.3 details on control centers for software operations. Section 2.4 describes FLS VISITOUR with a focus on the generation of appointment proposals. Because VISITOUR records content and time of all requests and responses in a local log file, we will extract response times and request parameterizations necessary for the envisioned response time prediction method from this file. Therefore, Section 2.5 presents software that can be used for real time extraction, processing and storage of log data. To generate response time data for the analysis envisioned in goal G2, we will use a load generator software. For that reason, Section 2.6 describes the scriptable load generator JMeter. Finally, Section 2.7 details on software for statistical analysis of response times and for building a control center that can make use of statistical models for anomaly detection.

2.1 Workload-Sensitive Response Time Analysis

Blocking a productive system to execute a standardized test procedure is often undesirable. Thus, monitoring data is the only source for detecting anomalies. However, analyzing monitoring data for performance anomalies is a difficult task, because productively running systems are usually faced with varying workload and varying request job sizes. A web service targeted at human users e. g., usually receives most requests during daytime and few at night. Thus, during phases with low workload, requests can be rapidly processed and can use full system resources, while during high-workload phases system resources are shared among requests. As a result monitored response times often show high statistical variance, making it difficult to detect anomalies [Rohr et al. 2010; Rohr 2015].

Rohr et al. [2010] developed a method to reduce the variance of response times by accounting for workload. The authors monitored response times of function calls together with the number of concurrently executed traces in the test system as workload indicator. Workload indicator values were used to define *workload classes* which in turn were used to group response time measurements. The authors showed that, the resulting average response time variance of the groups was lower than the globally computed response time

variance. Hence, within the groups, response times were more stable than when assessed jointly.

2.2 Performance Anomaly Detection

A popular approach to detect performance anomalies in productively used software systems uses (*statistical*) *regression models* to predict a certain performance *criterion* and then compare this prediction with an observation of the criterion [see e. g., Henning 2016; Bielefeld 2012; Frotscher 2013]. The predictions are generated from a set of predictors and a function, that defines how the criterion is computed from the predictors. This prediction function is the heart of the regression model, and is determined by (1) the model which constrains the function to a certain form, leaving some parameters to be deliberately chosen, and (2) a fitting of the model to empirical data to identify optimal values for these parameters.

Different regression models constrain the prediction function to different forms. *Linear regression* e. g., constrains the prediction function to linear functions, i. e., functions of the form $\hat{y} = a + \sum_{i=0}^{m} b_i x_i$ for $m \in \mathbb{N}$. In this definition, $\hat{y}$ is the prediction for the criterion y, and $x_0, \ldots, x_m$ are the predictors. In the case of the linear regression, the deliberately chosen parameters are the *coefficients* a, and $b_0, \ldots, b_m$. a is also called *intercept*. Optimal coefficient values for a set of $n \in \mathbb{N}$ observations are identified using the *method of least squares*. This method minimizes the *residual sum of squares* (RSS), which is the sum of all squared differences between the observed and the predicted criterion values, i. e., $RSS = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$. Each observation is thereby an $(m+1)$-tuple, consisting of an observed y, and m related observed predictor values. The RSS is thus a measure of the precision of the prediction of a linear regression model. A model with a smaller RSS generates predictions that are closer to the observed values, than a model with a larger sum of squares. Note, that fitting a model with no predictors to n observations yields $a = M(y)$. Hence, the prediction is computed as $\hat{y}_i = M(y)$ for all observations $1 \leqslant i \leqslant n$. Therefore, if no predictors are specified, the RSS is essentially equivalent to the variance of the criterion computed as $\mathrm{var}(y) = \sum_{i=0}^{n} (y_i - M(y))^2$.

A measure of the average deviation of the prediction from the observation is the *mean squared error* (MSE). It is computed as $\frac{1}{df_e} RSS$, where df_e are the *degrees of freedom* of the RSS. Degrees of freedom express the dimensionality of the prediction data. As a rule of thumb, df_e is the number of observations minus the number of predictors.

Another measure of the quality of a regression model is the Akaike Information Criterion (AIC). This measure expresses how economical a model is. Because the inclusion of predictors can never decrease the precision of the prediction (in the worst case a predictor does not change the prediction), models that generate good predictions with a low number of predictors are more economical than models that only generate slightly better predictions but use much more predictors. Essentially, the AIC expresses a ratio of the number of predictors included in the model and the model's RSS. Hence, models with low AIC values are preferable to models with high AIC values, even if the latter can yield

better predictions.

Regression models are not constrained to linear functions. Beside linear regression, there is e. g., *polynomial regression*, that fits a polynom, and *exponential regression* that fits an exponential function to sample data, and autoregressive integrated moving average (ARIMA) models that fit a function to periodic patterns of time series. The quality of all of these models can be expressed by the same parameters used to express the quality of a linear regression model, namely the RSS, the MSE and the AIC.

ARIMA models are popular in anomaly detection approaches [see e. g., Henning 2016; Bielefeld 2012; Frotscher 2013] , because they are especially designed to predict data that displays a certain periodicity. Shumway and Stoffer [2011] e. g., mention human voice, Electroencephalography, and stock data as application examples. ARIMA models are used in performance anomaly detection, because response times of productively running systems, often also show periodic patterns over the time of a day, a week, and a year. E. g., web applications targeted at human users, like social, media, and e-commerce often receive most requests during the evening and fewer requests during the day time, when people are at work or at school. However, if the data displays no periodic patterns, ARIMA model based predictions, are essentially equal to regression models that do not account for periodic patterns, like e. g., linear regression [Shumway and Stoffer 2011]. In cases, where the periodic pattern in the sample are artificial or coincidental, careless use of ARIMA models for prediction can even result in pathological prediction models, fitted to fake periodicity. Hence, Shumway and Stoffer [2011] recommend a close investigation of the periodic patterns in the data, before fitting an ARIMA model to the data.

A problem of using regression models for anomaly detection is, that fitting a complex model to a large data set can be a computational very demanding task. Thus, fitting such models to a continuously growing set of monitoring data will result in a gradual increase of the time needed for detecting an anomaly. In the implementations presented by Henning [2016], Bielefeld [2012], and Frotscher [2013], this is prevented, by the use of a *sliding window* for prediction. Instead of fitting the prediction model to the entire data, the model is continuously fitted to the $n \in \mathbb{N}$ last observations, or the observations from the last $m \in \mathbb{N}$ seconds, minutes, etc. Thus, the computational demand for detecting performance anomalies remains constant, even when the amount of available data increases. The rationale behind the use of sliding windows is, that observations from the near past are related more closely to the current system performance, than observations from the far past [Henning 2016]. Although this might hold true for some applications, prediction models are usually more robust, when fitted to many observations. Hence, the use of sliding windows can decrease the precision of the prediction, and increases the risk of modeling abnormal behavior.

Generation of biased models is a general problem that arises, when empirically identified prediction models are used for anomaly detection [Chandola et al. 2009]. The data to which the model is fitted might contain abnomal behaviour, thereby impairing the model's usability for identifying subsequent anomalies. The degree to which sliding window based

prediction models are biased depends on how much of the data from the sliding window results from abnomal behavior. In the worst case, anomalies slowly increase response times, gradually fitting models to more and more abnormal behavior. To avoid such pathological fitting processes, any technique that identifies anomalies based on predictions generated from past data must be supplemented with static thresholds for anomaly detection.

2.3 Control Centers for Software Operation

Control centers can be used for monitoring software systems. Giesecke et al. [2006] distinguish between control centers for software operation and control centers for software development. Control centers for software operation should reduce the time needed for detecting and resolving issues in a running application, thus increasing the availability of the service. Giesecke et al. [2006] emphasizes that software operation control centers are targeted at human operators. Hence, a good control center is one that visualizes the collected data in a way that supports issue detection and resolving.

2.4 FLS VISITOUR

FLS VISITOUR is a route scheduling software with a client-server architecture written in C# an executed in Microsoft's Common Language Runtime (CLR). FLS VISITOUR Server provides means for managing field service employees and jobs (*calls*) and performs the computationally demanding task of route scheduling and optimization. It uses a Microsoft SQL (MSSQL) database server to store information about employees, jobs and the current schedule. The services of VISITOUR Server are accessible through the Simple Object Access Protocol (SOAP) which is a protocol for remote procedure calls and the exchange of data.

One of the most important services provided by VISITOUR Server is the generation of appointment proposals for single jobs. Understanding how VISITOUR generates appointment proposals, requires an understanding of how VISITOUR models jobs, tours and employees. Therefore, Section 2.4.1 describes this model. Thereafter, Section 2.4.2 to Section 2.4.6 detail on different aspects of the generation of appointment proposals, and Section 2.4.7 summarizes these aspects into a comprehensive overview over the factors that influence the reponse time of VISITOUR' appointment proposal service. Requests for appointment proposals can be issued through VISITOUR's `Call` or `CallProposal` web service. Therefore, we subsequently detail on selected web services provided by VISITOUR, namely:

1. the `Call`, `CallProposal`, and `Calls` web services for handling jobs and generating or confirming appointment proposals (see Section 2.4.8)
2. the `Optimize` web service for generating and optimizing schedules for specified time windows (see Section 2.4.9),

3. the `DeletePlanning` web service for deleting the current schedule within a specified time window (see Section 2.4.10),
4. the `FieldManager` web service for creating/modifying database entries for field service employees (Section 2.4.11), and
5. the `RandomAddress` web service for generating random postal addresses (Section 2.4.12).

2.4.1 Route Scheduling Model

VISITOUR is adapted to real world route scheduling demands. Employees are assigned day-bound work times, overtimes, breaks, and holidays. Jobs are assigned a *state*, a duration and possibly timing constraints, like being tied to opening hours. VISITOUR uses states to model the life cycle of real world jobs. Jobs are created with state *new*. When an appointment is confirmed, the corresponding job receives state *confirmed*. When an employee departs for a job, the job receives state *on route*. During execution, the job's state is *in progress*. After a job has been served, it receives the state *finished*. When a previously confirmed appointment is canceled, the corresponding job is assigned the state *escalated*. Figure 2.1 shows VISITOUR's job states model as finite state machine with example transitions using the `Call` (see Section 2.4.8), and `DeletePlanning` (see Section 2.4.10) web service, and SQL update requests to the job database.

On every workday each available employee is assigned one *tour*. Jobs are scheduled by including them in tours. Tours consist of sequences of jobs and interim travel times.

2.4.2 Generation of Appointment Proposals

To generate an appointment proposal VISITOUR iteratively inserts the target job into different tours at different positions. Inserting a new job into a tour can result in a *domino effect*. To illustrated this, consider what would happen when a new job is added to a tour that already fills the employee's entire workday. In order to serve the new job, one or more previously confirmed jobs would have to be removed from that tour. Instead of canceling these jobs, VISITOUR tries to include them in other tours, which in turn can lead to more jobs dropping out of their respective tours.

The process of generating appointment proposals for a target job can roughly be divided into two phases:

1. Identifying *candidate positions*, where the target job could be inserted into the current schedule, i. e., the currently defined tours.
2. Comparing different route compositions with regard to a *cost function*.

Because real world route scheduling scenarios usually result in many possible route compositions, VISITOUR uses parameterizable heuristics to limit the computational effort. The parameterization allows customers to adapt VISITOUR's route scheduling routines to their needs, emphasizing speed or quality of optimization processes. In consequence, the

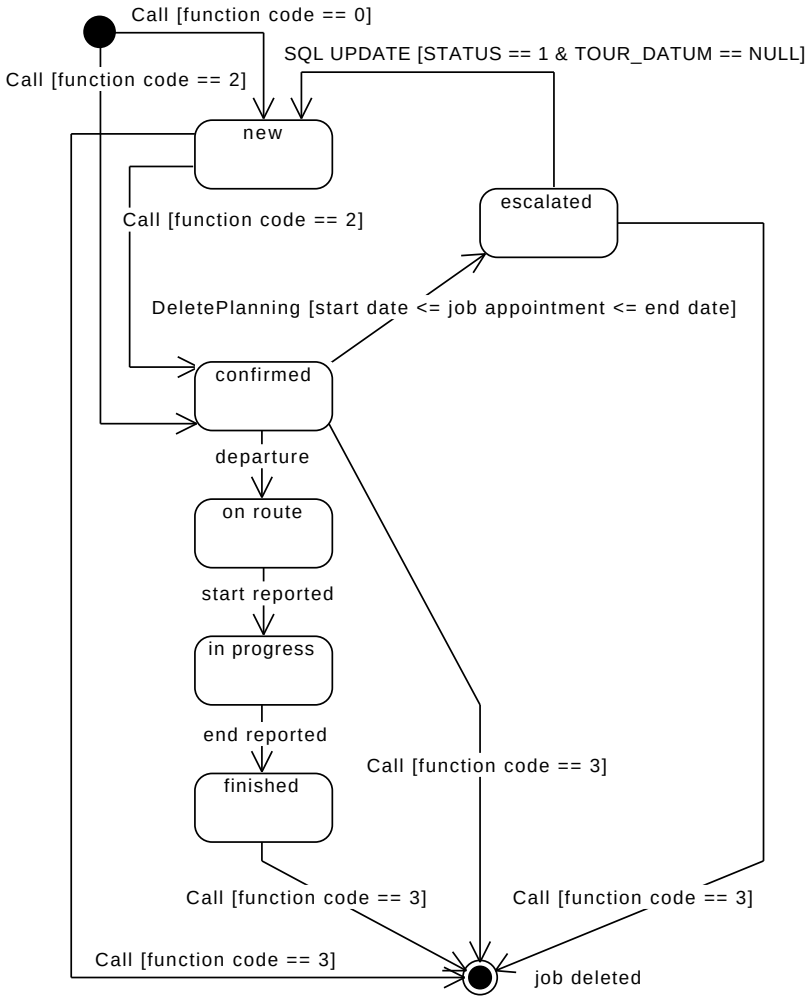

Figure 2.1. VISITOUR's job states. Example transitions are shown between job states. Arbitrary transitions are possible through updating job database entries.

computational demands of an appointment proposal depends heavily on the parameterization of these heuristics.

The heuristics offered by VISITOUR can be roughly separated into two groups, based on the phase during which they are applied. Phase one heuristics reduces the size of the problem by constraining the investigation to a subset of route compositions. The most popular phase one heuristics available in every VISITOUR Server installation identify candidate positions through distance thresholds. These heuristics are described in Section 2.4.3. Other heuristics in this group are available as payed extensions to the default installation. They

allow e. g., the identification of suitable positions for jobs based on skill requirements and customer base.

Phase two heuristics reduce the duration of the investigation. The most commonly used heuristic in this group works by setting a threshold for the number of route compositions to investigate. This heuristic is described in Section 2.4.5. Another phase two heuristic works by setting a threshold for the duration of the investigation in time units. This heuristic is however not available for generating appointment proposals for single jobs. It can only be applied to large scale scheduling operations, as offered by the `Optimize` service (Section 2.4.9).

2.4.3 Candidate Positions

In every VISITOUR Server installation the number of candidate positions can be limited by two parameters: the *optimization radius* and a threshold for employees resp. tours to include in the optimization (*tours threshold*). Both parameter are part of the server configuration and cannot be set as part of a `Call` or `CallProposal` request (see Section 2.4.8).

The optimization radius specifies a maximum distance in kilometers for including field service employees in optimizations. If an employee's base location is not further away than the optimization radius the employee's tour is included in the optimization, otherwise not. Note, that including a tour in an optimization leads to the inclusion of all jobs on this tour in the optimization. Thus, by default all jobs on this tour become subject to optimization.

The tours threshold utilizes the fact, that VISITOUR adds employees to optimizations nearest to furthers, i. e., the tour of the nearest employee is included first, then the tour of the second nearest, and so on. If the number of included tours in the optimization reaches the tours threshold, VISITOUR will stop including more tours and move on to comparing route compositions (phase two).

To clarify the relationship between these two parameters, consider a setup with tours threshold set to 30 and two jobs j_0, and j_1, where j_0 has 30 employees based within the optimization radius, and j_1 has 50. A request for an appointment proposal for job j_0 would include the tours of all 30 employees in the optimization. A request for an appointment proposal for j_1 on the other hand would include only the tours of the 30 nearest employees in the optimization, discarding the other 20.

Additionally, VISITOUR uses three employee-specific parameters that determine which employees are actually able to serve a target job and which only take part in the optimization to permit domino effects: the *deployment radius*, and the *tour length threshold*. The deployment radius specifies the maximum end to end distance in kilometers field service employees are allowed to travel from their base location to a target job. The tour length threshold specifies the maximum total distance an employee is allowed to travel on his tour. The total traveling distance is computed as the sum of all distances an employee has to travel on his tour, i. e., the distance between his base location and the first job on his tour, the distances between the subsequent jobs, and the distance between the final job on his tour and his base location. Both, the deployment radius and the tour length threshold can be set

for each employee individually or through setting server default values assigned to newly created employees.

To illustrate the interplay of all four parameters, consider an example setup where tours threshold is set to 30, optimization radius is to 300 km, and deployment radius to 100 km. Let's assume in this setting we have a job j_0, that has 20 employees based within 100 km, additional 10 employees within 200 km, and another 10 employees within 300 km. If an appointment proposal is requested for j_0, all 20 employees based within 100 km would be considered candidates for serving j_0. The tours of the 10 employees based within 200 km would be included in the optimization, but would not be considered candidates for the target job, only for jobs dropping out of other tours (provided the distance to these jobs is not greater than the employee's deployment radius). Finally, the tours of the 10 employees within 300 km would not be included in the optimization at all, because the total number of tours in an optimization is limited by the tours threshold.

To provide the distances between job locations and job and employee locations necessary for these heuristics, VISITOUR maintains a distance matrix (see Section 2.4.4). If a distance required by an optimization is not found in the distance matrix, it is automatically computed. Thus, distance calculations affect the response time of VISITOUR's appointment proposal service.

In conclusion, the most important factors influencing the job size of the optimization performed in the course of an appointment proposal request are:

- the number of included tours,
- the number of identified candidate positions,
- and the number of included jobs

The more tours are included in an optimization, the higher is the number of candidate positions. A high number of candidate positions leads to the inclusion of many possible route compositions in the investigation. Likewise, a high number of included jobs increases the complexity of the optimization, by increasing the number of route compositions. If only empty tours were included in an optimization, finding the best appointment proposal would be simple: just select the employee whose base location is closest to the job's location. The more jobs are included, the more possibilities arise to shift jobs between tours, thus, increasing the response time of an appointment proposal request.

2.4.4 Distance Calculation

To generate appointment proposals, VISITOUR Server maintains a *distance matrix*, storing distances between job/employee base locations. This distance matrix is extended whenever a job is included in an optimization for the first time, e. g., when the first appointment proposal is requested for a job. Thus, the first appointment proposal request is computationally more expensive than every following proposal requests targeting the same job.

The *standard distance matrix* used by VISITOUR is an $n \times n$ *square matrix*. This matrix stores

all pairwise distances. The default size of a standard distance matrix is 8000×8000. The maximum size is limited by the maximum size of arrays in C#. In a 64 bit environment, an array can contain up to $4 \cdot 10^9$ elements[1]. Thus, the maximum size of a standard distance matrix is 63245×63245, because $\sqrt{4 \cdot 10^9} \approx 63\,245.55$.

A popular optimization that reduces memory usage, is the use of a *rectangular* $n \times m$ matrix, where $m < n$. For each job this matrix stores only the distances to the m nearest job/employee base locations. This optimization is especially useful, when a huge number of locations must be handled.

