Master Thesis

Julius-Maximilians-
UNIVERSITÄT
WÜRZBURG

Predicting Performance Degradations of Microservice Applications

Martin Sträßer
Department of Computer Science
Chair of Computer Science II (Software Engineering)

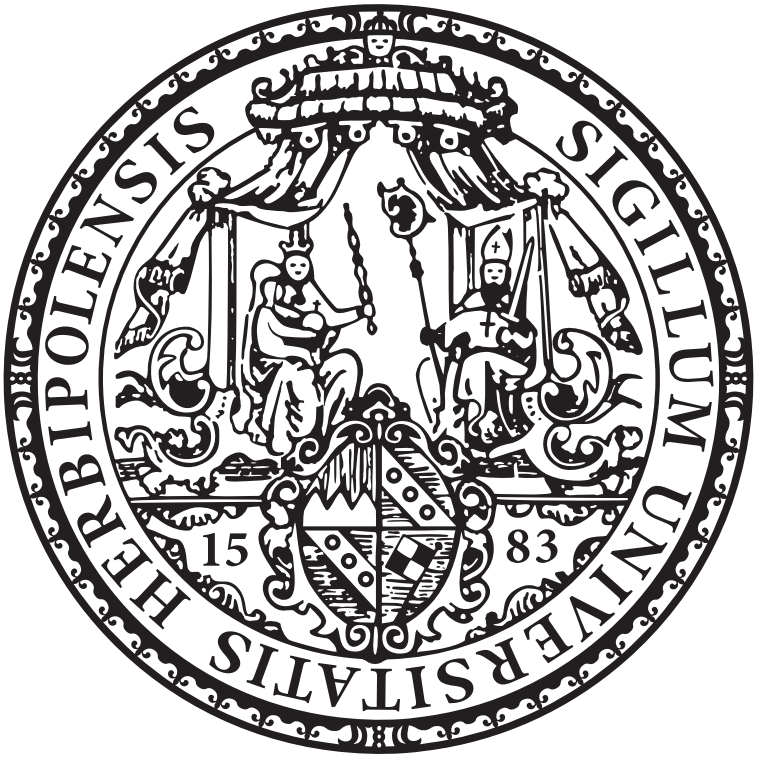
Prof. Dr.-Ing. Samuel Kounev
First Reviewer

Johannes Grohmann M.Sc.
First Advisor

Simon Eismann M.Sc.
Second Advisor

Submission
27. May 2020

www.uni-wuerzburg.de



Ich versichere, die vorliegende Masterarbeit selbstständig und unter ausschließlicher Verwendung der angegebenen Literatur verfasst zu haben. Darüber hinaus versichere ich, die Arbeit bisher oder gleichzeitig keiner anderen Prüfungsbehörde zur Erlangung eines akademischen Grades vorgelegt zu haben.

I hereby declare that the thesis submitted is my own unaided work. All direct or indirect sources used are acknowledged as references. This paper was not previously and is not simultaneously presented to another examination board.

Würzburg, 27. May 2020

.....

(Martin Sträßer)

Abstract

In recent years, a clear trend in the development of large-scale software systems and web applications has evolved. The so-called *microservice architecture* splits a large application in a set of *microservices*, which are clear-scoped and small-scaled programs. By doing this, the development process is simplified and also several benefits in operation and maintenance are created. However, new risks and challenges arise concurrently. One of the main upcoming challenges in this area is the performance evaluation and prediction, especially the prediction of performance degradations. Such declines can cause, for example, increased response times of websites and hence might influence the user experience and lead to violations of service level objectives.

In this thesis, we present a new approach for the prediction of performance degradations in microservice applications. Therefore, we identify the number of incoming requests as one of the most important causes of impaired performance metrics and use modern time series forecasting techniques to estimate the future load. Besides the load intensity itself, the distribution of the requests across the application topology has a big influence on the application performance. In order to understand and model the connections and dependencies between several microservices of one application, we extract architectural information from monitoring and tracing data at runtime. We summarize this information in a *request propagation model*, which is able to predict arrival rates for all microservices and their endpoints. Furthermore, we gather response times and other performance metrics of different microservices at runtime. These data serve as a basis for the training of the *performance inference model*, a machine-learning-based regression model, which calculates a performance prediction from estimated load intensities and architectural information. To enable a comprehensible output of the models, we use a traffic-light system to rate the predicted state of a microservice. In this work, we describe the theoretical aspects, present a generic architecture for realization, and provide a concrete implementation of this approach using open-source software.

In our evaluation, we use realistic workloads and two state-of-the-art microservice test applications. We define and perform measurements in five test scenarios, in which different application states and degradation sources are simulated. The results show that the models are able to learn the performance behavior and architectural dependencies of the applications quickly and without prior knowledge. The predictions are characterized by high accuracies of more than 95%. Depending on the test scenario, up to 72% of the measured performance degradations are predicted correctly. Moreover, the models are able to keep their prediction quality nearly constant even with higher prediction horizons. The advantages of our approach are that it is not application-specific, does not need prior knowledge, produces comprehensible results, and can be extended easily. A weakness of the current version is the strong dependence on the load forecast. This reliance can be reduced in future works by including and modeling more influencing factors for the performance from different sources.

There are two main contributions of this thesis. First, we propose a new abstract model for predicting performance degradations of microservice applications and a generic architecture for its realization. Second, we develop a concrete implementation of this model and perform a detailed evaluation, in which we point out the conceptual strengths and weaknesses of the approach.

Zusammenfassung

In den letzten Jahren hat sich ein klarer Trend in der Entwicklung von großen Softwaresystemen und Webanwendungen entwickelt und etabliert. Die sogenannte *Microservice-Architektur* teilt die Anwendung in verschiedene *Microservices*, kleinere Programme mit klar definierten Funktionen und Verantwortlichkeiten. Dies vereinfacht den Entwicklungsprozess und bietet auch zahlreiche Vorteile im Betrieb und der Wartung der Anwendung. Gleichzeitig entstehen aber auch neue Risiken und Herausforderungen. Eine der zentralen neuen Herausforderungen ist die Leistungsbewertung und -vorhersage der Anwendung, insbesondere die Vorhersage von Leistungseinbrüchen. Solche Einbrüche können sich z.B. in gestiegenen Antwortzeiten einer Webseite ausdrücken und damit Einfluss auf die Kundenzufriedenheit nehmen oder Vertragsverletzungen verursachen.

In dieser Arbeit stellen wir einen neuen Ansatz vor, um Leistungseinbrüche speziell in Microservice-Anwendungen vorherzusagen. Dazu haben wir die anliegende Last auf einem Microservice als einen der Hauptgründe für steigende Antwortzeiten identifiziert und verwenden moderne Methoden der Zeitreihenvorhersage um die Last in zukünftigen Zeitintervallen abzuschätzen. Neben der Lastintensität an sich, spielt auch die Verteilung der Last auf die verschiedenen Komponenten eine Rolle. Um den Zusammenhang zwischen den Microservices innerhalb einer Anwendung zu verstehen, extrahieren wir automatisch, auf Basis von gemessenen und verfolgten Nutzeranfragen, architekturelle Eigenschaften der Anwendung. Diese Informationen fassen wir in einem *request propagation model* zusammen, welches in der Lage ist, eine Vorhersage für die Ankunftsraten für alle Microservices und deren Endpunkte zu erstellen. Weiterhin erfassen wir, zur Laufzeit des Programms, Antwortzeiten und weitere Leistungsmaße von verschiedenen Microservices. Diese Daten dienen als Grundlage für das Training des *performance inference models*, einem Modell, welches auf maschinellem Lernen und Regressionsalgorithmen basiert und aus Lastintensitäten und architekturellen Informationen eine Leistungsvorhersage erstellt. Um eine einfach verständliche Ausgabe der Ergebnisse zu ermöglichen, bewerten wir den Zustand eines Microservices mit einem Ampelsystem. Wir beschreiben die theoretischen Grundlagen des Ansatzes, stellen eine generische Architektur zur Umsetzung vor und erstellen eine konkrete Implementierung mit Schnittstellen zu frei verfügbarer Software.

In unserer Auswertung verwenden wir realitätsnahe Lasten und zwei aktuelle Microservice-Anwendungen zum Testen unseres Ansatzes. Wir definieren und führen Messungen in insgesamt fünf Testszenarien und Experimenten durch, in denen unterschiedliche Zustände der Anwendungen und Fehlerquellen simuliert werden. Die Ergebnisse zeigen, dass unsere Modelle auch ohne Vorwissen in der Lage sind, schnell das Leistungsverhalten und die architekturellen Eigenschaften der Anwendungen zu erlernen. Die Vorhersagen sind gekennzeichnet durch eine hohe Vorhersagegenauigkeit von über 95%. Weiterhin werden je nach Szenario bis zu 72% der Leistungseinbrüche vorhergesagt. Darüber hinaus sind die Modelle in der Lage, auch über größere Zeiträume, Vorhersagen mit annähernd gleichbleibender Qualität zu treffen. Die Stärken unseres Ansatzes liegen vor allem darin, dass er anwendungsunabhängig und ohne Vorwissen angewendet werden kann, erweiterbar ist und nachvollziehbare Resultate erzeugt. Eine Schwäche der bisherigen Version ist die zu

große Abhängigkeit von der Lastvorhersage. Diese kann in zukünftigen Arbeiten reduziert werden, indem noch mehr Leistungsmerkmale der Anwendung und Eigenschaften der Anfragen von verschiedenen Quellen erfasst und modelliert werden. Die Hauptbeiträge dieser Arbeit sind zweigeteilt. Einerseits konzipieren wir ein abstraktes Modell zur Leistungsvorhersage in Microservice-Anwendungen und eine generische Architektur zur Umsetzung des Ansatzes. Auf der anderen Seite stellen wir eine konkrete Implementierung des Modells vor und machen eine detaillierte Auswertung, in der konzeptionelle Stärken und Schwächen des Ansatzes herausgestellt werden.

Contents

1. Introduction

In recent years, a new architectural style for the development of large-scale software systems and web applications has been established successfully. The so-called *microservice architecture* [1][2] uses a set of clear-scoped and small-scaled applications (*microservices*) to realize a larger overall functionality. This is contrary to traditional monolithic applications, which decompose the application into namespaces or packages, classes, and functions only. This centrality is both a curse and a blessing at the same time. One key advantage of a centralized architecture is the fast communication and data exchange between two components as they can be often realized using in-process calls and so happen independently of a network connection or protocol. The main disadvantage is that the whole application including all components has to be installed on one (virtual) machine. This leads to a large maintenance overhead on the one hand, as also small updates need a full rebuild and redeployment, and limited scalability on the other hand, as for a higher load a new (virtual) machine with the complete application has to be provided. In contrast to this, microservices are loosely coupled software parts, which can be developed, deployed and scaled independently.

Nevertheless, this new architectural style introduces also new challenges. One of the most important in practice is thereby the application performance evaluation and prediction, in particular the prediction of performance degradations. Such degradations can influence the user experience and lead to violations of service level objectives. In a centralized architecture, the application typically runs on a dedicated server, where influences of other applications or network parameters can be neglected. In contrast to this, microservice applications are typically deployed on multiple hosts and in diverse environments. This leads to the fact that the performances of different microservices can vary significantly and the performance prediction is impeded.

It is well-known that the load intensity has a huge influence on the application performance. However, in microservice applications, the load distribution across the different microservices and their instances has to be considered in addition to this. For example, if many users request the functionality of one single service at the same time, the performance of this service might decrease significantly, while other services are not fully utilized. This fact has not been targeted extensively in previous research. Moreover, most of the state-of-the-art approaches require prior knowledge or lack explainability of their results, e.g., [28] and [32]. In this work, we propose a new performance prediction model and framework for microservice applications, which tackles these problems. The approach is based on two kinds of models. The *request propagation model* captures the dependencies between

the microservices of one application and predicts the load distribution across the service instances. The *performance inference model* is a machine-learning-based regression model, which generates performance predictions and ratings from load forecasts and architectural information. The approach is extensible, comprehensible, and application-agnostic. Moreover, it does not require prior knowledge and works on monitoring data, which can be gathered by nearly all state-of-the-art application performance management tools. In the evaluation, we use *TrainTicket* [43] and *Teastore* [48], two recent microservice benchmark applications, and design five test scenarios, which simulate different application states and degradation sources.

There are two main contributions of this thesis. First, we propose a new abstract model for predicting performance degradations of microservice applications and a generic architecture for its realization. Second, we develop a concrete implementation of this model and perform a detailed evaluation, in which we point out the conceptual strengths and weaknesses of the approach. The following research questions (RQ) and goals outline the subfields of this work and introduce the most important milestones and aims of this thesis.

Goal 1. Acquire and process performance and tracing data from a microservice application.

RQ1.1. Which performance metrics should be chosen to represent the state of a microservice?

RQ1.2. Which tools can be used for load forecasting and measuring application performance?

RQ1.3. How can the outputs of these tools be integrated into a prediction framework?

Goal 2. Design and implement a novel performance prediction framework for microservice applications, which creates explainable and comprehensible results.

RQ2.1. How can different user behaviors and their influence on the application be modeled?

RQ2.2. How can the performance of single services be assessed using predicted user behaviors and topological information?

RQ2.3. Which output representation to choose, so that the results are readable and interpretable by humans?

Goal 3. Evaluate the framework on a suitable microservice demo application.

RQ3.1. Which microservice demo applications are suitable and available for the evaluation?

RQ3.2. Which metrics can be used for evaluating the proposed framework?

RQ3.3. What are the possible test scenarios and user workflows for the evaluation?

RQ3.4. Which influences have forecasting and monitoring intervals on the prediction power?

Goal 4. Evaluate the framework on a second application and compare the results.

RQ4.1. Which changes are needed to apply the framework to another application?

RQ4.2. How does the complexity of the application influence the prediction power?

RQ4.3. What are the strengths and weaknesses of the approach and how could its performance be increased in future work?

The remainder of this thesis is structured as follows. In Section 2, we describe the foundations of this work and introduce important terms and the deployed tools. In Section 3, we analyze related publications and name similarities and differences between these papers and our work. Section 4 describes the conceptual and theoretical view of our approach, while a concrete implementation is introduced in Section 5. In Section 6, we explain the methodology of our evaluation and the technical setup. Section 7 presents and discusses the results of the evaluation in detail. Finally, Section 8 completes this work and draws conclusions.

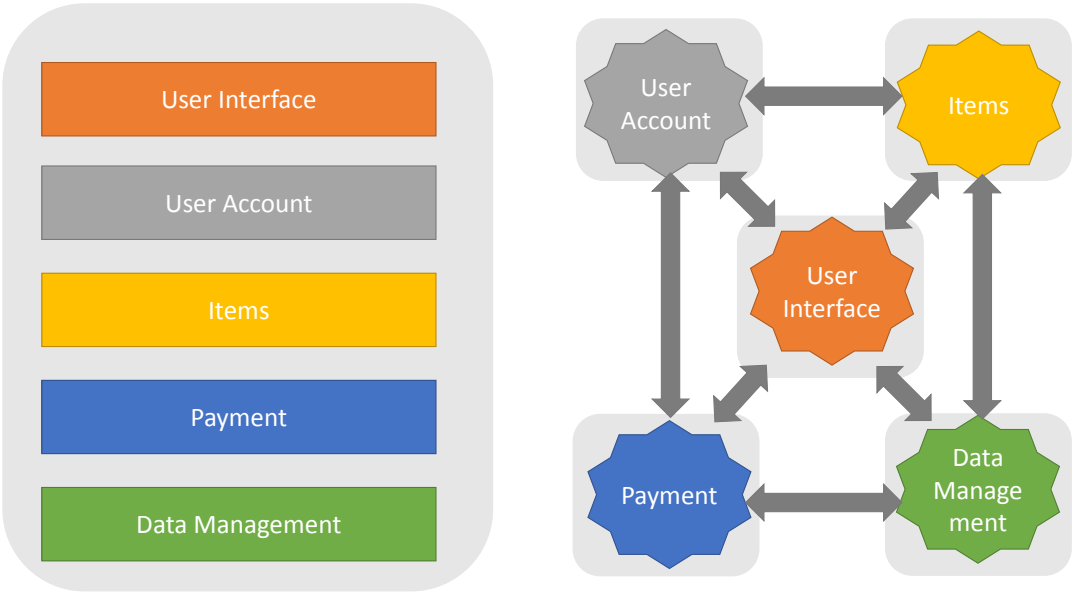
2. Foundations

In this chapter, we describe the foundations and basic knowledge required to understand the remainder of this thesis. Therefore, Section 2.1 describes background information on microservices, their characteristics, and deployment. Section 2.2 presents the most important terms and algorithms in the fields of machine learning. Section 2.3 wraps up this chapter and describes the open-source tools used in this work.