Distance and travel time calculations are based on road map data licensed from *TomTom*. This data includes information on the geographical localization of roads, road lengths and speed limits. End to end travel times are composed from travel times for single road sections which in turn are derived from section lengths and speed limits.

2.4.5 Proposal Iterations

The *proposal iteration heuristic* uses the iterative character of phase two to limit the computational effort for generating appointment proposals. In each iteration a possible route composition is chosen and it's *cost value* is computed which is used in a final step to compare different schedulings and choose one with minimal costs. VISITOUR allows users to set a threshold to limit the number of performed *optimization iterations*. When this threshold is reached, VISITOUR stops investigating further route compositions and returns the position belonging to the route composition with the lowest cost value as appointment proposal for the target job. The threshold is part of the server configuration and cannot be specified individually for every `Call` or `CallProposal` request.

2.4.6 Proposal Windows

In real world route scheduling it is often useful to generate multiple appointment proposals at once, e. g., in order to leave the final choice to the customer or the human dispatcher. VISITOUR supports this through the use of *proposal windows*. A proposal window is a continuous time period during *one* day for which an appointment proposal is requested. If multiple appointment proposals should be generated, a job can have multiple proposal windows assigned. Consider a job j_0 that can be served from 08:00 to 18:00 on one day and a job j_1 that can be served from 08:00 to 18:00 on two consecutive days. j_0 could be assigned one proposal window, whereas j_1 could be assigned two proposal windows. In the first case, VISITOUR would generate one appointment proposal and in the second case two appointment proposals. Most importantly, in the first case VISITOUR would only perform one optimization, whereas in the second case VISITOUR would perform two optimizations, one for each service window.

[1] Source: `https://msdn.microsoft.com/en-us/library/System.Array(v=vs.110).aspx`

2.4.7 Response Time Influences

As described in Section 2.4.3 through Section 2.4.5, the most important factors influencing the size of the computation triggered by an appointment proposal request are:

- the number of tours that are included in the optimization,
- the number of jobs that are included in the optimization,
- the number of candidate positions,
- the number of requested optimization iterations, and
- distance calculations

By influencing the computation, these parameters also influence the response time of appointment proposal requests. Most importantly, these parameters influence the response time of requests *independently* of other concurrently processed requests. To illustrate this, imagine a hypothetical situation where VISITOUR processes requests one after another. In this scenario, the response time of a request would depend exclusively on the identified *job size parameters*, and would not be affected by other requests.

In practice however, requests do not arrive one after another, and the generation of appointment proposals is partially performed concurrently, and sequentially. For every incoming `Call` resp. `CallProposal` request VISITOUR spawns a new thread that extracts the parameters for the proposal generation from the SOAP request's payload. The identification of candidate positions (phase one), and the iterative computation of cost values for different route compositions (phase two) are however sequentialized through a mutual exclusion condition, allowing at most one thread at a time. Consequently, when a thread executes phase one or two, the execution of both phase by other appointment proposal requests is stalled. The subsequent generation of the SOAP response is again performed concurrently.

Because requests are acceppted concurrently, and afterwards processed sequentially, they can affect each other's response time. Therefore, the current workload is another factor, that influences the response time of VISITOUR's appointment proposal service.

2.4.8 `Call`, `CallProposal` and `Calls` Web Sevices

The `Call` service is a multipurpose service for handling jobs. It allows users to create/modify single jobs, to generate, and confirm appointments, and to delete jobs. Request to this service specify i.a:

- a *function code*,
- a *job ID* to identify a job in the database,
- a location,
- start and end of the time interval during which the job should be scheduled, and
- a duration

Table 2.1. Mapping of `Call` request function codes to actions

function code	action
0	create or update the target job,
1	generate an appointment proposal
2	confirm an appointment
3	delete a job from the database
4	delete a job only if it's status is less than *on route* (see Section 2.4.1)
5	cancel a job's currently confirmed appointment

The function code, determines the action VISITOUR performs. Table 2.1 shows the mapping of function codes to actions.

The generation of appointment proposals and the confirmation of an appointment is possible without prior creation of the job. If no job is specified in the corresponding requests, a new database entry for the job is created. The confirmation of an appointment proposal is even possible without specifying a date and time for the appointment. In this case VISITOUR will automatically generate appointment proposals for the job and choose one with the lowest cost value.

Jobs in the database are identified by job IDs. On creation every job is assigned a unique *internal job ID*. Additionally, users can specify a unique *external job ID* for each job. These external IDs can e. g., be useful when calling VISITOUR services from third party customer relationship management (CRM) systems. To modify an existing job, `Call` request must specify the internal or external ID the job to modify together with the job parameters that should be updated in the database.

The response to a `Call` request contains a *return code* to indicate if the requested operation was successful, or what error occurred. An overview over the possible return codes for appointment proposal requests (`Call` requests using function code 1) can be found in Table 2.2.

The `CallProposal` service is semantically similar to the `Call` service used with function code 1, in that both services trigger the same actions in the server. I. e., issuing a `CallProposal` request for a job has the same effect as issuing a `Call` request with function code 1 for that job. However, requests and responses of both services differ with regard to the information, they contain. `CallProposal` request do not supply a function code, and the responses do not contain a return code. In case of failure, `CallProposal` responses return a text message informing the caller, that the server was unable to find a suitable appointment proposal.

The `Calls` service offers bulk job creation and modification. A `Calls` request is equivalent to multiple `Call` requests using function code 0. For each job users can specify the same parameters as in the corresponding `Call` request. Concerning the assignment and use of job IDs the rules described for the `Call` service also apply to `Calls` service requests.

2.4.9 `Optimize` Web Service

With the `Optimize` web service users can trigger the creation or optimize of the schedule for a specified time interval (*optimization window*). A request to this service must specify the start and the end of the optimization window. Furthermore, users can specify which unscheduled calls to include in the optimization in addition to the already scheduled jobs. Users can e. g., choose to include escalated jobs or jobs that are currently destined to be served, before or after the optimization window.

`Optimize` is often used to perform large scale prospective scheduling, like creating the schedule for the next week or next month. In contrast, the `Call` resp. `CallProposal` service is used to perform short-term (re-)scheduling of single jobs. Therefore, VISITOUR uses a separate threshold to limit the optimization iterations in optimization processes triggered by `Optimize`.

As detailed in Section 2.4.2 many of VISITOUR's route scheduling heuristics are based on distances. Therefore, VISITOUR first computes all missing pairwise distances for all employees and all jobs in the optimization window, when an `Optimize` request is received. Thus, the `Optimize` web service can be used to efficiently populate VISITOUR's distance matrix.

2.4.10 `DeletePlanning` Web Service

The `DeletePlanning` service allows users to cancel all currently confirmed appointments falling into a specified time interval at once. A request to this service must specify a start and an end point for the time interval used for canceling appointments. It is important to emphasize, that `DeletePlanning` does not delete job database entries or the distance matrix. Thus, `DeletePlanning` can be used to clear a schedule without loosing calculated end to end distances.

In real world route scheduling canceling appointments usually requires actions on the

Table 2.2. `Call` response return codes for appointment proposal requests (`Call` requests using function code 1)

return code	description
0	appointment proposal succefully generated
1	appointment proposal succefully generated, job location geocoded by city center (see Section 2.4.12)
2	job location not geocodable (see Section 2.4.12)
3	error in the specification of the time interval for the appointment
10	no valid appointment proposal found
30	job ID not found
-99	other error

side of the human dispatcher. Clients or field service employees have to be informed about the cancellation, and a new appointment has to be found. Therefore, VISITOUR uses the special job state *escalated* (see Section 2.4.1) to mark canceled jobs. By default, rescheduling escalated jobs can only be done by human dispatchers. In consequence, escalated jobs are by default not included in automatic (re-)scheduling actions, like `Optimize` requests (see Section 2.4.9). To re-include escalated jobs in automatic scheduling actions, the job state has to be changed back to *new*. In the case of the `Optimize` service the inclusion of escalated jobs is controlled through a parameter (see Section 2.4.9).

2.4.11 `FieldManager` Web Service

Adding or updating data base entries for field service employees is possible through the `FieldManager` service. Request to this service must specify at least:

- a surname and a forename,
- a start base location, and
- an end base location.

Start and end base location can be given as postal address or as coordinates. To modify an existing database entry the `FieldManager` request must specify the field service employee's internal or external ID. Internal IDs are assigned by VISITOUR. External IDs can be specified to map foreign IDs to field service employees. This is useful e. g., when calling VISITOUR services from third party CRM systems.

2.4.12 `RandomAddress` Web Service

For testing purposes it is often useful to automatically generate postal addresses to be used as job or employee base locations. This is possible through VISITOUR's `RandomAddress` web service. Requests to this service can specify:

- the number of addresses to generate,
- the random seed to use,
- the country and postal codes from which to draw postal addresses

The `RandomAddress` service returns only postal addresses for which *house number level precise geocoding* is possible based on the currently available road network data. Geocoding is the process of transforming a postal address to a geographic coordinate system, usually latitude and longitude. Because using postal addresses is prone to error (addresses could be typed incorrectly, or the road network data could be outdated), geocoding results are assigned a *precision level*. These precision levels are the result of several fall-back strategies VISITOUR uses to map a postal address to a latitude and a longitude. If the house number is not found, coordinates for the street center are returned (*street level precise geocoding*). If the

street is not found, coordinates for the city center are returned (*city level precise geocoding*), and if the city is not found, no coordinates are returned.

2.5 Log Analysis with Elastic Stack

The *Elastic Stack* (formerly ELK Stack) is formed by the open-source applications *Elasticsearch*, *Logstash* and *Kibana* that are released under the Apache License 2.0. *Elasticsearch* [Elastic 2016b] is a search and storage platform, making log data rapidly searchable. *Logstash* [Elastic 2016d] is a log collector and parser, bringing together logs from different sources. *Kibana* [Elastic 2016c] is a web front end for visualizing and querying logs stored in Elasticsearch. Supplemented with applications for shipping logs from remote nodes to a central log processor and storage, these tools form a system for centralized log management and analysis. Because Kibana does not provide features for using complex statistical models for response time prediction, we refrained from implementing the envisioned control center (see Section 1.2.2) as Kibana plugin. Instead, we chose to implement the control center using a programming language especially targeted at statistical data analysis (see Section 2.7).

2.5.1 Log Forwarding with the Beats Platform

An application that forwards locally generated log data to a central log processing system is called a *log forwarder* or *log shipper* [Kühnel 2013; Churilin 2013]. The *Beats Platform* [Elastic 2016a] is an open source framework developed by Elastic for creating log shippers that send logs to Logstash or directly to Elasticsearch. Log shippers developed with the Beats Platform are termed *Beats*. Elastic supplies Beats for shipping

- information extracted from network packets (*Packetbeat*),
- log files (*Filebeat*),
- metrics from the operating system and running services (*Metricbeat*),
- Windows event logs (*Winlogbeat*).

Filebeat replaces *logstash-forwarder* as log shipper for Logstash[2].

2.5.2 Log Management with Logstash

Logstash [Elastic 2016d] is an application that can collect logs from different sources and parse them into a desired target format (*log normalization*), using a pipes-and-filter pattern. Logstash thus acts as a preliminary stage for log storage and analysis. Input plugins handle the collection of logs from different sources. Logstash can e.g. read log data from sources like local log files, databases, syslog messages and log shippers build with the Beats Platform. In combination with log shippers, Logstash can thus be setup as a central log

[2]Source: `https://www.elastic.co/guide/en/beats/filebeat/current/migrating-from-logstash-forwarder.html`

processor, gathering logs from several nodes. Output plugins handle the use of different back ends for storing log data, including Elasticsearch. Filter plugins like *grok* handle the transformation of log data into a desired target format. Grok is the recommended filter plugin in for parsing unstructured logs into structured logs using regular expressions. Logstash also supplies a web front end for querying log data stored in Elasticsearch, but it provides only limited features for log data analysis. Therefore, Kühnel [2013], and Churilin [2013] recommend using Logstash only for pre-processing log data.

2.5.3 Search Platform Elasticsearch

Elasticsearch [Elastic 2016b] is a *document*-based, distributed full-text search and storage platform based upon the *Apache Lucene* [Apache Software Foundation 2016b] text search library. A document is a unit of information that can be indexed for searching. Documents are grouped in collections, termed *indices*. Indices can contain documents of different *type*. Types define the structure of the information contained in documents, i. e., the available fields and their data types. Documents are received, stored, and served in JavaScript Object Notation (JSON) format. New documents can be submitted via HTTP `PUT` requests. The documents stored in an index are made rapidly searchable through the creation of an *inverted index*. An inverted index maps text terms to locations in Elasticsearch's document stock. Thus, an inverted index can be used like a book's index to quickly jump to positions of interest. Searching can be done using the *Apache Lucene Query Syntax* or the JSON-based *Elasticsearch Query Language*. Elasticsearch is build for high performance, availability and horizontal scalability. High performance is provided through the inverted index, allowing near real time searching. Scalability and availability are provided through *sharding* and *replicas*. Sharding allows to split an index into several disjunct pieces (*shards*). A shard can be thought of as a fully functional index of its own. Sharding is used to scatter an index over an Elasticsearch *cluster*, formed by a group of Elasticsearch nodes. This allows for parallel of operations and load distribution. For optimal parallelization the number of nodes in a cluster should match the number of shards. Within a cluster it is possible to set up replications of shards, termed *replicas*. Replicas are nodes that contain copies of shards that can replace the original shard in case of a node failure.

2.6 JMeter

JMeter [Apache Software Foundation 2016a] is an open source load generator written in Java and released under the Apache License. JMeter is targeted at testing web applications using *test plans*. Test plans consist of test actions (*samplers*) nested in *thread groups* and *controllers*. *Thread groups* and *controllers* define the control flow of a test plan. Thread groups define sequences of a test actions that can be executed repeatedly concurrently by an arbitrary number of threads. Controllers can be nested in thread groups for more fine-grained regulation of control flow and as additional structuring elements. At a first glance

controllers and thread groups appear very similar, because both consist of test actions and both are used to manage the control flow. However, they differ in two important aspects:

1. Controllers cannot be used to define sections of concurrent execution. Test actions inside a controller are always executed by a single thread. Concurrent execution of controller actions is achieved by nesting controllers in thread groups.
2. Controllers can be nested inside each other, while thread groups cannot.

JMeter supplies e. g., controllers to loop over test actions (*loop controller*, *while controller*), execute or skip test actions based on a condition (*if controller*), nest the execution of a thread groups actions within another thread group (*module controller*), or simply to structure test plans (*simple controller*).

Samplers are the most basic units of a test plan. Samplers test single web service end points using different network protocols or execute scripts written in one of several supported programming languages. Script samplers can be used e. g., to automatically perform actions on service responses or to perform preparatory actions for requests, like loading data from files. JMeter supplies among others samplers to:

- issue HTTP requests (*HTTP request sampler*)
- interact with databases using Java Database Connectivity (*JDBC request sampler*),
- execute *Apache Groovy* [Apache Software Foundation 2017] scripts (*JSR 223 sampler*).

Java Database Connectivity (JDBC) is an interface for accessing relational databases from Java programs. A JDBC driver to access MSSQL databases is supplied by Microsoft [Microsoft Corporation 2017]. Apache Groovy is an object-oriented programming language that is dynamically compiled to Java Virtual Machine (JVM) bytecode. Groovy scripts can use Java libraries, thus giving access to the huge library infrastructure available for Java.

In addition to test actions, JMeter test plans can contain *timers*. Timers can be used to delay test actions by a fixed (*constant timer*) or random time interval (e. g., *uniform random timer*).

JMeter can be executed in a *GUI mode* and a *command line mode*. The GUI mode is recommended for building and debugging test plans. The command line mode is recommended for executing tests, because of it's smaller overhead concerning memory and CPU usage.

Because SOAP uses the HTTP protocol, JMeter is suitable for load testing VISITOUR's web services. The JDBC request sampler allows JMeter to access VISITOUR's database. In addition, complex setup procedures to prepare a VISITOUR server for testing, can benefit from JMeter's scripting features.

2.7 R

R [R Foundation 2016] is an open source interpreted programming language targeted at statistical data analysis and released under the General Public License (GPL). It supplies functions to perform statistical regressions, analysis of variance (ANOVA), and time series

analysis using ARIMA models (see Section 2.2). R is essentially a functional programming language, but also supports other programming paradigms. R programs are executed by R's command line interpreter. R's core functionality can be extended by user created *packages*, available e. g., through the Comprehensive R Archive Network (CRAN). CRAN supplies among others the packages:

- *shiny* for creating web applications in R,
- *elastic* to access Elasticsearch from R, and
- *methods* to support an object-oriented programming style in R (part of R's core since version 1.4)

The shiny package [RStudio Inc. 2016] provides a web framework for R. Shiny is targeted at building web applications (*shiny apps*) that make use of R's features for statistical data analysis. Shiny apps can be hosted using *Shiny Server*. For debugging and local usage, shiny apps can also be run from an R terminal. Shiny can make use of the CSS framework *bootstrap* [Twitter Inc. 2016] for creating *responsive websites*. Responsive websites automatically adapt the display of their content to the display width of the device accessing the website. Thus, shiny can be used to create websites that are meant to be accessed from both, mobile and desktop devices. Shiny supplies three components to support the development of web applications that react quickly to events, like user inputs: *reactive sources*, *reactive conductors*, and *reactive end points*. Reactive sources are values that generate events when they are changed. Changes can come from user inputs through HTML input elements (text, radio buttons, etc.) or automatic actions, like timed execution of code. Reactive conductors and end points are functions used to handle events. They are linked to reactive sources by the observer pattern. I. e., they are called when one of their observed sources is changed. Reactive conductors and end points differ with regard to propagating events. Reactive conductors can them self be observed by other reactive conductors and end points. I. e., if the value of a reactive conductor changes, all of its observers are called. Thus, reactive conductors can be used to generate chains of events. Reactive end points on the other hand do not propagate events. They are used for their side effects, like rendering website content or persisting data. Planful usage of these reactive components allows users to build websites that are (re-)generated quickly, constraining (re-)computation of the display to the necessary minimum.

The elastic package makes use of Elasticsearch's Representational State Transfer (REST) interface. It provides e. g., wrapper functions to list all indices of an Elasticsearch instance and to query an Elasticsearch index using the Apache Lucene or the Elasticsearch query language (see Section 2.5.3).

The methods package allows developers to use an object-oriented programming style for software development in R. The package defines a type `ReferenceClass` that is suitable to model objects with complex states [Wickham 2014] by reducing the overhead of passing arguments to functions. In R, function calls can be costly, when passing large data as arguments. The reason for this is that R uses call-by-value [see R Core Team

2016] for passing arguments to functions and (in most cases) copies the arguments to the function's environment. Using the `ReferenceClass` objects can reduce this overhead, by giving functions access to data without the need to pass this data as arguments. This is accomplished by implementing `ReferenceClass` objects as *environments*. Environments are R base types, consisting of a set of name-to-value mappings and a pointer to an enclosing environment (the root environment being the *empty environment*). R uses environments for name lookup, i. e., when R searches for a name's definition, it first searches for the name in the current environment. If the name is not defined in the current environment, R sequentially searches through the enclosing environments. By assigning the attributes of an object in a shared environment and setting this environment as enclosing environment for the object's methods, attributes can reliably be accesses from within methods without interference from variables in other environments. Thus, to avoid passing large data as arguments to a function, the data can be stored in an object's attribute and the function can be implemented as a method, accessing that attribute from within the function's body.