2.1. Microservices and Containers for Application-Level Virtualization

Large-scale applications with a monolithic architecture are characterized by large maintenance overheads and limited scalability. The microservice architecture overcomes these mitigations, as the services can be deployed independently of each other. The communication between the components is often realized via lightweight HTTP resource APIs, e.g., Representational State Transfer (REST). As a consequence of this, the whole application is not restricted to a single implementation technology, which for example means, that the services can be developed using different programming languages. On the one hand, this leads to a more flexible and faster development, as activities can be parallelized easily. On the other hand, simpler maintenance is achieved as well as increased interchangeability and failure resistance compared to monolithic applications. Moreover, microservices are better scalable in large-scale and cloud applications. In particular, the services can be scaled independently of each other. Figure 2.1 shows a simple online shop, which consists of several components, e.g., for payment and data management. If we choose the microservice architecture shown on the right side, we could start new instances of the user account service, as we observe or predict many users who need to log in or register. If we choose the monolithic architecture, we would need to replicate the whole application for scaling. All in all, microservices should have the following properties [2]:

- The services provide and require well-defined interfaces.
- Their internal behavior can be treated as a black-box.
- They are independently deployable and independent of implementation technologies.
- The services contain all their dependencies.
- They are stateless as best practice.



(a) Monolithic Architecture (b) Microservice Architecture

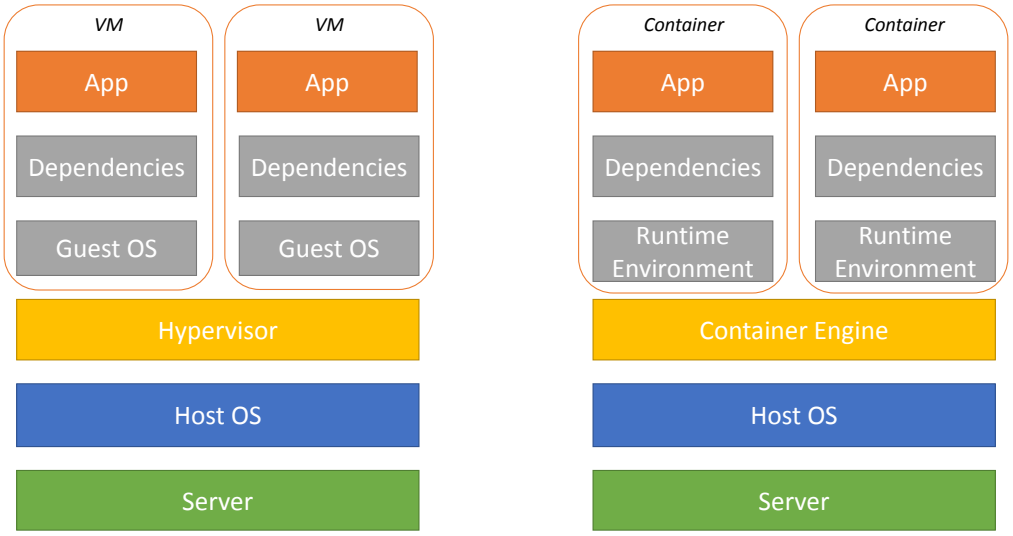
Figure 2.1.: Different Online Shop Architectures

The characteristics of microservices automatically lead to the question of how microservices can be deployed. Following the argumentation from above, they need a lightweight deployment. Traditional virtual machines have a lot of overhead, e.g., for configuration and start-up. They usually use a guest operating system (OS) with a hypervisor for virtualization. In contrast to this, application-level virtualization is based on one OS kernel, which is shared among different applications or *containers*, as Figure 2.2 shows. By doing this, the overhead is reduced.

A container normally comprises a runtime environment (RE), all binaries, as well as all dependencies or libraries. For a lightweight integration of these containers in the development process, two kinds of tools are required. First, a build automation and dependency management system, e.g., Apache Maven[1] or Gradle[2], is needed, as these tasks are non-trivial for real-world applications. Second, a container system, e.g., Linux Containers (LXC)[3] or Docker[4], has to define how to create, start, and manage containerized applications. Additionally, in large-scale applications, a container management software, e.g., Kubernetes[5] or Docker Swarm[6], can be used, e.g., for scaling tasks as well as starting and stopping larger groups of containers.

In this work, we use Docker containers for application virtualization and provide templates for Docker Compose for a deployment on a single machine and Docker Swarm for a deployment on multiple hosts. A Swarm cluster consists of several computing nodes, which can be either managers or workers. After the initialization of the cluster by a manager, nodes can join or leave the cluster easily. If we start an application with different services on a manager node, the service instances are split across the nodes in the swarm automatically. It is possible to set several options related to deployment and runtime via configuration files, such as deployment constraints, service replicas, and restart policies. Another impor-

[1] https://maven.apache.org/
[2] https://gradle.org/
[3] https://linuxcontainers.org/
[4] https://www.docker.com/
[5] https://kubernetes.io/
[6] https://docs.docker.com/engine/swarm/



(a) Virtual machines (b) Containers

Figure 2.2.: Different Deployment Options

tant feature of Docker is the possibility to set resource constraints for deployed services. Thereby, the CPU and RAM usage of containers can be limited. REST requests are routed and processed using an internal domain name system (DNS) server.

2.2. Machine Learning

In this section, we describe background information on machine learning. In the first paragraph, we provide an overview of important terms in this field and describe regression problems in a more detailed way. In the second paragraph, we introduce selected machine learning algorithms and describe their characteristics.

2.2.1. Overview

Machine Learning (ML) [4] is a subset of artificial intelligence and is widely used in many research fields. For prediction tasks as in our work, machine learning algorithms try to predict an output y from a given input x, where x and y can be both one- or multi-dimensional quantities. The function or dependency between x and y is learned during the training phase. There are three different forms of machine learning. In *supervised learning*, a set of input values with their corresponding correct output values is provided during the training phase. In contrast to this, *unsupervised learning* techniques do not provide reference outputs during training. Finally, *reinforcement learning* (RL) aims to determine the best action to execute in a given scenario. The quality of an action is rated by a reward function. An RL algorithm learns suitable actions for each situation via trial and error and does not use predefined training examples in general. Additionally, machine learning problems can be categorized by the form of output or solutions to calculate. An ML problem, which has a finite number of discrete solutions or solution classes, is called a *classification problem*. In contrast to this, a problem with continuous numeric solution values is called a *regression problem*. In this work, we use regression algorithms to predict continuous performance metrics of microservices. As a consequence, we describe fundamental concepts of regression problems in the following paragraph in a more detailed way.

In a multi-dimensional regression problem, which is solved via supervised learning techniques, a set of input vectors $x \in \mathbb{R}^m$ and corresponding output vectors $y \in \mathbb{R}^n$ is used as the training set. The aim of the training is, to learn the relationship $y = f(x) : \mathbb{R}^m \rightarrow \mathbb{R}^n$. Thereby, the type of the function $f(x)$ can be unknown or restricted in advance. For example, in linear regression problems, f has to be a linear function. After the training, the output of a previously unseen input is predicted by using the learned dependency. However, in real-world applications, there is a statistical prediction uncertainty. One reason for this is that the learned dependency function f is usually an approximation of the real dependency between x and y. Additional uncertainty is added, if x contains measured or predicted values itself and is therefore subject to measurement or prediction errors respectively.

A major challenge of ML-based prediction algorithms is *generalization*, which refers to the ability to predict (nearly) correct outputs across a large range of (previously unseen) inputs. If the algorithm models the (noisy) training inputs too well, the prediction quality for new input values decreases. This problem is called *overfitting*. In contrast to this, *underfitting* means that the algorithm performs badly on both training and test data. Underfitting normally requires the change of the algorithms hyperparameters or the use of a different approach.