Because R provides means to perform complex statistical analysis (most importantly linear, and non-linear regression, ANOVA and time series analysis using ARIMA models), it is suited for analyzing and predicting response times. Supplementing R applications with a web based graphical user interface is possible using shiny. Thus, shiny can be used to build a control center that makes use of R's features for statistical analysis for anomaly detection. Finally, accessing monitoring data stored in Elasticsearch from R is possible using the elastic package.

Chapter 3

Performance Test

This chapter details on the test setup and test procedure. We used two physical machines: one to run FLS VISITOUR Server (*VISITOUR Server test instance*) and the other to generate the workload for the former (*load generator instance*). The load generator ran JMeter version 3.1, the VISITOUR Server test instance ran 64 bit VISITOUR version 1606.3200.810.1, with German road network data version 201606.

We start with a description of the test design in Section 3.1. Thereafter, Section 3.2 describes the configuration and the setup procedure used to initialize the VISITOUR Server Test Instance. Thereafter, Section 3.3 describes the test procedure that was performed by the Load Generator. Section 3.4 describes the used hardware. The methods used to collect the response time, workload, and job size data generated by the performance test are described in Section 3.5, and finally, Section 3.6 describes the results of the analysis of the data.

3.1 Design

In Section 2.4.7 we concluded, that the response time of VISITOUR's appointment proposal service depends both on, workload and job size. The job size of an appointment proposal request depends on several *job size parameters*: (1) the number of tours that are included in the optimization, (2) the number of jobs that are included in the optimization, (3) the number of candidate positions, (4) the number of requested optimization iterations, and (5) distance calculations. The workload at any point in time is given by the number of concurrently processed requests.

To generate response time data for subsequent analysis we issued `Call` requests, varying both, job size parameters and the number of concurrently processed requests:

- The number of tours was explicitly varied through using three optimization radii. The random geographical distribution of employees and jobs in the test setting provided a further implicit variation of this parameter (see Section 3.2).
- To vary the number of jobs, we needed to create confirmed appointments in our test setting. The jobs corresponding to these appointments would then be included in subsequent optimizations. This was done through establishing different *schedule saturation levels* in the test setting (see Section 3.2.2). The jobs scheduled at each level were randomly selected in a preceding step.

- The number of candidate positions was varied explicitly via the manipulation of the optimization radius. The use of different schedule saturation levels resulted in a further implicit variation of this parameter.
- The number of requested optimization iterations was explicitly varied by using three different settings for this server configuration parameter.
- The necessity for distance calculations was eliminated in the test setting through initializing the VISITOUR Server test instance with a fully populated distance matrix (see Section 3.3.1).
- To vary the number of concurrently processed appointment proposal requests we used four threads, each generating series of requests, with a randomly selected delay between subsequent requests (see Section 3.3.3).

The used optimization radii were $R := \{150\,\text{km}, 300\,\text{km}, 450\,\text{km}\}$. The used saturation levels were $L := \{\texttt{low}, \texttt{mid}, \texttt{high}\}$, and the different settings for requested optimization iterations were $I := \{1000, 2000, 3000\}$. Thus, this test design yields a total of $|R| \cdot |L| \cdot |I| = 3^3 = 27$ groups. For each group we generated 600 observations, resulting in a total of $27 \cdot 600 = 16\,200$ observations. Table 3.1 gives an overview over the used test groups.

3.2 Test Setting

VISITOUR is a real world route scheduling software that performs route scheduling within a given settings of employees and jobs located at different geographical positions.

Therefore, we created a standard *test setting* wherein appointment proposals for single jobs were requested for a given *target date*. The use of a fixed test setting ensures comparability between different executions of this performance test. Hence, we automated the establishment of the test setting in a VISITOUR Server and included it in the initialization phase of our VISITOUR Server test instance (see Section 3.3.1). The test setting consisted of:

1. a set E of 150 field service employees (see Section 3.2.1), and
2. a set J, of 5000 jobs (see Section 3.2.2).

Because VISITOUR loads road network data on a per-country basis, we restricted all employee and job locations to Germany. As target date we chose Monday, 03. April 2017, because it is no holiday in Germany, and it is a date after the submission of this thesis and thus did not require schedule optimization in the past to be enabled during the execution of the test.

The specified test setting required VISITOUR to handle at most 5150 locations; 150 employee base locations (for each employee we used the same postal address for start and end base location) and 5000 job locations. Thus, the distance matrix had less than 5150×5150 entries, because employees and jobs could share locations. Therefore, we were able to use a default sized standard distance matrix (see Section 2.4.4) in our VISITOUR Server test instance.

3.2.1 Generation of Field Service Employees

For the test we created a set E of 150 field service employees at random locations within Germany. Each employee was generated with a deployment radius of 450 km, and a tour length threshold of 2000 km. Thus, all employees included in an optimization should be able to serve the target job (see Section 2.4.3). Field service employees were created with daily working hours 08:00 - 16:00 and being present on the target date (no sick leave or holiday-related absence). For simplification the generated field service employees were

Table 3.1. Test Groups

Saturation Level	Optimization Radius	Proposal Iterations	Group No.
low	150 km	3000	0
		2000	1
		1000	2
	300 km	3000	3
		2000	4
		1000	5
	450 km	3000	6
		2000	7
		1000	8
mid	150 km	3000	9
		2000	10
		1000	11
	300 km	3000	12
		2000	13
		1000	14
	450 km	3000	15
		2000	16
		1000	17
high	150 km	3000	18
		2000	19
		1000	20
	300 km	3000	21
		2000	22
		1000	23
	450 km	3000	24
		2000	25
		1000	26

given no breaks and no overtime working window. Thus, all scheduling was bound to the continuous scheduling window 08:00 to 16:00.

The employee base locations (city, street, and zip code) were created using VISITOUR's `RandomAddress` web service (see Section 2.4.12). In an additional filtering step postal addresses on German islands without a road link to German mainland road network were excluded to ensure that all employees had access to German mainland road network (see Appendix A.1). We also ensured that the employee *deployment areas* (the area given by the deployment radius around the employee's base location) put together covered the entire German mainland. This was done visually by plotting employee base locations on a map together with the smallest deployment radius used in the test (see Figure 3.1). The plot indicated, that all locations on German mainland were covered by at least one employee's deployment area. Additionally, a pretest ensured, that all jobs in the test setting could be served by at least one field service employee (see Section 3.2.2).

The generated addresses (city, street, and zip code) were stored as text file `employee-test-set.json` in JSON format. This text file can be found in the supplementary material of this thesis (see Appendix A.3).

3.2.2 Generation of Jobs

For the test we created a set J of 5000 jobs at random locations in Germany. Each job was assigned an ID of the form `jXXXX` where `XXXX` was a uniquely assigned job number starting at `0000` and going to `4999`. These IDs were used as external job IDs on job creation (see Section 2.4.8). Thereby, we were able to request appointment proposals and cofirmation for jobs using their external ID instead of having to specify internal job IDs. All jobs were created to be served on the target date between 08:00 - 16:00. This single, continuous scheduling window was chosen as *target scheduling window* to let VISITOUR generate exactly one appointment proposal per `Call` request (see Section 2.4.6). The duration of each job was set to 1 h. Consequently, each employee could serve up to 8 jobs per day when they required no traveling time. Therefore, the total number of servable jobs could not be greater than $|E| \cdot 8 = 150 \cdot 8 = 1200$.

The addresses (city, street, and zip code) used as job locations were created using VISITOUR's `RandomAddress` web service (see Section 2.4.12). Similarly, to the generation of employee base locations, postal addresses on German islands without a road link to German mainland road network were excluded in an additional filtering step to ensure that all jobs were accessible from German mainland road network (see Appendix A.1). The generated jobs (city, street, zip code, duration and external job ID) were stored as text file `job-test-set.json` in JSON format. This text file can be found in the supplementary material of this thesis (see Appendix A.3).

We conducted a pretest to ensure that every job in the test setting could be served by at least one field service employee. In this pretest we requested an appointment proposal for each job in J using VISITOUR's `Call` web service. The pretest was run in test setting where no jobs where scheduled.

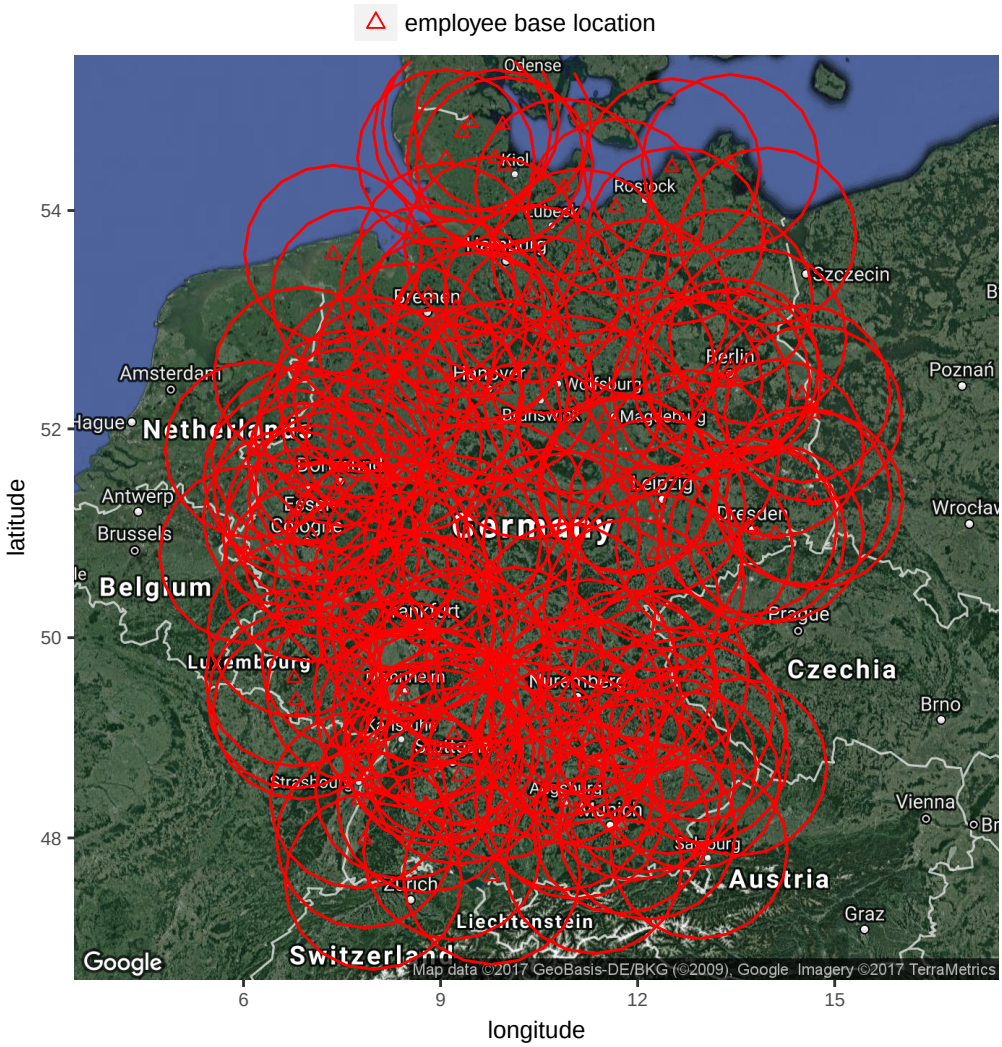

Figure 3.1. Base locations of all employees in the test setting with deployment radius 150 km.

Because the generation of candidate positions (and therefore the response time of the `Call` service) depends on the schedule saturation (see Section 2.4.3), we included a manipulation of the schedule saturation in our test. To find suitable *schedule saturation levels*, we conducted another pretest to explore the total number of jobs that could be scheduled in the test setting. This pretest consisted of three subsequent request to the `Optimize` web service, requesting a scheduling for the entire target date using 10 000 iterations. All three requests resulted in the scheduling of 950 jobs. With respect to this empirically

Table 3.2. Schedule Saturation Levels

Level	Set of Jobs	No. of Scheduled Jobs
low	J_{low}	300
mid	J_{mid}	600
high	J_{high}	900

found upper bound of jobs that could be served in the test setting, we defined the three schedule saturation levels shown in Table 3.2. For each schedule saturation level $l \in L := \{\text{low}, \text{mid}, \text{high}\}$ we created a set of jobs $J_l \subset J$. To set VISITOUR to the desired schedule saturation level, we scheduled all jobs in the corresponding set J_l. For each set we stored the external IDs of the jobs as string array in JSON format in a separate text file, creating the files `saturation-level-low.json`, `saturation-level-mid.json`, and `saturation-level-high.json`. These text files can be found in the supplementary material of this thesis (see Appendix A.3).

3.3 Test Procedure

The test procedure (*test plan*) was divided into three phases: *initialization*, *warm-up*, and the *strain* phase. The initialization phase was used to establish the test setting and to configure VISITOUR Server accordingly. During the strain phase, the workload and performance data for this analysis was captured. A warm-up phase was included, to *warm-up effects*. Effects that vanish over time are called warm-up effects. They bias only the first couple of response times in a test series. Warm-up effect result e. g., from loading data from secondary memory or just-in-time (JIT) compilation [see e. g., Waller and Hasselbring 2013; Singer 2003]. JIT compilation describes a technique, where source code is compiled to machine or byte code at run time as needed. JIT compilation is e. g., used in the JVM and the CLR, where VISITOUR server is executed. In the CLR compilation actions are usually triggered when a class is loaded (e. g., when the class is instantiated or one of its static methods is called for the first time). Thus, the first execution of a program or a section of code is often more expensive than every following execution. Warm-up phases are used to account for the effect of JIT compilation on response times in performance tests. Rohr [2015] e. g., used a warm-up of three minutes to counteract JIT compilation effects. However, Waller and Hasselbring [2013] noted that warm-up phases are affected by hardware and software configurations and therefore have to be individually chosen for every test based on preliminary empirical examinations of response times.

We used JMeter (see Section 2.6) to execute the test plan. Because we repeatedly observed unpredictable stopping of execution after several thousand requests, we split the test plan into several pieces. One piece for initialization and warm-up, and three pieces for the strain phase. We used a shell script (`jmeter.bat`) to sequentially execute the test

pieces. This shell script repeatedly started JMeter in command line mode executing on test piece after another. The initialization and warm-up procedure called JMeter with the test plan file `init_and_warm_up.jmx`, the strain phase pieces called JMeter with the test plan file `strain.jmx` using different command line parameters. The shell script and the test plan files are available in the supplementary material of this thesis (see Appendix A.3).

3.3.1 Initialization Phase

During the initialization phase the following steps were performed:

1. Set the tours threshold (`SUCHANZAHL`) to $|E| = 150$. Thus, in the test setting no field service employee was excluded from an optimization because the total number of employees in the optimization exceeded the tours threshold (see Section 2.4.3). In consequence, the number of employees in every optimization was fully determined by the job and employee base location and the current optimization radius

2. Set the maximum number of jobs on a tour (`OPT_MAXBESUCHEJETOUR`) to 8. As the name suggests, this parameter is used as a threshold to limit the number of jobs on a tour. We chose a threshold of 8, because in the test setting every field service employee could serve only up to 8 jobs on his tour.

3. Set default values for creating field service employees (see Section 2.4.3). This ensured that all field service employees in the test setting (which were created in step 5) were created as stated in Section 3.2.1. Listing 3.1 shows the SQL `UPDATE` query that was used for this.

   (a) Set default deployment radius (`OPT_MITARBEITER_EINSATZ_RADIUS`) for the creation of employees to 450 km.

   (b) Set default tour length threshold (`OPT_MITARBEITER_MAXTOURKM`) for the creation of employees to 2000 km.

   (c) Set default daily working hours to 08:00 - 16:00.

4. Create field service employees of the test setting. This was done using VISITOUR's `FieldManager` web service (see Section 2.4.11) and a JMeter loop controller (see Section 2.6). The controller read all employee base addresses from `employee-test-set.json` and issued a `FieldManager` request for each address.

5. Create jobs of the test setting. This was done with a request to VISITOUR's `Calls` web service (see Section 2.4.8), passing the data of all 5000 jobs to VISITOUR for batch job creation.

6. Initialize the distance matrix to eliminate the need for distance calculations during the strain phase using the `Optimize` web service (see Section 2.4.9). This was done through:

(a) Set the threshold applied to optimization iterations triggered by `Optimize` (`ITERATIONEN_LOKAL`) to 1, in order to minimize the duration of the optimization.

(b) Request a combined optimization of all jobs on the target date through the `Optimize` web service. After this request, the VISITOUR test instance has a fully populated distance matrix. As a side effect, `Optimize` also writes the created schedule to the data base.

(c) Delete the created schedule for the target date while preserving the distance matrix. This was done through:

   i. A request to the `DeletePlanning` web service (see Section 2.4.10). This deletes all scheduled job appointments, but not the distance matrix. As a side effect, the formerly scheduled jobs receive the status *escalated*.

   ii. Reset the status (`STATUS`) of all jobs to *new* (1) and clear all entries from tour dates, i. e., set `TOUR_DATUM` to `NULL` for all jobs. Thus, the formerly scheduled jobs are re-included in subsequent scheduling requests.

(d) Reset `ITERATIONEN_LOKAL` to 10 000.

The execution of the initialization phase took 9.02 min.

Listing 3.1. SQL query used to configure VISITOUR Server

```
UPDATE DEV_VISITOUR_WR.DBO.VTSSTAMM SET
SUCHANZAHL = 150,
OPT_MAXBESUCHEJETOUR = 8,
OPT_MITARBEITER_EINSATZ_RADIUS = 450,
OPT_MITARBEITER_MAXTOURKM = 2000,
ARBEITSZEIT_MO_VON = '08:00',
ARBEITSZEIT_MO_BIS = '16:00',
ARBEITSZEIT_DI_VON = '08:00',
ARBEITSZEIT_DI_BIS = '16:00',
ARBEITSZEIT_MI_VON = '08:00',
ARBEITSZEIT_MI_BIS = '16:00',
ARBEITSZEIT_DO_VON = '08:00',
ARBEITSZEIT_DO_BIS = '16:00',
ARBEITSZEIT_FR_VON = '08:00',
ARBEITSZEIT_FR_BIS = '16:00';
```

3.3.2 Warm-Up Phase

A pretest consisting of 10 000 `Call` requests suggested that 5000 requests would be sufficient to prevent warm-up effects in the strain phase. Consequently, we chose a warm-up of 5000 request for this test, requesting appointment proposals for randomly chosen jobs from the entire set of generated jobs J (see Section 3.2.2). The execution of the warm-up took 3.3 min.

3.3.3 Strain Phase

During the strain phase the following steps were performed *for each* optimization radius $r \in R := \{150\,\text{km}, 300\,\text{km}, 450\,\text{km}\}$:

1. Set the optimization radius to r.
2. For each schedule saturation level $l \in \{\text{low}, \text{mid}, \text{high}\}$:
   (a) Read the external IDs of the jobs in J_l from the corresponding text file (see Section 3.2.2).
   (b) Set the VISITOUR Server test instance to the desired schedule saturation level. Each job is scheduled through a request to VISITOUR's `Call` web service using the job's external ID and function code 2, causing the job to be automatically scheduled at the best possible position. This was done using a JMeter loop controller (see Section 2.6).
   (c) For each iteration value $i \in I := \{1000, 2000, 3000\}$:
      i. Set the number of optimization iterations to i.
      ii. Request appointment proposals for $n := 600$ randomly chosen currently unscheduled jobs using 4 threads (each thread executing 150 request using a JMeter loop controller). Between subsequent requests each thread waits for a random time interval $0 \leqslant t \leqslant 15\,000\,\text{ms}$. The time intervals were generated with a random uniform timer (see Section 2.6).
   (d) Delete the created schedule before establishing the next saturation level. This was done similarly to the deletion of the schedule during the initialization phase (see Section 3.3.1):
      i. Send a request to the `DeletePlanning` web service.
      ii. Reset `STATUS` to 1 and `TOUR_DATUM` to `NULL` for all jobs.

Note, that this test procedure did not simulate real world workload patterns of VISITOUR Server, because at the time of writing, there was no data on customer workload profiles available.

3.4 Hardware

Table 3.3 shows the hardware configuration of the VISITOUR Server test instance and the load generator. Because we used to separate physical machines, the generation of the workload and the generation of the appointment proposals did not compete for system resources during the test.

VISITOUR Server uses a lot of memory due to holding road network data (approx. 800 MB for Germany) as well as the potentially large distance matrix in memory during execution.

Table 3.3. Hardware Configuration

	VISITOUR Server Test Instance	Load Generator
Model	Dell Latitude E6540	Dell Optiplex 9010
RAM	8 GB	16 GB
OS	Windows 7 Professional SP1 64 bit	Windows 8.1 Professional 64 bit
CPU Model	Intel Core i7-4810MQ	Intel Core i7-3770
CPU Cores	4	4
CPU Clock Rate	2.80 GHz	3.40 GHz

Thus, we saw a risk of using too much of the physical machine's memory during test execution, resulting in an increased *page fault* rate and possibly even *thrashing*. A page fault occurs when a process accesses a page from its virtual memory that is not present in the physical memory and therefore must be fetched from secondary memory. Thrashing is the state when a machine spends so much time transferring pages to and from secondary memory, that these disk IO operations slow down the overall system performance. This can happen e.g. when the working sets of all processes together are larger than the machines physical memory. Consequently, page faults and particularly thrashing can diminish service response times. We conducted a pretest to screen the memory utilization during the execution of the test. The pretest ran the entire test procedure described in Section 3.3. We used *Microsoft Resource Monitor* utility shipped with Windows 7 SP1 to monitor the memory utilization during the test execution. The memory usage of the VISITOUR process peaked at approx. 1 200 000 kB = 1200 MB during the test. The overall memory utilization stayed below 4.2 GB. Thus, the occurrence of increased page fault rates or thrashing seemed unlikely during the test.

3.5 Data Collection

The data for the analysis was extracted from the local log file of the VISITOUR Server test instance. We extracted the `Call` requests, responses, and the corresponding `MonitoringInformation` log entries. Exemplary `Call` request, and response log entries can be found in Listing 3.2, and Listing 3.3. Listing 3.4 shows an exemplary `MonitoringInformation` log entry.

The first line of every log entry consists of the start marker `#####`, followed by a single character serving as *IO indicator*, an eight digit *log entry ID*, the log entry timestamp, and the *log entry type*. Possible IO indicators are `<` for requests, `>` for responses, and `i` for information log entries. The log entry ID is used to identify related log entries. E. g., the request, the response, and the `MonitoringInformation` belonging to the same appointment proposal generation process are all logged with the same log entry ID. To illustrate this, we chose log entries from the same appointment generation process as example log entries

(see Listing 3.4 through Listing 3.3). The timestamp of a log entry is composed of the date and the time, separated by a space. The date consists of four digits for the year, two for the month, and two for the day. The time consists of two digits for the hour, two for the minute, two for the second, and three for milliseconds. The type of a log entry is in most cases the name of the SOAP service that produced the log entry. In cases, where log entries are not directly connected to a service, the type is a name that identifies the kind of the entry. This applies e. g., to `MonitoringInformation` log entries. Thereafter, `Call` request, and response log entries display the information from the corresponding HTML request respectively response. The HTML header is thereby separated from the payload by an empty line.

Listing 3.2. Example `Call` request log entry

```
#####< 00013663 20170302 190112327 Call
POST http://192.168.1.29:82/VTS/Call
POST /VTS/Call
Connection: keep-alive
Content-Length: 251
Content-Type: text/xml; charset=utf-8
Host: 192.168.1.29:82
User-Agent: Apache-HttpClient/4.5.2 (Java/1.8.0_121)

<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
  <env:Body>
    <m:Call xmlns:m="http://www.tourenserver.de">
      <m:FunctionCode>1</m:FunctionCode>
      <m:VTID>265290</m:VTID>
    </m:Call>
  </env:Body>
</env:Envelope>
```

Listing 3.3. Example `Call` response log entry

```
#####> 00013663 20170302 190115322 Call
Server: FLS Server 1606.3200.810.1
Connection: keep-alive
Content-Type: text/xml; charset=UTF-8
Content-Length: 771

<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsi
    ="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="http://www.w3.org
    /2001/XMLSchema" soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding
```

```
    /" xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
    <CallResponse xmlns="http://www.tourenserver.de/">
      <CallResult>0</CallResult>
      <VTID>265290</VTID>
      <InfoText></InfoText>
      <Appointments>
        <Appointment>
          <FunctionCode>1</FunctionCode>
          <Status>0</Status>
          <Date>2017-04-03T09:46:00+02:00</Date>
          <Time>2017-04-03T09:46:00+02:00</Time>
          <Detour>112</Detour>
          <FMVTID>2908</FMVTID>
          <FMExtID>2908</FMExtID>
          <Info></Info>
          <Cost>271</Cost>
        </Appointment>
      </Appointments>
    </CallResponse>
  </soap:Body>
</soap:Envelope>
```

Listing 3.4. Example `MonitoringInformation` log entry

```
#####i 00013663 20170302 190115322 MonitoringInformation
<optinfo>
  <type>proCallProposals</type>
  <iterations>3000</iterations>
  <duration>2883</duration>
  <calls>212</calls>
  <tours>54</tours>
  <cand>11453</cand>
  <dimaadd>0</dimaadd>
  <dimafound>621</dimafound>
  <dimahit>13382478</dimahit>
  <dimamiss>0</dimamiss>
</optinfo>
```

The logging of `MonitoringInformation` entries is not enabled by default in VISITOUR Server. It requires the *Monitoring Extension* that was specifically developed to supply information on request job size parameters for this thesis. `MonitoringInformation` log entries

Table 3.4. Fields in `MonitoringInformation` log entries

Field	Description
type	Operation that triggered the optimization. Possible values are: `proCallProposals` (optimization was trigged by a `CallProposal` or a `Call` request using function code 1), and `proOptimize` (optimization was triggered by an `Optimize` request).
iterations	Number of requested optimization iterations.
duration	Duration of the optimization in ms.
calls	Number of jobs included in the optimization (see Section 2.4.3).
tours	Number of tours included in the optimization (see Section 2.4.3).
cand	The *cumulated sum* of candidate positions: the candidate positions for the target job, plus the candidate positions for all jobs affected by domino effects in the course of the optimization, plus one dummy slot to put escalated jobs in.
dimaadd	Number of entries added to the distance matrix in the course of the generation of the appointment proposal (see Section 2.4.3).
dimafound	Number of distances needed for the generation of the appointment proposal and already present in the distance matrix.
dimamiss	Number of unsuccessful distance matrix lookup operations.
dimahit	Number of successfull distance matrix lookup operations.

supply additional information on the generation of appointment proposals and confirmations, triggered through the `Call`, `CallProposal` and `Optimize` web services. Table 3.4 gives an overview over the fields of `MonitoringInformation` entries. The provided fields include several job size parameter, the *duration* of the optimization, and measures of distance matrix operations. The logged job size parameters are *tours*, *calls*, *cand* and *iterations*, providing information on the number of included tours, and jobs, the number of candidate positions, and the number of requested optimization iterations (see Table 3.4 for a description). The optimization duration is the duration between the start end the end of the optimization, i. e., the *response time* of the mere optimization. Although the optimization code section is performed mutually exclusive, there is no guarantee that the optimization is also executed uninterruptedly. It could be interrupted by a thread, that executes a different code section, not falling under the control of the mutual exclusion condition, like e. g., a thread handling a SOAP request or response. Hence, the workload can affect the optimization duration. Consequently, the optimization was not included into the subsequent analysis as a job size parameter.

`MonitoringInformation` supplies four measures of distance matrix operations: dimadd, dimafound, dimamiss, and most importantly, dimahit. Dimaadd is the number of distances that were added to the distance matrix in the course of the optimization. Dimafound is the complementary measure, capturing the number of distance matrix entries needed for

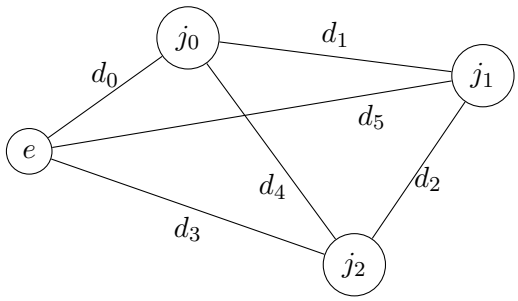

Figure 3.2. Example route scheduling setting with one field service employee e and three jobs j_0, j_1, and j_2

the optimization but already pre-calculated in the distance matrix. In contrast, dimahit and dimamiss are measures of distance matrix *lookup operations*. Dimahit is the number of successful lookup operations, and dimamiss is the number of unsuccessful. To illustrate the relationship between all four measures of distance matrix operations, consider the route schedulin setting displayed in Figure 3.2. This setting consists of one employee base location e, and three job locations j_0, j_1, and j_2, and the distances d_0 through d_5. Let us assume that in this setting, d_0, d_2, d_3, and d_4 are present in the distance matrix, whereas d_1, and d_5 are not. An optimization that investigates the routes $r_0 : e \to j_0 \to j_2 \to j_1 \to e$, and $r_1 : e \to j_2 \to j_1 \to j_0 \to e$, would thus result in:

- dimaadd $= 2$, because d_5 is necessary for r_0, and d_1 for r_1,
- dimafound $= 4$, because d_0, d_2, d_3, and d_4 were used in the investigation and already present in the distance matrix,
- dimahit $= 6$, because d_0, d_2 were both successfully looked up twice, and d_3, d_4, were each successfully looked up once,
- dimamiss $= 2$, because d_5 was looked up unsuccessfully during the investigation of r_0, and d_1 was looked up unsuccessfully during the investigation of route r_1. Note, that in cases where the distance matrix is large enough to hold all distances needed in the course of an optimization, dimamiss is always equal to dimaadd, because misses occur only once. However, if a distance matrix is used that holds only the n nearest locations, like e. g., a rectangular distance matrix (see Section 2.4.4), misses can occur and trigger distance calculations without adding the calculated distance value to the matrix.

Although the job size parameters tours, calls, candidates, and requested iterations were identified as important job size predictors (see Section 2.4.2), dimahit can be considered an even more important job size parameter. Dimahit reflects not only the number of investigated routes, jobs, and candidate positions for the target job, but also the routes, jobs, and candidate positions used to account for domino effects. Furthermore, dimahit is also affected by the number of performed optimization iterations.

The log processing setup implemented for the control center (see Section 4.3) was used to extract the log entries from the log file after the test procedure was finished. This setup consisted of Filebeat, Logstash and Elasticsearch (see Section 2.5). Filebeat was used to forward `Call` and `MonitoringInformation` log entries to Logstash. For a detailed description of the utilized Filebeat configuration see Section 4.3. Logstash was used to aggregate the data extracted from the `Call` request, response, and `MonitoringInformation` log entries. For each appointment proposal, the information from the request, the response, and the `MonitoringInformation` was combined in a single event. Then we used Logstash to compute a total *response time* in milliseconds from the request and response timestamp and store it in the combined event. Thereafter, the event written as type `call_aggregate` into the index `performance-test-calls`. Table 4.1 gives an overview over the structure of this index. A detailed description of the utilized Logstash configuration can be found in Section 4.3.

As a final step, we loaded the data into R using the `queryElasticsearch` method of the `VisitourInstance` class from the implementation of the control center to retrieve the data from Elasticsearch as `data.frame` (see Section 4.2). `queryElasticsearch` computes a *concurrency score* for each appointment proposal request. This concurrency score was calculated similarly to the response time based pwi_1 metric used by Rohr et al. [2010] to compute the workload intensity produced by concurrently executed function calls. For a request r, received at time $\mathrm{st}(r)$ and returned at time $\mathrm{rt}(r)$, we define the concurrency score function $\mathrm{cs} : r \to [1, \infty) \subset \mathbb{R}$ as:

$$\mathrm{cs}(r) := \frac{1}{\mathrm{st}(r) - \mathrm{rt}(r)} \sum_{s \in R} \mathrm{overlap}(r, s)$$

where R is the set of requests generated by the performance test procedure, and overlap is the overlap interval of two requests, defined as:

$$\mathrm{overlap}(s, r) := \begin{cases} 0 & \text{if } \mathrm{st}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{st}(r) - \mathrm{rt}(s) \geqslant 0 \wedge \mathrm{rt}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{rt}(r) - \mathrm{rt}(s) > 0 \\ \mathrm{rt}(s) - \mathrm{st}(r) & \text{if } \mathrm{st}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{st}(r) - \mathrm{rt}(s) < 0 \wedge \mathrm{rt}(r) - \mathrm{st}(s) \geqslant 0 \wedge \mathrm{rt}(r) - \mathrm{rt}(s) \geqslant 0 \\ \mathrm{rt}(s) - \mathrm{st}(s) & \text{if } \mathrm{st}(r) - \mathrm{st}(s) \leqslant 0 \wedge \mathrm{st}(r) - \mathrm{rt}(s) < 0 \wedge \mathrm{rt}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{rt}(r) - \mathrm{rt}(s) \geqslant 0 \\ \mathrm{rt}(r) - \mathrm{st}(s) & \text{if } \mathrm{st}(r) - \mathrm{st}(s) \leqslant 0 \wedge \mathrm{st}(r) - \mathrm{rt}(s) < 0 \wedge \mathrm{rt}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{rt}(r) - \mathrm{rt}(s) < 0 \\ 0 & \text{if } \mathrm{st}(r) - \mathrm{st}(s) < 0 \wedge \mathrm{st}(r) - \mathrm{rt}(s) < 0 \wedge \mathrm{rt}(r) - \mathrm{st}(s) \leqslant 0 \wedge \mathrm{rt}(r) - \mathrm{rt}(s) < 0 \\ \mathrm{rt}(r) - \mathrm{st}(r) & \text{if } \mathrm{st}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{st}(r) - \mathrm{rt}(s) < 0 \wedge \mathrm{rt}(r) - \mathrm{st}(s) > 0 \wedge \mathrm{rt}(r) - \mathrm{rt}(s) < 0 \end{cases}$$

Figure 3.3 displays the possible overlap scenarios expressed in the cases of the overlap function on a timeline.

In words, $\mathrm{cs}(r)$ is the sum of all overlaps, divided by the request's duration. Because every request overlaps itself, $\mathrm{cs}(s) \geqslant 1$ for all $r \in R$. Like [Rohr et al. 2010] we assume, that time can be expressed in natural numbers, i. e., $\mathrm{st} : r \to \mathbb{N}$, and $\mathrm{rt} : r \to \mathbb{N}$. Because computers measure time only with a certain precision (usually full milliseconds), this assumption is met when computer measured times are used.

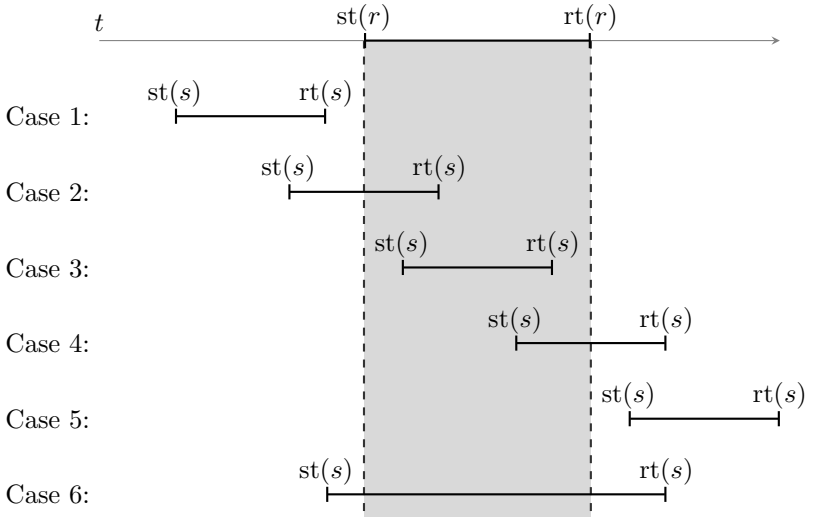

Figure 3.3. Possible overlaps of two requests r, s

3.6 Results

We performed a preliminary analysis to inspect the data for distance calculations, time-outs, and warm-up effects. The results of this analysis are described in Section 3.6.1. Thereafter, we analyzed the distributions of the concurrency score (Section 3.6.2) and the job size parameters (Section 3.6.3) generated by the performance tests. In Section 3.6.4, we investigated their influence on the response time of VISITOUR's appointment proposal service. We close the analysis with a comparison of different sliding window sizes for fitting prediction models (Section 3.6.5). All reported analysis were carried out with R (Section 2.7). The data and the used script file can be found in the supplementary material of this thesis (see Appendix A.3).

3.6.1 Preliminary Analysis

Distance calculations, time-outs, and warm-up effects (see Section 3.3) can lead to increased response times. Therefore, we performed a preliminary analysis to inspect the data for signs of these interfering influences. The analysis showed, that Logstash did not encounter time-outs during the performance test. I. e., the VISITOUR Server test instance responded to every appointment proposal request of the test within the chosen time-out interval of 10 min. Furthermore, the analysis showed, that no distance calculations were performed during the warm-up and strain phase (dimaadd was 0 for all observations).

Figure 3.4 shows the response times that were observed during the warm-up phase. We analyzed these response times for signs of warm up effects. Only the first response time

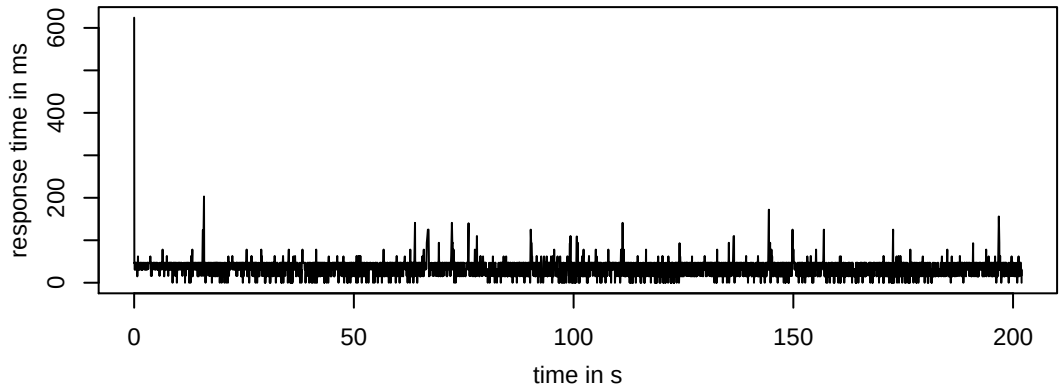

Figure 3.4. Response Times of the appointment proposal service during warm-up

(624 ms) appeared to be biased. Thereafter, the response time of the appointment proposal service appeared to be rather stable with a mean of 32.74 ms and a standard deviation of 16.04 ms.