2.2.2. Selected Algorithms

There are a lot of ML algorithms, which are suitable for both classification and regression problems. These include linear models, neural networks, (sparse) kernel methods, graphical models, and mixtures [4]. In the following, the discuss four algorithms, which are often used in the analyzed literature for prediction tasks, and therefore interesting for our approach, in a more detailed way.

The *k-nearest neighbors algorithm* is one of the most comprehensible machine learning algorithms and has shown effectiveness in many problem scenarios. Given a set of training vectors $x_1, x_2, ..., x_n$ and their outputs $y_1, y_2, ..., y_n$, the output y_{n+1} of a new input vector x_{n+1} needs to be predicted. Therefore the k nearest points in the neighborhood of x_{n+1} are selected. The distance between two input vectors can be computed using a norm, e.g., the euclidean norm. k determines the number of neighbors to be consulted for the prediction and must be set appropriately. A low value of k means that the predicted output is strongly geared to the training set or historical data. Contrary to this, a high value of k means that also points with a higher distance to x_{n+1} influence the prediction. From all selected neighbors of x_{n+1}, the values of the corresponding outputs are averaged, which creates the prediction for the new input. The advantage of this procedure is that it produces explainable results, as it uses a simple type of learning from the past. Moreover, the algorithm performs well on balanced data. However, it is obvious that the quality of the result depends strongly on the choice of the hyperparameter k. Furthermore, the efficiency and speed of the algorithm decrease significantly with large and multi-dimensional datasets.

The *Naive Bayes algorithm* is widely used for classification tasks and can be applied to regression problems as well [5]. The algorithm is based on a conditional probability model. Given an input vector $x = (x_1, x_2, ..., x_n)$, an output y is chosen, which maximizes the conditional probability $P(y|x_1, x_2, ..., x_n)$. This decision technique is called *maximum a posteriori (MAP)* decision rule. The Naive Bayes approach assumes, that all n entries (features) of the input vector x are statistically independent variables. This assumption does not often hold in real-world applications but is sometimes a sufficient approximation. With this assumption the conditional probability $P(y|x_1, x_2, ..., x_n)$ can be written as shown in Equation 2.1, where α denotes a scaling factor.

$$P(y|x_1, x_2, ..., x_n) = \frac{1}{\alpha} \cdot P(y) \cdot \prod_{i=1}^{n} P(x_i|y) \tag{2.1}$$

Naive Bayes is an ML algorithm, which works best with supervised learning, as the probabilities $P(x_i|y)$ can be learned during training. One of the advantages of this approach is that in classification problems the correct output is chosen, whenever the correct class is more probable than all other classes and even if the independence assumption is wrong [6]. However, it is shown, that Naive Bayes has performance problems when dealing with continuous input values and unbalanced outputs [7][8]. Based on the Naive Bayes approach, many advanced regression techniques evolved, like the Bayesian linear regression.

Decision trees [4] aim to combine various prediction models and can be used for both classification and regression problems. The value range of the input vector x is split into several regions and, depending on the actual value of x at a certain time, the corresponding model is selected for predicting the output. For example, in a regression problem, we can assign a constant output value or vector to a specific region of the input values. This approach is easy to understand by humans, as one can determine the traversal through a decision tree for specific input. However, a single decision tree is sensitive to its training data and a small change of these results in different trees and therefore different predictions. The *Random Forest algorithm* [9] tries to prevent this overfitting by combining the predictions of multiple decision trees. Therefore, a single tree makes its decision based on m randomly selected features out of the n total features in the input vector. After the evaluation of all trees, the final output of the Random Forest algorithm is calculated by averaging the outputs of every single tree. All in all, this procedure leads to a powerful and high accuracy mechanism and the randomization elements prevent overfitting.

Another wide-spread approach for classification tasks are *Support Vector Machines* (SVM) [10]. If SVMs are applied to regression problems, we speak of *Support Vector Regression* (SVR) [11]. The basic aim of SVMs is to separate the input data into different classes using hyperplanes, such that the distance (margin) from the closest data point to the separating plane is maximized. An easy example is to split a two-dimensional dataset into two classes by choosing that linear function which maximizes the margin. If the dataset is not linearly separable, the input space is transformed into a higher-dimensional feature space using a kernel function. This procedure is called the *kernel trick*. Popular mappings are linear or polynomial functions as well as radial basis functions [12]. To choose the best separating hyperplane, the SVM has to solve an optimization problem, usually by utilizing computationally efficient quadratic programming algorithms. SVMs are able to find guaranteed optimal solutions, as the optimization problem is of convex nature. Moreover, SVMs are implemented in nearly every ML framework and offer many extensions such as transductive SVMs for semi-supervised learning.

2.3. Tools

In the following, we describe the open-source tools used in this work as they play a major role in the evaluation process. First, Section 2.3.1 introduces the Pinpoint monitoring tool, which we use to gather monitoring data from our microservice applications. Second, Section 2.3.2 describes the HTTP load generator, while the time series forecasting library GluonTS is introduced in Section 2.3.3.

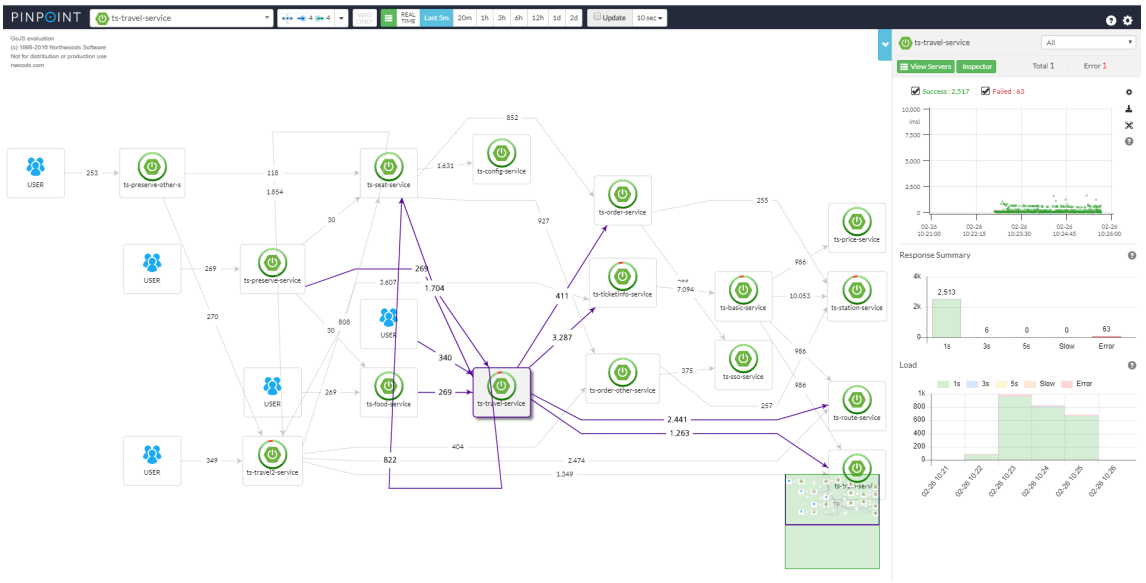


Figure 2.3.: Server Map of Pinpoint

2.3.1. Pinpoint Monitoring

Pinpoint [13] is an open-source application performance management (APM) tool for monitoring applications written in Java or PHP[7]. The gathered data make it possible to obtain the application topology as well as request metrics and call stacks. For Java applications, the binary needs to be started with the Pinpoint agent attached using the `javaagent` flag. The source code of the application itself does not need to be modified, as the Pinpoint agent uses bytecode instrumentation for monitoring. The agents' settings can be adjusted using a configuration file, which specifies sampling rates, monitored metrics, and more. The agent and service use custom names for identification. All agents send their data to the Pinpoint Collector, which processes the data and saves them into an Apache Hbase[8] database. Therefore, the IP address of the machine hosting the Collector needs to be specified in the configuration file of each agent. The Pinpoint Web UI finally reads and visualizes the collected data.