3.6.2 Concurrency Score

Figure 3.5 shows the distribution of the concurrency score in the sample. The distribution shows that predominantly small values were observed. The computed sample mean, and the dispersion statistics also reflect this trend. The mean concurrency score was $M = 1.65$, the standard deviation was $sd = 0.76$, the 25 % quantile was $q_{0.25} = 1$, and the 75 % quantile was $q_{0.75} = 2.1$. Hence, less than 25 % of observed concurrency score values were greater than 2.

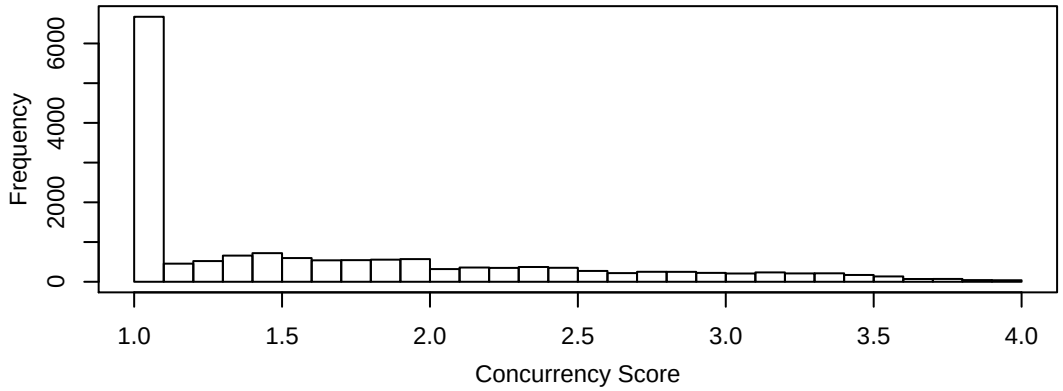

Figure 3.5. Distribution of the concurrency score in the strain phase

Table 3.5. Summary statistics of job size parameters

	M	sd	min	$q_{0.25}$	$q_{0.75}$	max
tours	47.98	31.36	2.00	18.00	72.00	130.00
calls	195.54	157.97	3.00	74.00	282.00	781.00
candidates	10885.57	12037.81	10.00	1369.00	15816.50	62493.00
dimafound	592.50	211.71	308.00	412.00	717.00	1340.00
dimahit	7612761.80	7470834.91	27132.00	2162340.25	10680318.00	39360737.00

3.6.3 Job Size Parameters

Figure 3.6 displays the distributions of the job size parameters tours, calls, candidates, dimafound, and dimahit (see Section 3.5). Table 3.5 shows the sample means and standard deviations. The distributions and the dispersion statistics indicate, that the manipulations of the job size parameters applied in the course of the performance test (see Section 3.1) resulted in a high variability of job size parameters.

3.6.4 Effects on Response Time

Figure 3.7 shows the response time plotted against the concurrency score and the job size parameters. The plots do neither indicate any particular ploynomial of exponential relationship between the response time and the concurrency score nor between the response time and the job size parameters.

As a first measure for the effect of the concurrent processing of requests on the response time of VISITOUR's appointment proposal service we computed the intercorrelation between the concurrency score and the response time. The found intercorrelation ($r = 0.57$) indicated, that concurrent processing influenced the response times in the test setting. However, Table 3.6 shows that dimaadd ($r = 0.69$) displayed a higher intercorrelation with the response time, and the parameters calls, candidates and dimafound displayed similar ones (all $0.52 \leqslant r \leqslant 0.55$).

High to medium intercorrelations were also found between the concurrency score and the job size parameters calls, candidates, dimafound, and dimahit respectively (all $r \geqslant 0.62$). Further high intercorrelations were found between the job size parameters tours, calls, and candidates (all $r \geqslant 0.8$). The lowest intercorrelation was found between the number of requested iterations and the response time ($r = 0.24$).

The various high intercorrelations between workload and job size parameters indicated, that some observed correlations with the response time are in fact caused by a third parameter effecting the response time as well as the parameter in question. I. e., the direct effects of some parameters on the response time could vanish, if the effects of all other parameters are accounted for. To identify such fake direct effects and to find an economical prediction model that limits the number of predictors to the ones with the

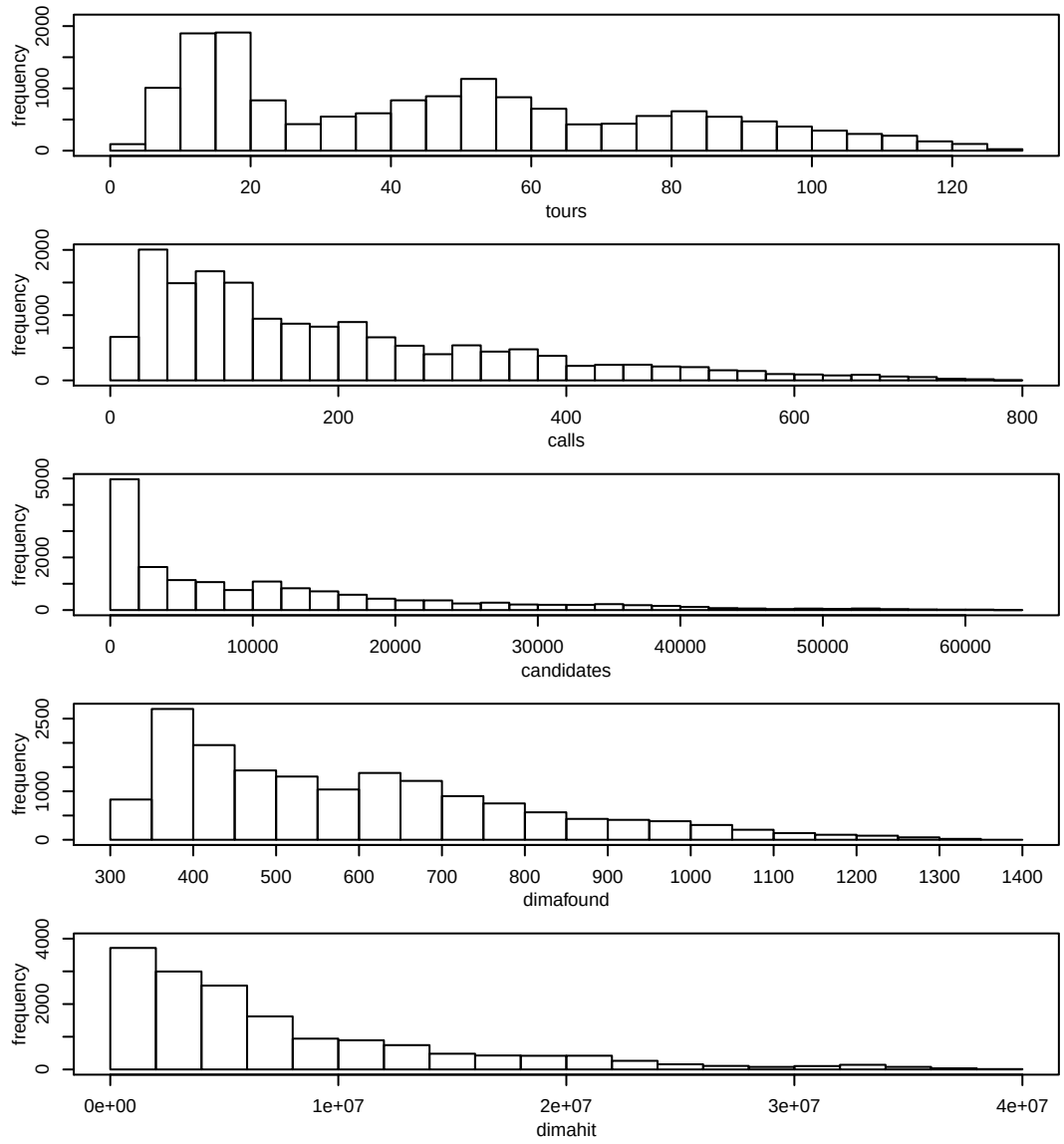

Figure 3.6. Distributions of job size parameters

strongest influences on the response time we conducted a stepwise regression analysis. Because Figure 3.7 did not indicate polynomial or exponential relationships, we used linear regression. An ARIMA model was also considered inappropriate, because the data was not simulated using a realistic workload or job size profile. Hence, all periodic patterns in the

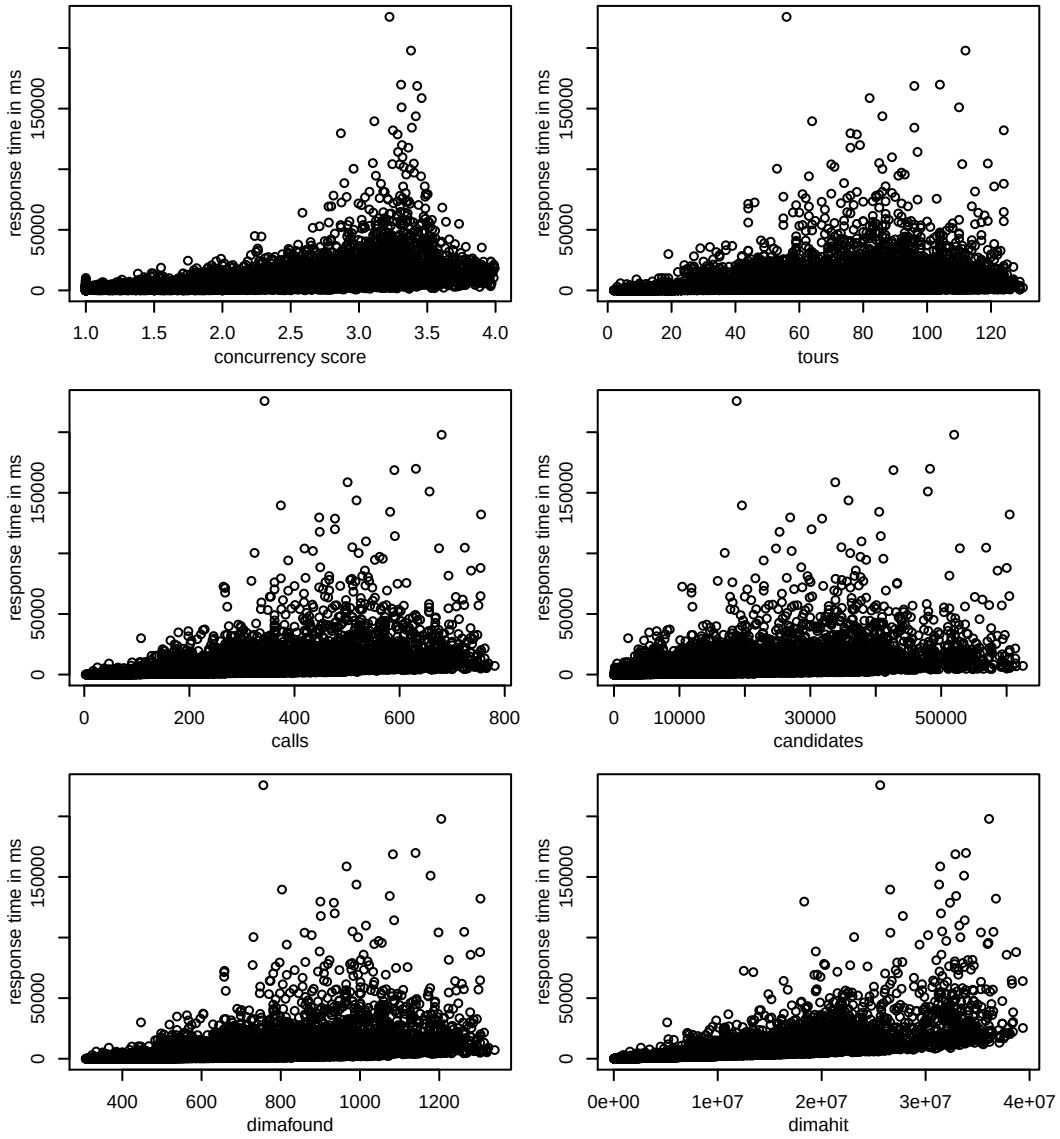

Figure 3.7. Response time plotted against concurrency score and job size parameters

data are at best coincidental.

We started the stepwise regression with a model with no predictors, predicting the response time solely by the sample's mean response time (*base model*). Then, we added predictors to the model step by step. In each step, we added the predictor accounting for

most of the models remaining RSS (see Section 2.2). Then we recomputed the RSS for the new model and continued to the next step. Table 3.7 shows the steps we performed. In the first step we added dimahit as a predictor to the model, accounting for 47 % of the RSS of the base model. Then we added the concurrency score the model, accounting for another 1 % of the remaining RSS. Because adding more predictors to the model would only result in a reduction of the remaining RSS by less than 1 %, we refrained from performing more steps. In total, the inclusion of dimahit and the concurrency score as predictors resulted in a reduction of the RSS by 47.5 %, compared to the base model. Thus, in the test setting, this prediction model reduced the variance of the response time by almost half.

3.6.5 Sliding Window

In the previous section we used all available data to identify an efficient prediction model, using only the strongest influences on the response time as predictors. For real time monitoring, it is however often not desirable to fit a model to all available data. As the number of observations grows, the fitting of the model becomes more and more costly, gradually slowing down the monitoring application. Sliding Window based prediction approaches limit the fitting of prediction models to a fixed number of observations, and often provide sufficient precise predictions suited for anomaly detection (see Section 2.2).

Choosing the right window size is of critical importance, when sliding window based prediction is used. The keep the costs for computing the prediction low, a small window size is desirable. However, large window sizes might increase the precision of the prediction. To find a good default sliding window size to use in combination with the identified prediction model, we compared predictions generated with different sized sliding windows. For each sliding window size w we computed a resulting *mean absolute prediction error* as a measure of the precision of the generated predictions. This mean absolute prediction error was computed as $\frac{1}{16200-w} \sum_{i=w+1}^{16200} |y_i - \hat{y}_i|$ where y_i and $\hat{y}_i$ are the observed and the predicted

Table 3.6. Intercorrelations

	t_r †	(1)	(2)	(3)	(4)	(5)	(6)
tours (1)	0.36						
calls (2)	0.55	0.80					
candidates (3)	0.53	0.85	0.98				
requested iterations (4)	0.24	0.00	0.00	0.00			
dimafound (5)	0.52	0.90	0.98	0.98	0.00		
dimahit (6)	0.69	0.55	0.81	0.76	0.41	0.77	
concurrency score	0.57	0.45	0.67	0.62	0.29	0.63	0.77

† response time

Table 3.7. Stepwise Regression

	RSS	AIC
Base Model		296181
tours	1.23×10^{12}	293994
calls	9.81×10^{11}	290289
candidates	1.02×10^{12}	290858
requested iterations	1.33×10^{12}	295201
dimafound	1.03×10^{12}	291136
dimahit	7.48×10^{11}	285909
concurrency score	9.46×10^{11}	289711
+ *dimahit*		285909
calls	7.48×10^{11}	285908
candidates	7.48×10^{11}	285905
dimafound	7.48×10^{11}	285901
tours	7.47×10^{11}	285885
requested iterations	7.46×10^{11}	285861
concurrency score	7.41×10^{11}	285744
+ *concurrency score*		285744
calls	7.4×10^{11}	285737
candidates	7.41×10^{11}	285744
dimafound	7.4×10^{11}	285728
tours	7.39×10^{11}	285714
requested iterations	7.39×10^{11}	285703

response time of the i-th observation (the i-th appointment proposal request). Each $\hat{y}_i$ was generated by fitting a regression model that used dimahit and the concurrency score as predictors to the data from the preceding sliding window, i. e., the observations $i - w$ to $i - 1$ inclusively. This model was then used to compute the prediction $\hat{y}_i$ from dimahit and concurrency score of the i-th observation. To illustrate this, consider the generation of $\hat{y}_{201}$ using a sliding window of size 200. $\hat{y}_i$ would be computed as $\hat{y}_{201} = a + bx_{201} + cz_{201}$, where x_{201} is the dimahit, and z_{201} is the concurrency score form the 201-th observation, and a, b, and c are the coefficients resulting from fitting a regression model to the data from the first to the 200-th observation.

Note, that for each sliding window, the first w observations are not included the computation of the mean absolute prediction error. The reason for this is that for these observations the number of preceding observations is smaller than the specified sliding window size. Because fitting models to fewer observations can affect the precision of the predictions generated with these models, we decided to excluded the first w observations from the computation of the mean absolute prediction error for each sliding window size

Table 3.8. Mean error of sliding window based response time prediction for different sliding window sizes

(a) Selected Window size from 25 to 6400

Window Size	Mean absolute error
25	2103.54
50	2028.86
100	1990.35
200	2003.16
400	2051.1
800	2115.6
1600	2218.61
3200	2422.16
6400	3069.49

(b) Window sizes from 50 to 800 in steps of 50

Window Size	Mean absolute error
50	2028.86
100	1990.35
150	1986.67
200	2003.16
250	2010.23
300	2024.57
350	2036.15
400	2051.1
450	2064.35
500	2078.48
550	2093.03
600	2095.16
650	2100.31
700	2104.22
750	2109.56
800	2115.6

w.

For comparison, we also computed the mean absolute prediction error for the non-sliding window based version of the model. This mean absolute prediction error was computed as $\frac{1}{16200}\sum_{i=1}^{16200}|y_i - \hat{y}_i|$, where every prediction $\hat{y}_i$ was calculated as $\hat{y}_i = a + bx_i + cz_i$ using the coefficients a, b, and c from the model fitted to the entire data. The calculated value was 2808.

In a first analysis, we investigated sliding windows sized 25 to 6400, step by step doubling the sliding window size. Table 3.8a displays the results from this analysis. The smallest mean absolute prediction errors were found for window sizes below 800. Therefore, we performed a more fine-grained analysis to investigating sliding window sizes from 50 to 800, in steps of 50. The results of this investigation are displayed in Table 3.8b. The results indicated, that 150 observations is a suitable default sliding window size for predicting response times of VISITOUR's appointment proposal service using a linear regression model with predictors dimahit and concurrency score.

Because constraining prediction models to sliding windows affects the precision of the prediction, we performed a final analysis to compare the effects of adding dimahit and concurrency score as predictors to the base model when a sliding window of 150 observations was used. The different models were compared with regard to their mean

Table 3.9. Mean absolute prediction error of prediction models with window size 150

	Mean absolute prediction error
(base model)	2288.78
+ dimahit	2077.25
+ concurrency score	1986.67

Note: The first 150 observations were excluded from the computation of the mean absolute prediction error of each model. For these observations the number of preceding observations for fitting a prediction model was smaller than the chosen sliding window size.

absolute prediction error in the sample. This analysis showed, that adding dimahit as predictor to the prediction model reduced the mean absolute prediction error by 9.24 %. Adding concurrency score as predictor reduced the mean absolute prediction error by another 3.96 % compared with the base model. Table 3.9 shows the mean absolute prediction errors computed for all three models.

Chapter 4

Control Center

This chapter details on the control center that was envisioned in goal G3 (see Section 1.2.3). The control center was developed with R using the R web framework shiny (see Section 2.7). We start by describing the *use cases* identified for the control center in Section 4.1. A use case describes the interactions between an actor (e. g., a user) and a software to achieve a goal. To implement the control center, we used the object-oriented programming style supplied by R's `methods` package. Section 4.2 details on the classes of the control center implementation. Thereafter, we describe the deployment of the control center and the setup used for monitoring VISITOUR Server instances (Section 4.3). Finally, Section 4.4 describes the control center's user interface and functionality.

4.1 Use Cases

We identified six use cases for the control center. The use cases *real time anomaly detection*, and *analyze static monitoring data* are concerned with the analysis of response time, workload, and job size data. In the former the operator monitors a VISITOUR Server instance in real time, in the latter, the operator queries the instance's monitoring data for data from a specified time interval. If an anomaly is detected in a monitored instance, the operator is notified by email. This is described by the use case *warn operator*. The *add instance* and *remove instance* use cases deal with adding resp. removing instances to resp. from the control center, and the *set defaults* describes the setting of default values for creating new instances, and for the email configuration used for sending email notifications. Figure 4.1 shows the identified use cases in a use case diagram.

4.2 Classes

Real time prediction of response times for multiple VISITOUR Server instances using a sliding window approach requires the control center to handle potentially large monitoring data that is subject to frequent updates, each adding and dropping parts of the data, while retaining most of it. Hence, VISITOUR Server instances can be modeled as objects, whose state changes whenever the data is updated. Consequently, we chose to use `ReferenceClasses` to implement the control center, because they are suited for modeling objects with complex

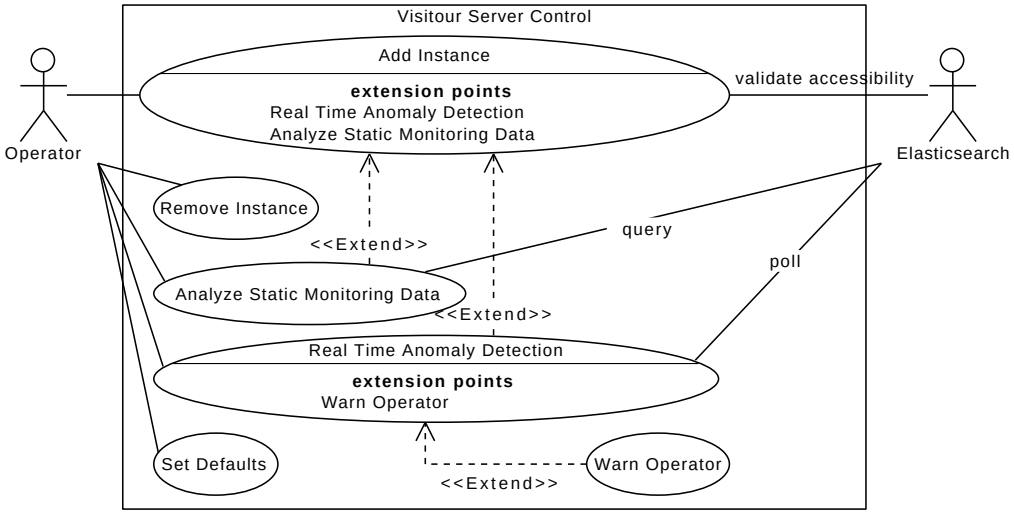

Figure 4.1. Use case diagram for the control center.

states [Wickham 2014] and can reduce the overhead caused by passing large data to functions. The class diagram in Figure 4.2 shows the classes that were used.

The `VisitourInstance` class models a single running VISITOUR Server that is monitored for performance anomalies. A `VisitourInstance` is connected to *one* Elasticsearch index, from which the monitoring data is retrieved. The interactions with Elasticsearch are implemented, using features provided by the elastic package (see Section 2.7). The connection to Elasticsearch is specified through a host name resp. IP, a port, and an index name. Host and port are part of the instance's `VisitourOptions` object (see below). The used transport protocol is HTTP. The connection can be established using the `connect` method. This method will throw an error, if the host or the index is unreachable. The data in the specified index is assumed to be structured as described in Section 4.3. This is however currently not checked by `connect`. Querying Elasticsearch for monitoring data is possible via the `queryElasticsearch` method. This method retrieves documents from the selected index and returns the data as a `data.frame`. A `data.frame` is a matrix-like data structure, where every column can store values of another type. We could e. g., create a `data.frame` where the first column stores `integer` values, the second `character` (R's string type), and the third `numeric` (R's floating point number type). Thus, `data.frames` can be used to mix data of different types for combined analysis. Every call to `queryElasticsearch` must specify `start` and `end` of a time interval. This interval is used as a filter in the query to Elasticsearch, limiting the returned results to documents whose `response_timestamp` (see Section 4.3) falls into the specified interval. If called with `caching = TRUE`, the returned `data.frame` is cached in the object's `data` attribute, and the start resp. end time used to filter documents are saved in the object's `start` resp. `end` attribute. In all calls to `queryElasticsearch` that use caching,

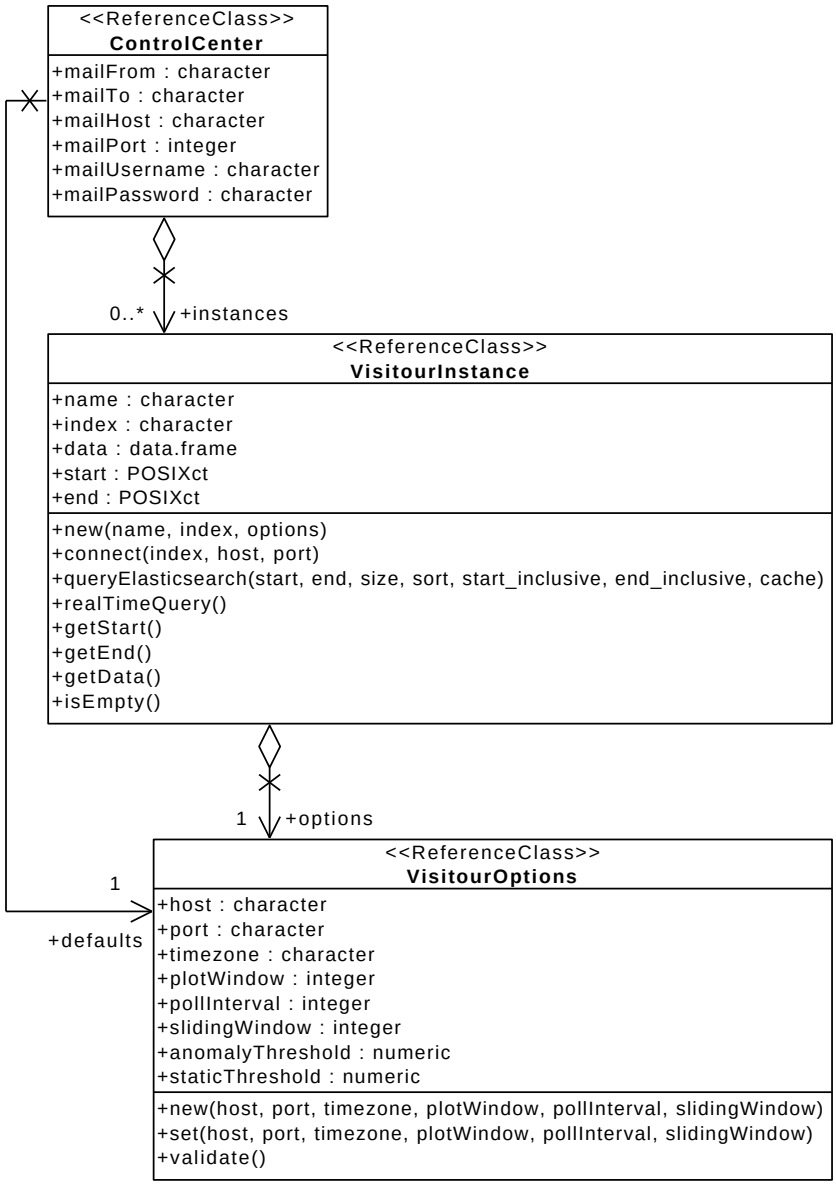

Figure 4.2. VISITOUR Control Center Class Diagram

monitoring data will be loaded from the cache, if possible. This is done by computing the overlap of the interval given by the *attributes* `start` and `end`, and the interval given by the `start` and `end` *parameter* of the function call. The data falling into the overlap is loaded

from cache (i. e., the object's `data` attribute), and only the missing data is retrieved from Elasticsearch. The cache is designed to hold the result of a single call to `queryElasticsearch`. I. e., after a cached call to `queryElasticsearch`, `start` and `end` attributes are set to the `start` and `end` values used in the call, and all data dropping out of this interval is removed from the cache. Caching monitoring data was added because we observed noticeable lags, when combining several thousand results retrieved from Elasticsearch to a `data.frame`. All real time monitoring features of the control center (see Section 4.4) use caching to reduce these lags,

Live monitoring of response time, workload, and job size data is accomplished in the control center by regularly polling the Elasticsearch indices for recently added data. We added the `realTimeQuery` method to conveniently query Elasticsearch for data falling into the interval between the end of the cached data and the current system time. This method is a wrapper, calling `queryElasticsearch` using the `end` attribute (if available) as `start`, and the system time as `end` parameter. For convenience, we also added getter methods for `data`, `start`, and `end`, as well as a method (`isEmpty`) to test if the cache is empty. Note, that a non-empty cache can contain zero data, if there is no workload and response time data for the specified interval available in the index. For such a cache `isEmpty` returns `FALSE`. The rationale behind this is that the corresponding interval can be safely excluded from subsequent cached request.

The `VisitourOptions` class is a container for options usable both, as part of the configuration of a single `VisitourInstance`, and as default values for creating new `VisitourInstances` in the control center. The attributes `host` and `port` are used for connecting to a system running Elasticsearch. The `timezone` is used to set the time zone for querying an index, i. e., the time zone of the `start` and `end` parameter of `queryElasticsearch`, and the `start` and `end` attribute of an `VisitourInstance` object. Currently the only supported time zone is Coordinated Universal Time (UTC). The `plotWindow` specifies the length of the time interval (in seconds) that is displayed in the *real time view* of the control center (see Section 4.4). More precisely, the start of the displayed interval is given by the system time minus the specified `plotWindow`. Its end is given by the system time. The (approx.) number of seconds to wait between polls to an Elasticsearch index is given by the `pollInterval`. The `slidingWindow` attribute specifies the size of the sliding window (in observations) used to fit the statistical model for predicting the most recent response time (see Section 3.6.5). By default it is set to 150. The `anomalyThreshold`, and the `staticThreshold` attributes store thresholds for anomaly detection. The `staticThreshold` is applied directly to monitored response times, i. e., if a reponse time exceeds this threshold, an email notification is send to the operator, provided that this is the first anomaly since the last reset of the instance's *warning cache*. This cache tracks if a warning had been issued for this instance before. The purpose of this is to prevent the control center from sending an arbitrary large number of warning emails to the operator. The `anomalyThreshold` is applied to the *anomaly score* of an observation, which is computed as proposed by Henning [2016] as $A_D(y, \hat{y}) := \frac{y-\hat{y}}{y}$, where y is the observed response time, and $\hat{y}$ is the predicted. Similar to the `staticThreshold`, a notification email

is send to the operator, if an anomaly score exceeds the `anomalyThreshold`.

Finally, the `ControlCenter` class models a control center capable of monitoring multiple VISITOUR Server installations at once for performance anomalies and sending notification emails. Every `ControlCenter` maintains its own set of `VisitourOptions` used as default values for creating new `VisitourInstances`. For the email notification setup, the `ControlCenter` the credentials (email address, host, port, username and password) for the email accound used to send, and a recipient email address (`mailTo`). The control center shiny app is currently designed to be used by a single operator. Therefore, only a single global `ControlCenter` object is maintained by the server. This object is shared between all sessions and persisted across sessions.

4.3 Deployment

The control center shiny app can be deployed on a Shiny Server, e. g., by placing the app's files (`server.R`, `ui.R`, `global.R`, and the files in the `www`, and `modules` subdirectory) in a subdirectory of the server's directory for hosting apps. The app files are available in the supplementary material of this thesis (see Appendix A.3). A full setup capable of real time monitoring VISITOUR Server instances requires however additional components: a running Filebeat instance on every monitored system, as well as a Logstash and an Elasticsearch instance for processing and storing monitoring information. The full setup necessary for monitoring VISITOUR Server instances is shown as deployment diagram in Figure 4.3.

On every VISITOUR Server, Filebeat (see Section 2.5.1) is used to forward `Call`, `CallProposal` and `MonitoringInformation` log entries from the server's local log file to Logstash. Other log entries are currently discarded to keep the monitoring related network traffic to the necessary minimum. When monitoring multiple server system at once, it is necessary to provide means for mapping monitoring data to a corresponding VISITOUR Server instance. Because VISITOUR can be hosted on customer systems, neither the host name nor the IP (the system could be part of a private network behind a gateway) of the server provide reliable identification. Therefore, each Filebeat instance is assigned a name. The name can be set in the instance's configuration file. It can be deliberately chosen and is included in the information send to Logstash. Thus, Logstash can identify the source of a forwarded log entry, provided each monitored VISITOUR Server runs its own Filebeat instance, using a unique name. By default, Filebeat also adds auxiliary data to each forwarded log entry. This auxiliary data consists of the host name, the path to the log file, and Filebeat's version number. A template configuration file for Filebeat (`filebeat.yml`) is available in the supplementary material of this thesis (see Appendix A.3).

Logstash (see Section 2.5.2) is configured to extract timestamps, job IDs, log entry IDs, function codes and return codes from the forwarded log entries for `Call` resp. `CallProposal` request, and the corresponding responses. Because `CallProposal` requests are semantically similar to `Call` requests using function code 1 (see Section 2.4.8), we decided to treat them alike, using the same Elasticsearch document type for indexing and supply-

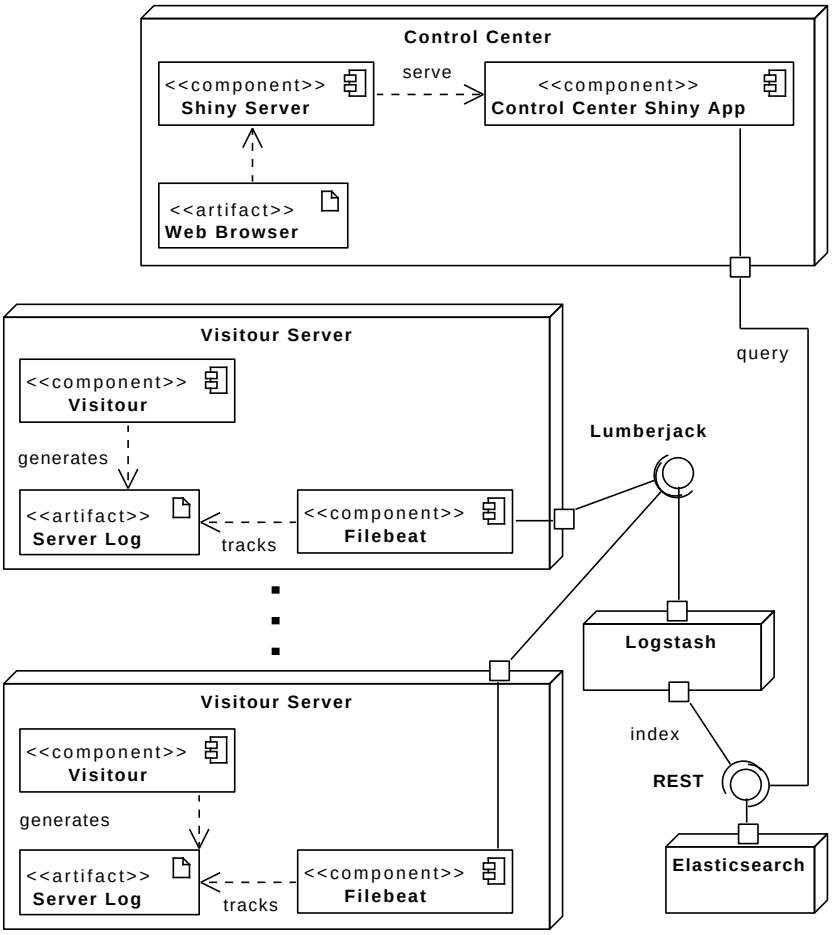

Figure 4.3. Setup for colleting monitoring data from VISITOUR Server installations.

ing the same information. Therefore, Logstash is configured to fill in the information `CallProposal` requests and responses lack compared to their `Call` counterparts (see Section 2.4.8). `CallProposal` requests are assigned function code 1, and responses are assigned return code 0, if they contain an appointment proposal and return code 10, if they do not. From the `MonitoringInformation` log entries Logstash extracts all supplied job size parameters (see Section 3.5). In a final step, all information belonging to an appointment proposal (i. e., the information extracted from the corresponding request, response, and monitoring information), is combined into a single event using Logstash's `aggregate` filter. Matching requests to responses and monitoring information is thereby done using the log entry ID, which is the same for all log entries belonging to the same request. In the course of combining the information, Logstash also extends the information in the combined

event by a total response time, computed from the difference between the timestamps of the response and the request in milliseconds. The combined events are then indexed into Elasticsearch using the document type `call_aggregate`, while the original events are dropped. The name of the target index is thereby generated from the name assigned to the Filebeat instance by appending `-calls`. Thus, all log entries forwarded from an instance named `customer-x` will be stored in the index `customer-x-calls`. If necessary, this index will be automatically created using the template displayed in Listing A.1. If the index is created beforehand, the user has to make sure, that its mapping for type `call_aggregate` conforms to the mapping indicated by Listing A.1. Otherwise, indexing the combined events will return an error. The auxiliary data provided by Filebeat is also written to the index. The rules file for handling the forwarded log entries (`visitour_monitoring.conf`), as well as the configuration file (`logstash.yml`) used for Logstash are available in the supplementary material of this thesis (see Appendix A.3).

Elasticsearch (see Section 2.5.3) is used to persist monitoring data. For each monitored system, the monitoring information is stored in a dedicated index. Thus, querying monitoring data from multiple system has to be done by specifying multiple index names in the query. A description of the fields available in each index can be found in Table 4.1. Listing A.1 shows the template used to generate indices for storing monitoring data.

4.4 User Interface

The control center uses a sidebar layout. The sidebar contains one tab for each monitored VISITOUR Server instance, giving access to the instance's dashboard (*instance dashoard*), one tab to add new instances (*add instance view*), and one tab to set default values for creating new instances and specifying the email configuration used for sending email notifications (*settings view*). A button to hide the sidebar is available in the control center's title bar.

The instance dashboard uses a tabbed layout, displaying three tabs in a horizontal tab bar positioned underneath the title bar. The *real time* tab opens a view for real time monitoring (*real time view*). The *query* tab opens a view for displaying monitoring data on a query basis (*query view*). Changing the settings of an instance is possible via the *instance management view*. By default, the real time view is presented to the user, when the instance dashboard is visited. Figure 4.4 shows the real time view of an example VISITOUR Server instance, Figure 4.5 shows the query view. Both views display response time, workload, and job size data. The most important difference between them is that the real time view is updated periodically, while the query view is not. The real time view is targeted at supporting software operators in detecting performance anomalies (use case *real time anomaly detection*; see Section 4.1), whereas, the query view can be used for retrospective analysis of monitoring data (use case *analyze static monitoring data*; see Section 4.1).