Figure 2.3 shows the starting screen of the Pinpoint Web UI. This view delivers a rough overview of all collected data. In the left part of the picture, the server map is displayed, which visualizes the instrumented microservices and the connections between them. The label of each edge in the graph represents the number of requests sent in the last interval from the origin to the destination node. The interval length can be adjusted with the control button on top of the server map, in this case, the requests of the last five minutes are displayed. In the right part of the picture, an overview of the performance data for a selected service is shown. Moreover, one can obtain the chronological sequence of response times and occurred exceptions. Besides of that server map view, Pinpoint provides the transaction view, where metrics and call stacks of single requests can be queried and analyzed. Furthermore, Pinpoint collects monitoring data from the JVMs, such as heap usage. Beyond the graphical user interface, Pinpoint provides the Web API, where all the described data can be queried as JSON objects. Additionally, the functionality of Pinpoint can be extended by adding further endpoints as plugins. We use Pinpoint and the Web API in particular to gather and query the monitoring data of Java microservices. We describe this process as well as the transaction view more detailed in Chapter 5.

[7]This section refers to Pinpoint version 1.8.4, which is the release used in this work.
[8]`https://hbase.apache.org/`

2.3.2. HTTP Load Generator

HTTP Load Generator [14] is a tool for generating HTTP requests to simulate varying load intensities. Hence, it can be used for testing web or microservice applications. The generator itself is a Java application, which can be used in two different modes. In the `loadgenerator` mode, the application generates and sends new HTTP requests to a specific target. In the `director` mode, multiple load generator instances can be coordinated. The director sends instructions to a dedicated port where the generators listen to. Therefore, the director needs the IP addresses of every generator at startup time. As an additional parameter, the load generator receives a CSV file, which contains the desired load intensity over time and hence specifies the duration of the experiment as well. Moreover, the number of threads that are used for load generation can be specified. Thereby, each thread can be interpreted as a single user that has its own usage context with respect to the test application. As a result, a user pool with a flexible size evolves. For each thread, the associated cookies are saved and reused for each request, which is important, for example when testing applications with a login mechanism.

By default, the director would assign the task to generate a new request to every thread in turns. This behavior can be changed by setting the `randomize-users` flag. If this option is chosen, the thread, which generates the next request, is picked pseudo-randomly from the user pool. On the one hand, this keeps the load reproducible and, on the other hand, creates some entropy in the types of incoming requests at the test application. The load generator works with synchronous communication within one user context, which enables the possibility to perform a certain request based on the response of a previous one. In order to prevent deadlocks in cases of overloads or service failures, a request timeout can be set at which requests are dropped. Moreover, the load generator produces configurable logging output and a statistical summary, where successful and failed requests are listed.

One of the most important parameters for the load generator is the request definition script, which specifies the targets and payloads for the requests. This LUA script typically contains the base URL (e.g., `http://myapp.test`) and the two routines `onCycle` and `onCall`. The behavior of a single user is modeled as a cycle. We define n states or request types and traverse them in a fixed order for every user. After the traversal, the cycle is restarted automatically. The routine `onCycle` is called whenever a new cycle is started and typically sets or resets all variables needed in the next traversal. The routine `onCall` computes the next URL and HTTP request type (`GET` or `POST`) for the next request to be sent. As a part of this work, we added support for specifying JSON payloads in the request definition script, which can be sent when performing a `POST` request. As a parameter, `onCall` receives the current call number, which is a positive integer. As an example, one can imagine a simple load script for a website, where the first call performs the user login (e.g., a `POST` request to `http://myapp.test/login`) and the second call is used to display shopping items (e.g., a `GET` request to `http://myapp.test/items`). After a user has performed these two requests, his cycle is restarted. We use the HTTP load generator in our evaluation to generate requests for our test applications. A detailed description of the load scenarios is given in Section 6.2.

2.3.3. AWS GluonTS Time Series Forecasting

GluonTS [15] is a Python library for time series modeling and forecasting published by Amazon Web Services (AWS) in 2019. It is focused on deep-learning-based models but provides probabilistic models and components as well. Moreover, functions and tools for datasets and plotting are included. GluonTS is built around the deep learning framework Apache MXNet[9] and is available as open-source software[10]. We use estimators from the

[9] `https://mxnet.apache.org/`

[10] Download available at: `https://github.com/awslabs/gluon-ts`

GluonTS library in this work as load and performance forecasters. GluonTS is a state-of-the-art time series library and has been used in other previous studies as well, e.g., [16] and [17]. It offers powerful models and tools for time series forecasting and is easy-to-use. We describe the used estimators, their settings, and performance later in this work.

3. Related Work

As mentioned earlier, our aim in this work is to develop an online performance degradation prediction system based on machine learning for applications composed of microservices. As a consequence, the related work splits into several groups as pictured in Figure 3.1. In Section 3.1, we describe earlier approaches and findings for performance or failure prediction in component-based software systems, which use machine learning techniques. In Section 3.2, we analogously describe related work in the fields of online failure prediction, whereas Section 3.3 reviews publications, which target performance prediction for applications composed of microservices. Section 3.4 summarizes publications, which describe ways to model user behavior and include these models for performance estimation or prediction. Finally, Section 3.5 describes some microservice test applications, which are used for the evaluation of several performance algorithms.

3.1. Machine Learning for Performance Prediction in Component-Based Software Systems

Becker et al. [18] name seven general requirements for performance prediction methods of component-based software: accuracy, adaptability, compositionality, scalability, analyzability, cost-effectiveness, and universality. The use of machine learning is a possibility to investigate and learn complex dependencies, especially in case they are unknown at design time. As a consequence, machine learning is a prediction technique, which is both adaptable and universal, as it can be used for different applications. This universality is also an important requirement for our work, whereas scalability and cost-effectiveness play minor roles in the conception phase.

Matsunaga and Fortes [19] evaluate the applicability of multiple machine learning algorithms on the prediction of resource consumption by different applications. More precisely, the authors use classification trees, support vector machines, and the k-nearest neighbors algorithm to predict execution times, disk, and memory consumptions. It is shown, that all of the mentioned approaches deliver competitive results on the evaluated applications. Moreover, it is concluded, that as much information or attributes as possible should be used as inputs for the ML algorithms if sufficient computing resources are available. This leads to better results and less relevant information will have a minor influence on the prediction automatically.

Andrzejak and Silva [20] use different classification algorithms to predict performance degradation in SOA applications caused by software aging. Similar to the work by Mat-

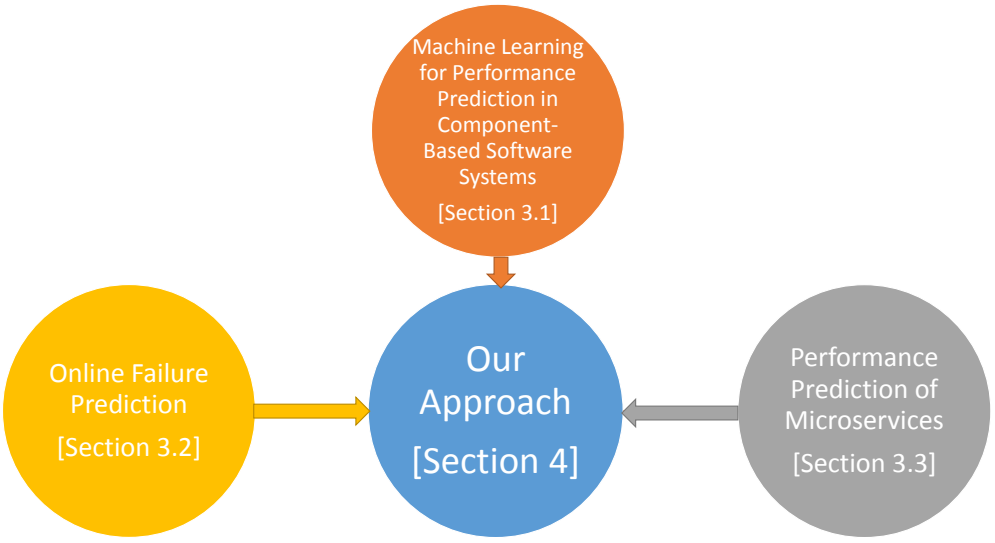


Figure 3.1.: Research Fields Influencing Our Approach

sunaga et al. mentioned above, it is shown that decision trees, Bayesian methods, and support vector machines can be applied to this problem and produce statistically significantly good results. The authors point out that competitive results can be archived even with small training sets and affordable computational costs. Hassan et al. [21] provide a taxonomy for ML algorithms and some statistical estimators. Moreover, they list important evaluation metrics in the fields of performance prediction. Bianchini et al. [22] point out that machine learning techniques are a promising and powerful approach in performance and resource management tasks in modern cloud systems.