The real time view displays response times, workload, and job size parameters (see Section 3.5) of the appointment proposal service, plotted over a time axis. Because anomaly detection is based on response times, the response time plot is positioned at the top of

Table 4.1. Fields used in Elasticsearch indices for storing monitoring data (document type `call_aggregate`)

Field	Description
`tours`	`tours` field of `MonitoringInformation` log entry (see Table 3.4).
`dimaadd`	`dimaadd` field of `MonitoringInformation` log entry (see Table 3.4).
`optimization_type`	Operation that triggered the optimization. For possible values see Section 3.5.
`duration`	Time needed to generate the appointment proposal (time between the `request_timestamp` and `response_timestamp`) in ms.
`log_entry_type`	Name of the used SOAP service. Possible values are `Call` or `CallProposal`.
`response_timestamp`	Timestamp extracted from the request log entry.
`request_timestamp`	Timestamp extracted from the response log entry.
`source`	Path to the log file from which the entry was extracted. The path is relative to the root directory of the source system.
`beat_name`	Name assigned to the Filebeat instance (see Section 4.3).
`dimahit`	`dimahit` field of `MonitoringInformation` log entry (see Table 3.4).
`beat_version`	Filebeat's version number.
`requested_iterations`	`iterations` field of `MonitoringInformation` log entry (see Table 3.4).
`candidates`	`cand` field of `MonitoringInformation` log entry (see Table 3.4).
`calls`	Number of jobs included in the optimization (see Section 2.4.2).
`call_result`	Return code of the request. For possible values see Section 2.4.8.
`call_id`	Internal ID of the target job.
`log_entry_id`	ID of the log entry (see Section 3.5).
`optimization_duration`	`duration` field of `MonitoringInformation` log entry (see Table 3.4).
`dimafound`	`dimafound` field of `MonitoringInformation` log entry (see Table 3.4).
`dimamiss`	`dimamiss` field of `MonitoringInformation` log entry (see Table 3.4).
`timeout`	Boolean value indicating if Logstash's `aggregate` filter timed out, while waiting for a response log entry. The threshold for a timeout is set to 600 s in the used Logstash configuration (`visitour_monitoring.conf`)
`host_name`	Host name of the machine running VISITOUR Server.
`function_code`	Function code used in the request. For possible values see Section 2.4.8.

the real time view. The plots of the workload and job size parameters are displayed in a shared panel using a tab layout. The plot of each parameter is displayed in a dedicated tab. The updating frequency of all plots is given by the poll interval set for the corresponding VISITOUR Server instance. The real time view provides a button *remove warning* that resets the instance's warning cache, when clicked (see Section 4.2).

The query view displays the same plots as the real time view, but is targeted at analyzing static monitoring data, like e. g., extracted from a log file submitted by a customer for performance analysis. Therefore, this view is not updated automatically. Via the inputs for start and end, operators can deliberately set the displayed time interval. The query is executed, when the *update* button is clicked. If start and end are not specified, the application displays the entire data available in the instances Elasticseach index, provided

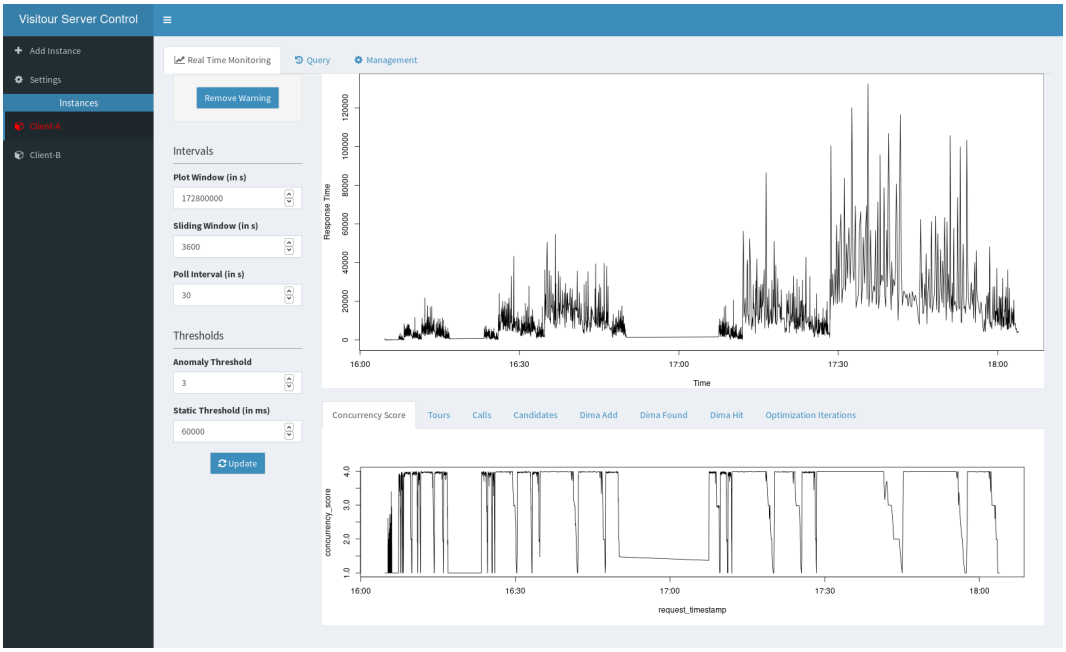

Figure 4.4. *Real time view* of a VISITOUR Server instance with opened concurrency score tab

that the `max_result_window` of the index is large enough. The `max_result_window` is part of the configuration of an index and is used by Elasticsearch to limit queries to this index to a maximum number of results. By default, it is set to 10 000. If a query tries to retrieve more results, Elasticsearch returns an error. The control center does currently not handle this kind of error, and therefore all queries issued from the control center are presently constrained to 10 000 results.

The instance management view allows operators to delete instances, via the *delete instance* button, which removes the instance from the control center when clicked. Figure 4.6 shows the management view of an example VISITOUR Server instance.

New VISITOUR Server instances can be added to the control center through the add instance view (see Figure 4.7). In this view operators can specify a name for the new instance and the Elasticsearch connection (host, port, and index) to use. Further inputs are supplied to set the polling interval, the plot window, the sliding window for response time prediction, and the time zone used for queries. The new instance is created, when the *add* button is clicked. On creation, a tab labeled with the instance's name is added to the sidebar as a link to the instance's dashboard.

The settings view (see Figure 4.8) allows operators to set default values for creating new instances and specify the email configuration used for sending email notifications. The default values for instance creation are pre-filled into the inputs of the add instance view,

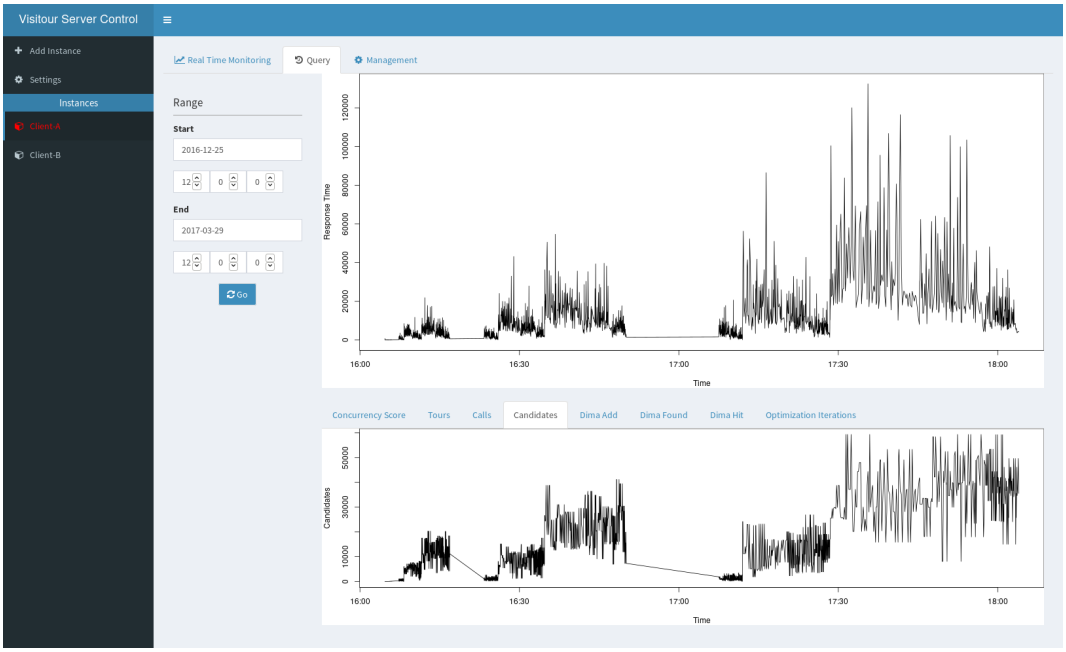

Figure 4.5. *Query view* of a VISITOUR Server instance with opened candidates tab

in order to speed up the creation of new instances. The idea behind the usage of default values for instance creation is, that in real time monitoring scenarios, instances often share certain settings, like e. g., host and port of the Elasticsearch instance. Thus, pre-filling the inputs for creating new instances relieves the operator from having to repeat frequently used settings. Default values can be set for:

- host and port used for Elasticsearch connections,
- the time zone to use for queries,
- the plot window,
- the sliding window,
- the poll interval, and
- the thresholds used for anomaly detection.

When the settings view is opened, the inputs in the settings view appear pre-filled with the current default values and email configuration. Changes to these values are saved when the *save* button is clicked, provided that the entered values are valid. This is checked using the `validate` method of the `ControlCenter`'s `VisitourOptions` instance (see Section 4.2). If the validation fails, the returned error is displayed in a notification appearing in the lower right corner of the display area.

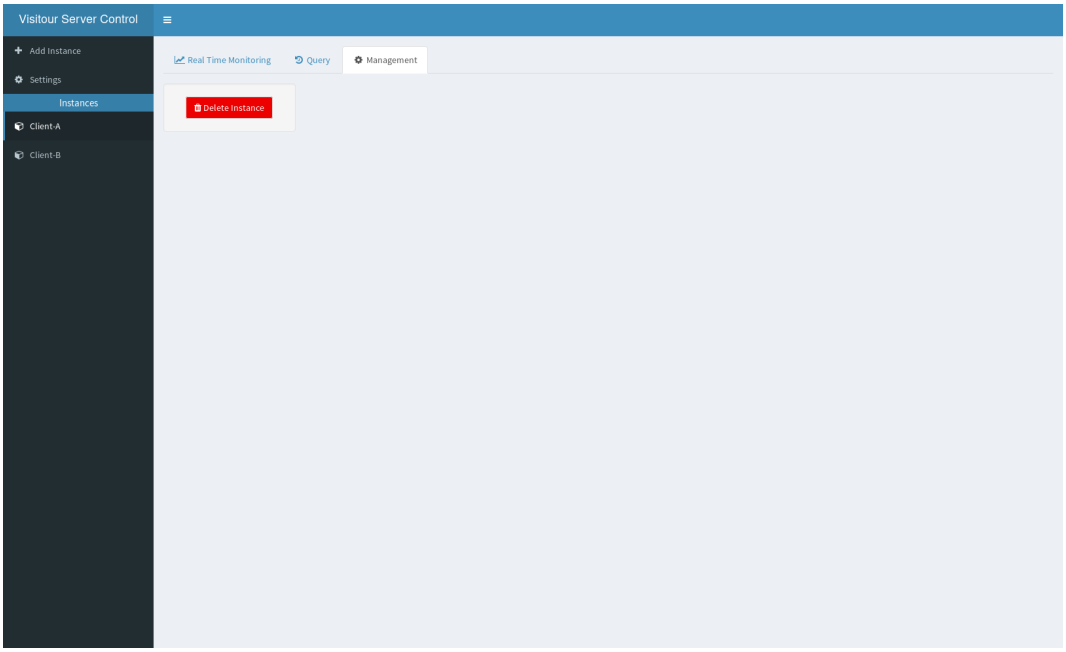

Figure 4.6. *Instance management view* of a VISITOUR Server instance

A dedicated *send test mail* button is provided to test the configuration for notification emails. When this button is clicked, a test mail is send to the specified recipient. If sending fails, an error is displayed.

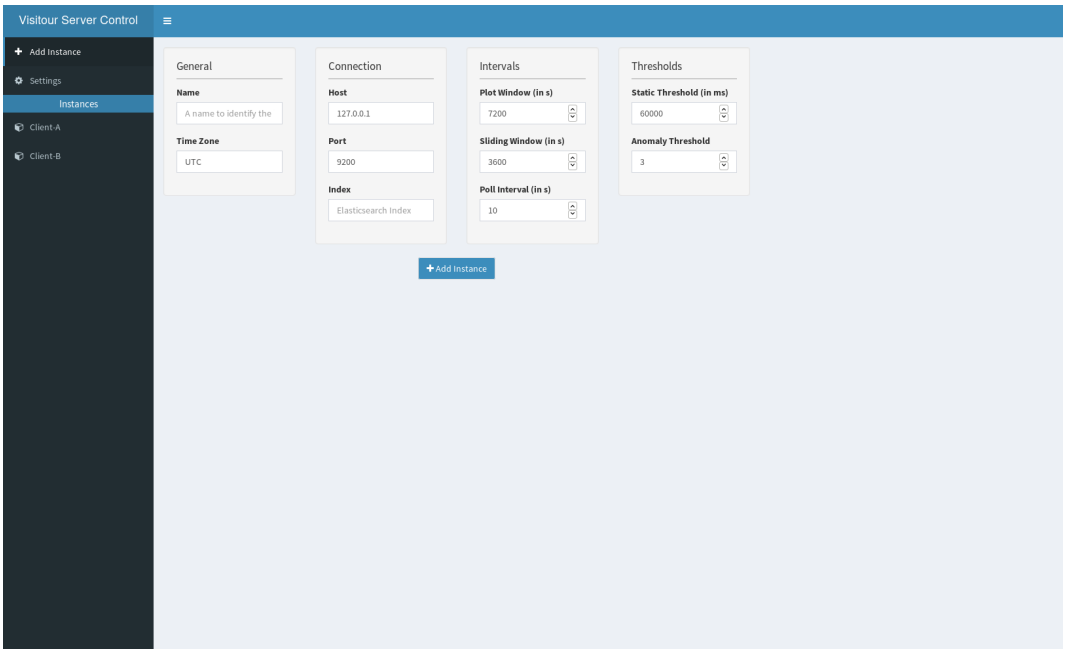

Figure 4.7. *Add instance view* of the control center

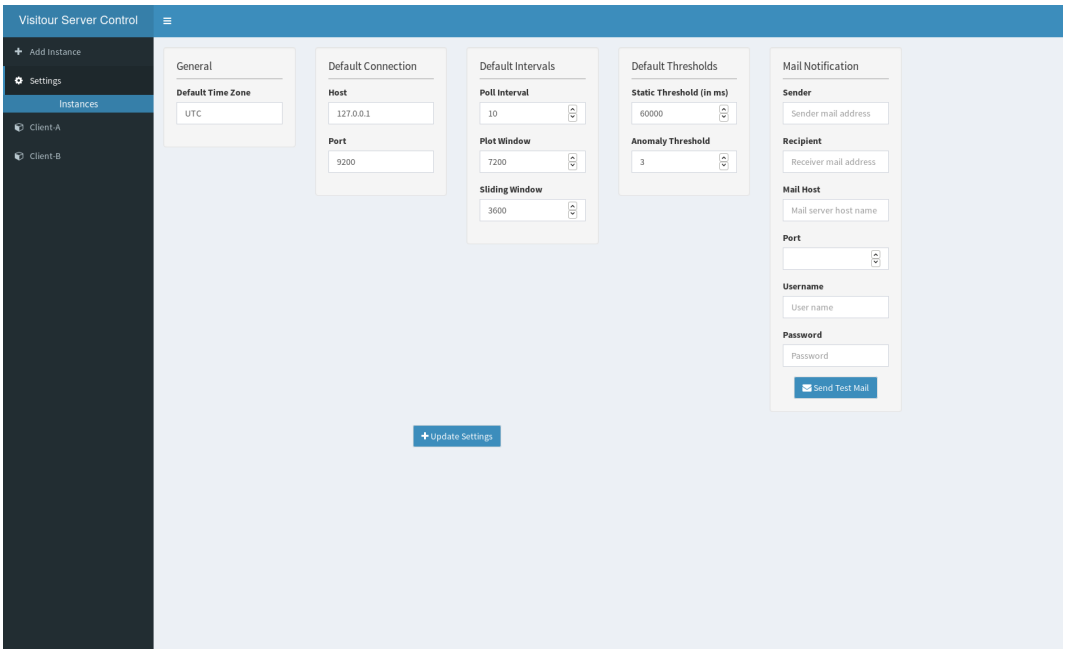

Figure 4.8. *Settings view* of the control center

Chapter 5

Evaluation

In this chapter we discuss the results from the conducted performance test (Chapter 3) and the implementation of the control center (Chapter 4). We start by an evaluation of the performance test results in Section 5.1. Thereafter, in Section 5.2 we discuss our implementation of the control center.

5.1 Performance Anomaly Detection

The investigation of the data showed no evidence that distance calculations and warm-up effects affected the response times during the strain phase (see Section 3.6.1). Thus, we considered the elimination of these interfering effects through the chosen initialization and warm-up procedure a success.

The analysis showed only a small intercorrelation between the number of requested iterations and the response time. This indicated that the number of requested iterations is only weakly related to the response time of VISITOUR's appointment proposal service. A possible explanation is that the number of requested iterations does not capture the complexity of the performed iterations. Iterations become e. g., more complex, the more domino effects have to be accounted for. Another possible explanation is that the number of actually performed iterations could differ from the number of requested iterations. In cases where scheduling is simple, VISITOUR could finish optimizing before the requested number of iterations is reached.

The analysis of the concurrency score showed a strongly skewed distribution in the sample, and a low sample mean and variance (see Section 3.6.2). Hence, the chosen manipulation of workload did mainly generate low workload. According to Rohr et al. 2010, there is no linear relationship between response time and workload. Instead, the effect of workload on response time increases with growing workload. Low workload does not significantly increase response times. For medium workload, the response time increases linearly with the workload, and for high workload, response time increases super-linear. Thus, we believe that the effect of workload on the response time of VISITOUR's appointment proposal service is underestimated in our analysis, because we did not generate medium or high workload. This lack of simulating high workload threatens the *external validity* of our findings. External validity is the extend to which experimental findings can be generalized to other contexts, most importantly real world contexts. The control center however might

still provide good anomaly detection, even in case of high workload, because we chose to keep workload as predictor in the implementation. Chapter 6 details on possibilities to adapt the test procedure and the test setting to cover high workload scenarios.

Another major thread to the external validity of our findings stems from the fact, that the test procedure did not simulate real world usage patterns of FLS VISITOUR Server. In a real world scenario, the workload of a VISITOUR Server instance could follow a periodic pattern to some degree, like many server applications. Workload could e. g., be higher during the evening time than during the day. A realistic simulation of workload was however not possible for two reasons. The main reason was that customer usage data is confidential and thus, can only be analyzed with customer approval. Because this thesis was intended as a first feasibility analysis, we refrained from approaching customers for their approval. The second reason was that VISITOUR Server is used by customers with very difference usage profiles and performance requirements. Firstly, customers differ greatly with regard to the number of field service employees and the number of jobs to manage. Secondly, customers use different VISITOUR Server configurations, and most importantly run VISITOUR with different extensions enabled. Extension can have a major influence on the response times of the appointment proposal service, e. g., through reducing the number of candidate position to investigate (see Section 2.4.2). Thus, even when customer workload profiles were available, they could not be easily compared to each other, and findings applying to one usage profile might not apply to other usage profiles. For the same reason, defining a representative usage profile from different customer profiles could turn out difficult.