3.2. Online Application Failure Prediction

Grohmann et al. [23] propose a taxonomy for the prediction of service level objective (SLO) failures. They divide papers in this research area up by the prediction target, time horizon, and applied modeling type. Salfner et al. [24] summarize findings in the fields of online prediction of software failures in their review. A taxonomy on online failure prediction methods as well as an overview of common metrics is provided. They characterize the aim of online failure prediction as the inference of possible failures in the near future based on recent measurement data from the system at runtime. In contrast to this, the root cause analysis tries to identify the reason for a failure, which occurred in the past. The authors define the term proactive fault management as the process of gathering measurement data, predicting future failures, and performing preventive actions. This last step is not targeted in this work, as we still assume a human in the work chain, which decides on possible actions. However, our proposed output format provides indications, which action could prevent performance degradations.

HORA [25] is an online software failure prediction tool, which combines traditional failure predictors and software architecture models. As a result, the algorithm is able to approach failure propagation throughout the application. Therefore, two different types of models are used: an architecture dependency model and a failure propagation model. Time series forecasting is utilized to predict the breakdown of a component. As a result, the failure probabilities of each component are calculated. To deduce the influence of possible component failures on other components, Bayesian inference and tables are used. It is

shown, that this approach creates a higher true positives rate than mechanisms without architectural knowledge. However, the use of Bayesian tables is a simplified model to failure propagation, which is not practical in production environments as it cannot model the influence of replicas, different deployments, circuit breakers, and similar accurately. In contrast to this, our approach is resistant to such factors.

Zhang et al. [26] present *OPred*, an online performance predictor for service-oriented systems. They divide their approach into four steps. In the first step, performance monitoring, useful performance data are acquired from all relevant nodes and services. Second, in the records sharing phase, all obtained data are collected in a performance center, which so gains global knowledge about the system and its components. In the third step, the service failure prediction, an offline evolutionary algorithm and an online incremental algorithm are combined to learn a time-aware latent feature model. In the last step, the system performance prediction, the overall system performance is calculated by combining information about the service compositions and the response times of each service. Therefore, different formulas are used for simple sequences of services, branches, loops, or similar. It is shown, that OPred is able to generate good results evaluated in a realistic online shopping scenario. In contrast to this mechanism, our approach is able to work completely online and uses machine learning techniques for performance prediction. Moreover, we use explicit models for both request and failure propagation.

3.3. Performance Prediction of Microservices

Bao et al. [27] provide a performance model and description for microservices. The authors model the overall request processing time of a microservice as a composition of three parts: the request caching time, the execution time of the business process triggered by the request, and the processing time of transactions to a backend database or of I/O operations. Moreover, hardware features, such as the available RAM or CPU cores, are identified as additional influencing factors for the microservice performance. Jindal et al. [28] present the tool *Terminus*, which analyzes the resource consumption of microservices and tries to both maximize the overall application performance and minimize the resource consumption by changing the deployment configuration. The authors introduce the metric *microservice capacity*, which represents the maximum number of requests, which a microservice can process without violating any service-level objectives (SLO). To evaluate the performance of a single service, a sandboxing strategy is used. Therefore, calls initiated by the evaluated service are interrupted and redirected to dummy services, which respond in near-constant time. In contrast to this, in our approach, the performance of one single service does not have to be measured under special conditions. Similar to our work, the performance is predicted using a regression model, the Theil-Sen estimator [29][30].

Du et al. [31] design an anomaly detection system for container-based microservices. Therefore, realtime performance data are acquired using monitoring agents and sent to a data processing module that uses supervised machine learning for anomaly detection. Different ML algorithms, such as k-nearest neighbors, support vector machines, Naive Bayes, and random forest are tested. During the training phase, a load generator simulates user requests and, after labeling the acquired performance data, a classification model is created. To simulate realistic anomalies in the system, a fault injection module is used, which can create high CPU consumption, memory leak, network package loss, or high network latency. After evaluation of a virtual IP multimedia subsystem, the authors conclude that all algorithms except support vector machines perform equally well and the k-nearest neighbors algorithm is especially well suited in scenarios with only one service.

Seer [32] is an online cloud performance debugging system that uses deep learning for performance prediction. Performance data of deployed microservices are acquired through

RPC-level and distributed tracing. Seer is trained to recognize communication patterns between microservices and patterns over time, e.g., resource consumption or the number of requests. The basic principle is that the performance in the near future can be inferred using old data and deep learning. The architecture of the neural network contains both convolutional and recurrent layers. The authors evaluate their approach both on the DeathStarBench benchmark suite (see Section 3.5) and a real-world cloud application. They were able to archive high accuracies with their approach in both scenarios. In contrast to Seer, our mechanism does not use deep neural networks as the results are less explainable.

Microscope [33] uses causality graphs to identify root causes for failures in microservice applications. First, network parameters and SLO metrics for each service are measured and processed by a monitoring unit. In the dependency graph, two types of interactions are defined: communicating service dependency and non-communicating dependency. Communicating service dependency means that one service directly sends requests to another. In contrast to this, non-communicating service dependency means that the performance of one service influences another service even if they do not exchange messages with each other. An example of a non-communicating dependency are two services, which share common resources, e.g., in case they are deployed on the same physical host. To build the dependency graph, the authors utilize a parallelized version of the PC algorithm [34].

Whenever a service is ranked as abnormal, the cause inference algorithm is triggered. This algorithm is based on traversals across the dependency graph. For example, a service is identified as a root cause candidate, if all of its neighbors are still rated as normal. To sort all the candidates and identify the most probable root cause, the correlation between the SLO metrics of the frontend and the candidate service is calculated. The authors evaluate their approach on the SockShop benchmark (see Section 3.5) and are able to archive both high precision and recall. Other root case localization algorithms for microservices based on call or dependency graphs are [35] and [36]. In contrast to these works, we utilize load forecasting to predict future performance degradations. We model the dependency between two services as functions and approximate these dependencies using regression algorithms. We explicitly consider the number of incoming requests to a microservice as a causal trigger for performance degradations. Possible root causes are implicitly pinpointed in our output representation. An explicit algorithm can be integrated as an extension as well as possible performance degradations caused by co-located services.

3.4. Modeling User Behavior for Software Performance Estimation

Modeling and prediction of user behavior have always been a field of interest for web service researchers and many companies both for economical and technical reasons. A popular way to model user behavior are so-called Customer Behavior Model Graphs (CBMG) [37]. These contain different states of a customer as nodes and each edge between two nodes is associated with a transition probability between two states. The most common application of CBMGs is the prediction of the next request a user performs on a website. This information can then be further used for workload predictions and performance estimation of web servers. It has been shown that such graphs can be analyzed efficiently using Markov chains [38]. Almeida et al. [39] point out that CBMGs are indeed suitable as a workload model for web services and can be combined with queueing networks (QN) for performance prediction.

Roy et al. [40] use such a combination of CBMGs and QNs for a capacity planning process of component-based software systems. In a later work [41], this approach has been

extended to autoscaling applications in cloud environments. In contrast to these works, we do not use Customer Behavior Model Graphs or queueing networks. Nevertheless, the graphical representation of our request propagation model introduced in Chapter 4 looks fairly similar to those of the CBMGs. However, CBMGs model the user behavior in order to predict future user requests while our approach tries to model the communication and requests between different microservices of one application. For our evaluation, we defined a fixed sequence of requests every user performs. However, in future work, CBMGs might be included to design more realistic workloads for evaluation on the one hand, and to improve the workload forecasting on the other hand.

3.5. Microservice Demo Applications

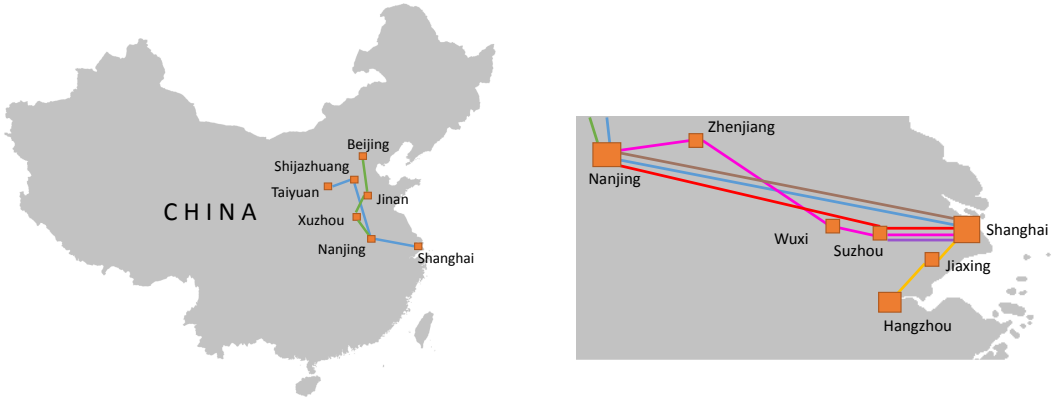
In order to evaluate algorithms and mechanisms for performance prediction in the fields of microservice applications, researchers need demo applications or benchmarks. Industrial software is often closed-source and therefore not available to researchers as well as often too complex for the actual aim. Demo applications should replicate the behavior of real-world software as closely as possible on the one hand, but on the other hand should be comprehensible, observable, and simpler than software in production environments. Aderaldo et al. [42] point out twelve requirements for microservice benchmarks classified into three categories. Moreover, the authors evaluate the conformity of four demo applications to these requirements. In the following, we describe TrainTicket [43], which is the main benchmark used in this work to evaluate our approach. Afterward, we describe other relevant microservice demo applications from the literature.

3.5.1. TrainTicket

Overview. TrainTicket[1] is a microservice-based application, which simulates the website of a railway company. The system contains two different roles: administrator and customer. The administrator is able to configure and manage user accounts, bookings, train connections, prices, and much more. To access the management interface, the user has to login through a dedicated webpage. Customers can search for train services, buy tickets, and manage their bookings. All in all, a complete workflow from searching and buying train tickets to check-in at the railway station can be simulated. Pre-configured user accounts, train services, and bookings are created at application start-up. The default track network is shown in Figure 3.2 and can be extended with additional connections and stations by the administrator. TrainTicket also contains some built-in tools for tracing individual requests.

Architecture and Implementation Technologies. The TrainTicket application consists of 42 microservices. 38 of them are implemented in Java using the Spring Framework[2] and 21 use MongoDB[3] databases as data storages. The remaining services are implemented in Python, Node.js, and Go. The user interface is provided by an NGINX[4] web server. The services communicate using HTTP requests and REST endpoints. Figure 3.3 shows the connections between all services obtained by a source code analysis as a graph. Each node represents one service, an edge from service A to service B means that A sends requests to B. Blue nodes represent Java services, which can be instrumented using Pinpoint, while

[1] We use the version dated 30 Aug 2019, source code available at [44]
[2] `https://spring.io/`
[3] `https://www.mongodb.com/`
[4] `https://www.nginx.com/`



(a) Interregional Trains

(b) Local Trains in the Shanghai-Nanjing Region

Figure 3.2.: Pre-Configured Rail Network of the TrainTicket Demo Application

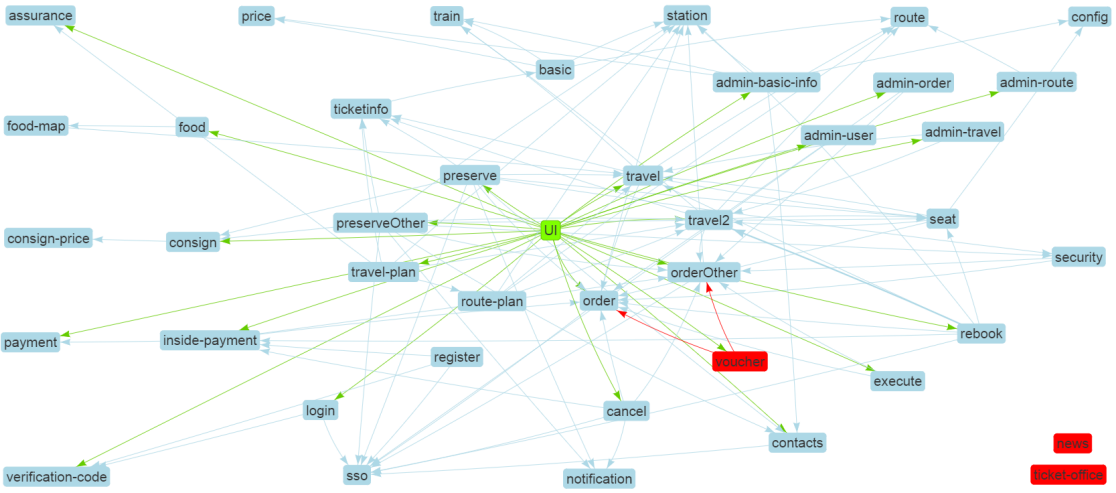


Figure 3.3.: TrainTicket Application Topology

red services are implemented in other languages. The user interface and its calls[5] are displayed in green. On the one hand, we observe a strong linkage between the components. On the other hand, we identify different roles within the application. For example, while the verification code service can be seen as a backend service, as it is only called by other services and does not depend on other instances, the rebook service is only called from the web UI and depends on many other services. In the application version used in this work, both the news and the ticket office service are not connected to the core application but can be called via HTTP requests.

Deployment Variants. The built microservice executables run inside Docker containers. TrainTicket already delivers default Docker files for each service. At runtime, the webserver, all services, and all connected databases run inside dedicated container instances. The user can choose between a local deployment on one machine using Docker Compose and a distributed deployment. Therefore, configuration files for Docker Swarm and Kubernetes as container orchestrators are provided. We describe our evaluation setup and deployment in Section 6.1.

[5]Note, that only calls initiated either by the default or admin UI are shown. Calls from test or benchmarking interfaces are omitted.

Discussion. We use TrainTicket as our main demo application, as it has beneficial characteristics for our use case. First, the application is designed for distributed deployment and provides different blueprint files, which reduce the configuration effort. Second, TrainTicket is a comprehensible application, where each service has a clear scope and responsibility. It uses modern implementation technologies and is still actively developed and supported. As the vast majority of services are implemented in Java, Pinpoint is the only monitoring tool we have to use to gather data from almost the whole application. The extent of the application with more than 40 business logic services is the largest in the analyzed literature and offers a lot of possibilities for our evaluation. Nevertheless, we can vary the complexity at any time by executing different call chains, as each service can be called via REST separately. By doing this, we can start analyzing single services and then enlarge our scope. TrainTicket has been used for different studies before, for example in the fields of model-driven engineering [45] and fault analysis and debugging [46][47].

3.5.2. Other Demo Applications

Since the trend of using microservices has emerged, open-source demo applications have been developed. However, these programs usually function rather as technology demonstrators than as benchmarks for performance evaluations. These include inter alia IBM's *Acme Air*[6] and *Spring Cloud Demo* applications[7]. *SockShop*[8] is a microservice demo application, which has been used in many previous studies, also for performance investigations [33]. Similar to TrainTicket, SockShop supports also different container orchestration tools and provides pre-built Docker images. Nonetheless, all of the mentioned applications have in common, that they consist of only a few microservices. In the following, we describe two recent reference applications in a more detailed manner.

Teastore [48] is an open-source microservice benchmark, which has been developed for performance studies explicitly. It consists of different services, where each service has own characteristics related to its performance. Similar to SockShop, it simulates an online shop, where a customer can buy tea and related equipment. The authors show the usability of TeaStore in three different contexts: performance modeling, cloud resource management, and energy efficiency analysis. The application supports container orchestration tools and contains built-in tools for performance studies and pre-built container images. The main difference to TrainTicket is the complexity of the business logic. Compared with more than 40 TrainTicket services, TeaStore consists of only five components. However, the lower number of services makes it easier to replicate the whole application, which is well suited for scaling and load balancing studies. We use the TeaStore for a minor part of our evaluation studies. We describe the setup and application in Section 6.2 further.

DeathStarBench [49] is a benchmark suite with multiple microservice reference applications. All of them are characterized by a modular and extensible design. The authors state that one of the main goals of DeathStarBench is to analyze the effects of microservices on different levels, from hardware implications to the application as a whole and scalability issues. Similar to TrainTicket, the services are developed using different programming languages and open-source frameworks and provide interfaces for tracing tools. All in all, the topologies and characteristics of five applications are described, from which two are available as source code yet. In contrast to TrainTicket, the implementations and frameworks are more diversified, as each application uses microservices in at least six different programming languages. As a result, it is more difficult to monitor all services, as different monitoring tools might be needed.