In summary, the conducted performance test provides first evidence, that accounting for job size parameters of appointment requests could improve anomaly detection in server applications. However, to ensure the external validity of this finding, further investigations are necessary, examining other server applications, and high workload scenarios.

5.2 Control Center

As envisioned in Section 1.2.3 we implemented a control center to support the detection of anomalies in the performance of VISITOUR Server. This control center used the prediction model identified in the performance test to detect anomlies in the reponse time of the appointment proposal service.

The control center was targeted at being used by a single human operator and thus, maintains only one global `ControlCenter` object. Using the control center with multiple operators could induce race conditions, when two or more operators simultaneously add or remove monitored instances. This risk could be eliminated by maintaining a `ControlCenter` object for every operator. This would require the extension of the control center with a user management. The chosen implementation was designed to be ready for adding user management features. All user-specific features of a control center (i. e., the instances to monitor, and the default values for creating new instance) are part of the `ControlCenter`

class that represents an abstraction of a user-configured control center. Hence, handling multiple user-specific control center configurations can be realized by extending the control center shiny app with mechanisms to manage multiple `ControlCenter` instances. The professional version of Shiny Server can be used to supplement the control center with a user management. This would also increase the security of the web application, because Shiny Server provides means for secure user authentication by username and password.

Another area where the current implementation lacks security is the connection to Elasticsearch. Currently, the control center only support basic features for connecting to an Elasticsearch index, namely through specifying a host, a port, and an index name. It does not provide means to use Elastisearch's features for secure authentication, e. g., via a user name and a password. Furthermore, the control center currently relies on the insecure HTTP protocol for data exchange with Elasticsearch. The security of the transport could be increased by adding support for the more secure HTTPS protocol to the control center.

The handling of error arising from Elasticsearch queries is another area, where the implementation of the control center could be improved. Most importantly, the control center expects the data in the specified Elasticsearch index to have a certain structure, without pre-checking it. If the data is structured differently, the control center will most likely encounter an error when trying to query the index or when processing returned results. The resistance to this kind of errors could be improved by checking that the mapping of the selected index conforms to the mapping indicated by Listing A.1. This can be done by retrieving the current mapping of the selected index and comparing it with the required mapping before querying the index for data.

Another major point, that currently impaires the usability of the control center for analyzing large monitoring data is the limitation of queries to 10 000 results (see Section 4.2). By limiting the number of retrievable observations, this threshold can limit the size of the time interval displayable in the query and the real time view. The reason for this limitation is that by default Elasticsearch constrains the maximum number of documents that can be retrieved from an index by a single query to 10 000. Although this threshold could be raised, by increasing the index `max_result_window`, Elastic recommends to not exceed the default of 10 000, because heap memory and time needed for search requests are proportional to the number of documents affected by the request[1]. A possible solution to relieve the control center from this limitation is described in Chapter 6.

In summary, we implemented a control center that is capable of displaying monitoring data for several VISITOUR Server instances in real time and that uses the empirically identified prediction model to automatically detect anomalies in the appointment proposal service and notifies the operator when anomalies occur. The implementation reached a state, where the control center is ready to be used by a single human operator. The implementation can however be further improved by adding a user management, enhancing security, and extending the features for querying and connecting to Elasticsearch.

[1] Source: `https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-search-after.html`

Chapter 6

Future Work

As discussed in Section 5.1, the developed performance test procedure did not simulate medium and high workload scenarios. In future investigations, this shortcoming could be met in different ways. One possibility is to reduce the delay between subsequent `Call` requests issued by a thread. Another way is to increase the number of threads. This is however limited by the load generator's system resources, because only a certain number of threads can be executed concurrently without interfering to much with each other. A more promising approach would be to use multiple machines to generated appointment proposal requests. JMeter supports this kind of *distributed testing* by allowing a master-slave setup of JMeter instances. In this setup the JMeter master instance coordinates the test execution, whereas the JMeter slave instances receive test action commands from the master and execute the requested test actions. This feature could be used to simulate realistic medium or high workload scenarios, where multiple clients access the services provided by VISITOUR Server.

As indicated in Section 5.2 the current implementation of the control center could be enhanced by implementing means to retrieve more than 10 000 results from an Elasticsearch index. For this purpose, Elastic recommends using the `search_after` query parameter[1]. This parameter can be used to split a single query that would return more than 10 000 into multiple queries, based on the value of a field that holds a unique value for every document in the index. The idea is to specify a sorting on the selected field in the query, and then set an offset for returning results using `search_after`. More precisely, Elasticsearch will skip all results whose field value is smaller or equal (relative to the selected sorting), to the specified `search_after` and start returning results after the value is surpassed. Thus, retrieving more than 10 000 documents can achieved by setting `search_after` in each query (except the first) to the last value the preceding query returned for the selected field. Hence, the first query is used to retrieve the first 10 000 documents, the second query retrieves the next 10 000 documents, and so on. To use `search_after` in a query to Elasticsearch, the search must be specified using the JSON-based Elasticsearch Query Language. Such queries are supported by the elastic R package through the `body` parameter of the `Search` function.

Another interesting area for future work could be the implementation of the developed performance test procedure as *regression test*. Regression tests are standardized test procedures that are applied to every release of a software in order to detect performance declines

[1] Source: `https://www.elastic.co/guide/en/elasticsearch/reference/current/search-request-search-after.html`

across sequential releases. Using of our test procedure as regression test is possible, because the procedure was completely automated through a shell script that successively executes the JMeter test plans for initialization, warm-up and strain phase. The integration of the test procedure into a test infrastructure could be accomplished similarly to the integration of custom application benchmarks into a continuous integration system presented by Waller et al. [2015]. A shell script akin to that used by Waller et al. [2015] could be used to deploy new VISITOUR Server builds to a dedicated test machine and trigger the execution of the performance test script on another machine. After test execution, the log file would have to be automatically fetched from the machine that ran VISITOUR Server (e. g., through making the server's log directory accessible from the network) and send to a machine running Filebeat with our `filebeat.yml` configuration template (see the list of supplementary files in Appendix A.3) forwarding appointment proposal events to Logstash. Note, that the VISITOUR machine should not run Filebeat itself during test execution, because this would draw system resouces away from VISITOUR. The Logstash instance receiving the appointment proposal events from Filebeat can be configured with our `logstash.conf` (see the list of supplementary files in Appendix A.3) to store the log entries in an Elasticsearch index. From there, common regression test statistics, like mean response times could be extracted.

Chapter 7

Conclusions

In this thesis, we set out to empirically identify a method to detect anomalies in running VISITOUR Server instances while accounting for workload and request job size. We utilized this method for the implementation of a control center to support operators in detecting performance anomalies. As a foundation for both, the empirical examination of the influences of workload and job size on response times, and the control center, we created a setup for extracting workload, job sizes, and response time data from VISITOUR's local log file.

The empirical examination of the influence of workload, and job size on response times, exhibited first evidence, that accounting for job size can improve anomaly detection when a prediction based detection approach is used. The effect of workload on response times could however not be reliably investigated, because the analysis revealed an insufficient manipulation of the workload in the test procedure.

The implemented control center displays workload, job size and response time data for multiple VISITOUR Server instances in real time. It can automatically detect anomalies and notify the operator by email if necessary. Further development can enhance the implementation by adding a user management, improving error handling, security, and access to large scale monitoring data.

Appendix A

Appendix

A.1 Excluded Postal Addresses

Postal addresses on German islands without a road link to German mainland road network were excluded from the list of possible job and employee base locations generated for the performance test setting (see Section 3.2). Therefore, the following zip codes were excluded: 18565, 25845, 25846-25847, 25849, 25859, 25863, 25869, 25929-25933, 25938-25942, 25946-25949, 25952-25955, 25961-25970, 25980, 25985-25986, 25988-25990, 25992-25994, 25996-25999, 26465, 26474, 26486, 26548, 26571, 26579, 26757, 27498, and 83256.

A.2 Elasticsearch Index Template

Listing A.1 shows the `curl` command used to generate the index template used by Logstash to create new indices for storing monitoring data.

Listing A.1. `curl` command used to create the template `call_aggregates` which is used for generating new indices for storing monitoring data (identified through the suffix *-calls*)

```
curl -i  -XPUT 'http://localhost:9200/_template/call_aggregates' -d '{
  "template": "*-calls",
  "settings": {"number_of_shards": 1, "number_of_replicas": 1},
  "mappings": {
    "call_aggregate": {"_source": {"enabled": true},
      "properties": {
        "@timestamp": {"enabled": false, "type": "date"},
        "beat_name": {"type": "keyword", "ignore_above": 256},
        "beat_version": {"type": "text", "index" : true},
        "call_id": {"type": "long"},
        "call_result": {"type": "integer"},
        "function_code": {"type": "integer"},
        "calls": {"type": "long"},
        "candidates": {"type": "long"},
        "dimaadd": {"type": "long"},
        "dimafound": {"type": "long"},
```

```
        "dimahit": {"type": "long"},
        "dimamiss": {"type": "long"},
        "duration": {"type": "long"},
        "host_name": {"type": "keyword", "ignore_above": 256},
        "log_entry_id": {"type": "long"},
        "log_entry_type": {"type": "keyword", "ignore_above": 256},
        "operation": {"type": "keyword", "ignore_above": 256},
        "optimization_duration": {"type": "long"},
        "optimization_type": {"type": "keyword", "ignore_above": 256},
        "request_timestamp": {"type": "date"},
        "requested_iterations": {"type": "long"},
        "response_timestamp": {"type": "date"},
        "source": {"type": "text", "index": true},
        "tags": {"enabled": false},
        "timeout": {"type": "boolean"},
        "tours": {"type": "long"}
}}}}'
```

A.3 List of Supplementary Files

Table A.1 gives an overview over the files available in the supplementary material of this thesis provided on CD.

Table A.1. List of files provided on the CD containing the digital supplementary material of this thesis

Filename	Description
Files used in the performance test procedure	
`employee-test-set.json`	Postal addresses of field service employees.
`job-test-set.json`	Postal addresses of jobs.
`saturation-level-low.json`	External IDs of jobs scheduled in saturation level low.
`saturation-level-mid.json`	External IDs of jobs scheduled in saturation level mid.
`saturation-level-high.json`	External IDs of jobs scheduled in saturation level high.
`jmeter.bat`	Shell script used to sequentially execute the test plans for the initilization, warm-up and strain phase.
`init_and_warm_up.jmx`	JMeter test plan for the initialization and warm-up phase.
`strain.jmx`	JMeter test plan for the strain phase.
Results from the performance test	
`test.RData`	The data from the performance test used in the analysis in Section 3.6.
`analysis.R`	R script used for the analysis of the performance test data in Section 3.6.
Files of the control center shiny app	
`server.R`	R script containing the server function.
`ui.R`	R script containing the definition of the UI.
`global.R`	R script containing shared code.
subdirectory `www`	CSS files used by the control center.
subdirectory `modules`	Other R files used by the control center, e. g., the definitions of the classes (see Section 4.2).
Files used to configure Filebeat, and Elasticsearch to process VISITOUR's log file	
`filebeat.yml`	Filebeat configuration (see Section 4.3).
`visitour_monitoring.conf`	Logstash's rules file for handling log events (see Section 4.3).
`logstash.yml`	Logstash configuration (see Section 4.3).

Bibliography

[Apache Software Foundation 2016a] Apache Software Foundation. *Apache JMeter*. Available: http://jmeter.apache.org/. Version 3.1. 2016. (Cited on page 19)

[Apache Software Foundation 2016b] Apache Software Foundation. *Apache Lucene*. Available: http://lucene.apache.org/. Version 6.2. 2016. (Cited on page 19)

[Apache Software Foundation 2017] Apache Software Foundation. *Apache Groovy*. http://groovy-lang.org/. Version 2.4.8. 2017. (Cited on page 20)

[Avritzer et al. 2006] A. Avritzer, A. Bondi, M. Grottke, K. S. Trivedi, and E. J. Weyuker. Performance assurance via software rejuvenation: monitoring, statistics and algorithms. In: *International Conference on Dependable Systems and Networks (DSN'06)*. June 2006, pages 435–444. (Cited on page 2)

[Bielefeld 2012] T. C. Bielefeld. Online performance anomaly detection for large-scale software systems. Received b+m Software & Systems Engineering Award 2012. Diploma thesis. Kiel University, Mar. 2012. URL: http://eprints.uni-kiel.de/15488/. (Cited on pages 2, 6, 7)

[Chandola et al. 2009] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: a survey. *ACM Comput. Surv.* 41.3 (July 2009), 15:1–15:58. URL: http://doi.acm.org/10.1145/1541880.1541882. (Cited on pages 2 and 7)

[Churilin 2013] A. Churilin. Choosing an open-source log management system for small business. Master's thesis. Tallinn University of Technology, 2013. URL: http://de.slideshare.net/fixnix/choosing-an-open-source-log-management-system-for-small-business. (Cited on pages 18, 19)

[Elastic 2016a] Elastic. *Beats platform*. Available: https://www.elastic.co/de/products/beats. Version 5.0. 2016. (Cited on page 18)

[Elastic 2016b] Elastic. *Elasticsearch*. Available: https://www.elastic.co/de/products/elasticsearch. Version 5.0. 2016. (Cited on pages 18, 19)

[Elastic 2016c] Elastic. *Kibana*. Available: https://www.elastic.co/de/products/kibana. Version 5.0. 2016. (Cited on page 18)

[Elastic 2016d] Elastic. *Logstash*. Available: https://www.elastic.co/de/products/logstash. Version 5.0. 2016. (Cited on page 18)

[Frotscher 2013] T. Frotscher. Architecture-based multivariate anomaly detection for software systems. Master thesis. Kiel University, Oct. 2013. URL: http://eprints.uni-kiel.de/21346/. (Cited on pages 2, 6, 7)

[Giesecke et al. 2006] S. Giesecke, M. Rohr, and W. Hasselbring. Software-betriebs-leitstände für unternehmensanwendungslandschaften. In: *Tagungsband Informatik 2006, Band 2*. Volume P-94. Lecture Notes in Informatics. Gesellschaft für Informatik e.V., Oct. 2006, pages 110–117. URL: http://eprints.uni-kiel.de/14543/. (Cited on page 8)

[Henning 2016] S. Henning. Visualization of performance anomalies with kieker. Bachelor's Thesis. Christian Albrechts Universität zu Kiel, Sept. 2016. URL: http://eprints.uni-kiel.de/34141/. (Cited on pages 2, 6, 7, and 50)

[IEEE Standards Board 1990] IEEE Standards Board. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990* (Dec. 1990), pages 1–84. (Cited on page 2)

[Kühnel 2013] J. Kühnel. Centralized and structured log file analysis with open source and free software tools. Bachelor's thesis. Fachhochschule Frankfurt am Main, 2013. URL: http://www.kuehnel.org/bachelor.pdf. (Cited on pages 18, 19)

[Lenstra and Kan 1981] J. K. Lenstra and A. Kan. Complexity of vehicle routing and scheduling problems. *Networks* 11.2 (1981), pages 221–227. (Cited on page 1)

[Li et al. 2007] J.-Q. Li, P. B. Mirchandani, and D. Borenstein. The vehicle rescheduling problem: model and algorithms. *Networks* 50.3 (2007), pages 211–229. URL: http://dx.doi.org/10.1002/net.20199. (Cited on page 1)

[Menasce and Almeida 2001] D. A. Menasce and V. Almeida. *Capacity planning for web services: metrics, models, and methods*. 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2001. (Cited on page 2)

[Microsoft Corporation 2017] Microsoft Corporation. *Microsoft JDBC Driver for SQL Server*. Available: https://www.microsoft.com/en-us/download/details.aspx?id=11774. Version 6.0. 2017. (Cited on page 20)

[R Core Team 2016] R Core Team. *R Language Definition*. Available: https://cran.r-project.org/doc/manuals/R-lang.html. 2016. (Cited on page 21)

[R Foundation 2016] R Foundation. *R*. Available: https://www.r-project.org. Version 3.3.1. 2016. (Cited on page 20)

[Rohr 2015] M. Rohr. Workload-sensitive timing behavior analysis for fault localization in software systems. Doctoral thesis/PhD. Kiel, Germany: Faculty of Engineering, Kiel University, Jan. 2015. URL: http://eprints.uni-kiel.de/27337/. (Cited on pages 5 and 28)

[Rohr et al. 2010] M. Rohr, A. van Hoorn, W. Hasselbring, M. Lübcke, and S. Alekseev. Workload-intensity-sensitive timing behavior analysis for distributed multi-user software systems. In: *Joint WOSP/SIPEW International Conference on Performance Engineering (WOSP/SIPEW '10)*. New York: ACM, Jan. 2010, pages 87–92. URL: http://eprints.uni-kiel.de/14441/. (Cited on pages 2–5, 37, and 61)

[RStudio Inc. 2016] RStudio Inc. *Shiny*. Available: https://shiny.rstudio.com/. Version 0.14.2. 2016. (Cited on page 21)

[Shumway and Stoffer 2011] R. H. Shumway and D. S. Stoffer. *Time series analysis and its applications*. Springer, 2011. (Cited on page 7)

[Singer 2003] J. Singer. Jvm versus clr: a comparative study. In: *Proceedings of the 2Nd International Conference on Principles and Practice of Programming in Java*. PPPJ '03. Kilkenny City, Ireland: Computer Science Press, Inc., 2003, pages 167–169. URL: http://dl.acm.org/citation.cfm?id=957289.957341. (Cited on page 28)

[Sipser 2006] M. Sipser. *Introduction to the theory of computation*. Volume 2. Thomson, 2006. (Cited on page 2)

[Spliet et al. 2014] R. Spliet, A. F. Gabor, and R. Dekker. The vehicle rescheduling problem. *Computers & Operations Research* 43 (2014), pages 129–136. URL: http://www.sciencedirect.com/science/article/pii/S0305054813002670. (Cited on page 1)

[Twitter Inc. 2016] Twitter Inc. *Bootstrap*. Available: http://getbootstrap.com/. Version 3.3.7. 2016. (Cited on page 21)

[Waller et al. 2015] J. Waller, N. C. Ehmke, and W. Hasselbring. Including performance benchmarks into continuous integration to enable devops. *SIGSOFT Softw. Eng. Notes* 40.2 (Apr. 2015), pages 1–4. URL: http://doi.acm.org/10.1145/2735399.2735416. (Cited on page 66)

[Waller and Hasselbring 2013] J. Waller and W. Hasselbring. A benchmark engineering methodology to measure the overhead of application-level monitoring. In: *Proceedings of the Symposium on Software Performance: Joint Kieker/Palladio Days 2013*. CEUR Workshop Proceedings, Nov. 2013, pages 59–68. URL: http://eprints.uni-kiel.de/22326/. (Cited on page 28)

[Wickham 2014] H. Wickham. *Advanced r*. Chapman and Hall, 2014. (Cited on pages 21 and 48)