[6] `https://github.com/acmeair`
[7] `https://github.com/spring-cloud-samples`
[8] `https://microservices-demo.github.io/`

4. Approach

In microservice applications, a single user request usually causes a sequence of internal calls. We assume that many microservice performance metrics, such as the response time, depend on the number of incoming requests within a time interval. Our idea is to predict the user requests and their propagation through the application. Using this information, we infer the performance of the microservices. Our approach is split into eight steps, whereof six represent the core functionality and the last two are possible extensions. Figure 4.1 provides an overview of our approach. In the first step, the model structure extraction, we collect and analyze gathered monitoring data to extract connections between the services and identify communication patterns. These data are the basis for our request propagation model, which is built in an iterative way. In step three, we utilize load forecasting to predict the user requests in the next prediction interval. Afterward, we predict how these user requests propagate on all microservices of the application. As a result, we obtain the arrival rates for every service. These features function as inputs for our performance inference model. In step six, we predict performance metrics and provide state ratings for every service by iteratively stepping through the application topology. As optional extensions, we propose root cause reporting for predicted performance degradations and self-improvement, which aims to identify causes for wrong predictions and adjusts prediction models, if necessary. In the following, we describe these eight steps in a more detailed fashion. A simple example of the calculations made by the approach is provided at the end of this chapter.

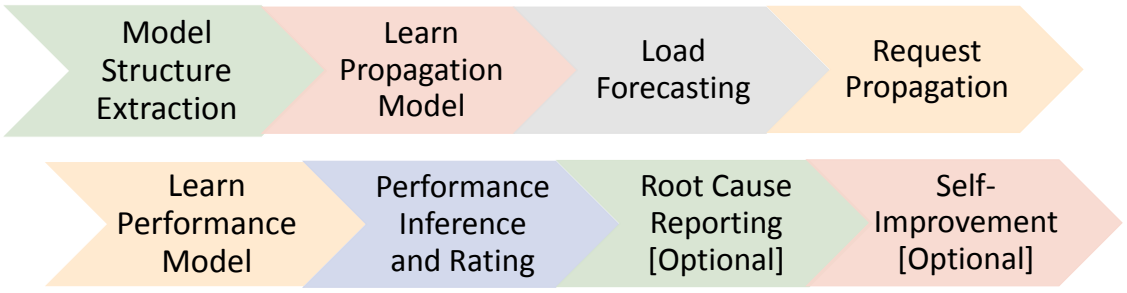


Figure 4.1.: Approach Overview

Terminology and Fundamental Definitions. Before we describe our approach in more detail, we introduce some important terms, which will be used in the remainder of this work. In the following, the term *service* represents an application component, which has a clear-scoped functionality. Deployment properties do not play any role on this abstract level and are not considered here, which means that in practice, a service can consist of one or multiple instances, for example. A service always consists of one or multiple *endpoints*. An endpoint is an interface of a service, which can be assessed by other services or by a user. In the fields of microservice applications, these endpoints can be REST endpoints. The differentiation between different endpoints of one service gives us the possibility to consider multiple workload classes for one service. This is important for a good performance prediction and will be discussed later in more detail. As mentioned earlier, the application topology plays an important role in our approach. A service can have different roles within an application. A *backend service* is a service, which does not call other services and responds to incoming requests only. A *frontend service* is a service, which is called by the user and therefore initializes the processing of user requests. However, a frontend service can also be called by other services as well.

Model Structure Extraction. In this first step of our approach, monitoring and tracing data of the application are gathered and processed. These data serve as training samples for our prediction models. Our approach works based on two kinds of training data. In order to extract the application topology, we need information about the internal requests, which are made to process an incoming user request. This information might be available in the form of call stacks or call trees, which are generated by many state-of-the-art monitoring frameworks. Additionally, measured performance values for every service must be available. Depending on the availability, more metrics, which influence the service performance, can be measured optionally.

Request Propagation Model. The extracted data from the previous step are used as inputs for the request propagation model, which describes how many additional internal inter-service calls are needed to process an incoming user request. We visualize this model with the *propagation matrix* D.

$$D = \begin{pmatrix} d_{0,0} & d_{0,1} & \cdots & d_{0,s-1} \\ d_{1,0} & d_{1,1} & \cdots & d_{1,s-1} \\ \vdots & \vdots & \vdots & \vdots \\ d_{s-2,0} & d_{s-2,1} & \cdots & d_{s-2,s-1} \\ d_{s-1,0} & d_{s-1,1} & \cdots & d_{s-1,s-1} \end{pmatrix}$$

Each entry $d_{i,j} : \mathbb{R}^{m_i} \to \mathbb{R}^{m_j}$ in D represents a function, which maps incoming requests at service i to a number of calls of service j initiated by service i. Obviously, every propagation function returns zero for all inputs smaller or equal to zero because we assume that, if service i receives no requests, it does not call any other service j. More clearly spoken, it is a representation of the application topology and describes how incoming requests are propagated through the application. Both the input and the output of a function $d_{i,j}$ are vectors. The dimensionality is thereby determined by the variable m_a, which represents the number of endpoints of service a. As a consequence, our model is able to assess both incoming requests per service and the distribution across its endpoints. This is useful for performance prediction, as different endpoints might have different performance metrics, e.g., considering response times, an HTTP POST request might take longer to process than a GET request. The matrix D has s^2 entries overall, where s represents the number of (monitored) services in the application. An example topology and a corresponding matrix

representation is given at the end of this chapter. Note, that in this step deployment information is neither needed nor considered.

To build this model in an iterative way, we initialize all functions $d_{i,j}$ in a way that the output is always the null vector, which represents no dependency between the services i and j. After an observation interval, we update those functions $d_{i,j}$, where a request between service i and j has been recorded. By doing this in the following observation intervals as well, we build a representative model of the application topology step by step. We expect that the model will converge against a steady state after a certain time, as all communication patterns have been observed. If the matrix does not change significantly over multiple time steps, it might be practical to turn the model updates off.

Special Request Propagation Model Instances. In the following, we describe some selected instances of the request propagation model. Thereby, we use the characteristic of the model that is a representation of the application topology. An application topology can be visualized as a directed graph, where each node represents a service, and an edge from node A to node B means service A sends requests to service B. Our request propagation model has nearly the same graph representation. The only difference is that the edges are the propagation functions. Consequently, we can describe paths in this graph by composite functions. For example, a path from A over B to C can be written as the composite function $d_{A,C} = d_{B,C} \circ d_{A,B}$. We hereby use the default definition of the composition operator $\circ$[1].

The first important type of request propagation models are *acyclic models*. A request propagation model is acyclic, if no path $d_{i,i} = d_{i,j} \circ d_{j,k} \circ ... \circ d_{z,i}$ exists, where not at least one component function is the null function. In other words, if we draw the topology graph and omit all edges, which are null functions in the request propagation model, the resulting graph must be acyclic. Analogously, a request propagation model is cyclic, if at least one path $d_{i,i} = d_{i,j} \circ d_{j,k} \circ ... \circ d_{z,i}$ exists, where none of the component functions is the null function. The second important type of request propagation models are *regular models*. A request propagation model is regular, if the sequence

$$d_{i,i}(x)^n = \underbrace{d_{i,i} \circ d_{i,i} \circ ... \circ d_{i,i}}_{\text{n times}}$$

converges to zero for $n \in \mathbb{N}$ tending to infinity, every cycle $d_{i,i}$ and every input vector x. Such models represent topologies, where cycles exist but no infinity loops. This means that after a certain time no requests are sent through the cycle anymore, if no new requests enter the cycle. Every acyclic model is also regular. In addition to this, request propagation models can be distinguished by the types of the propagation functions. For example, a request propagation model is *linear*, if every propagation function $d_{i,j} : \mathbb{R}^{m_i} \to \mathbb{R}^{m_j}$ has the form $d_{i,j}(x) = c_{i,j} \cdot x$, where $c_{i,j} \in \mathbb{R}^{m_j \times m_i}$ is a constant matrix.

Load Forecasting. In general, user requests are the root cause of transactions between microservices. As a consequence, we need load forecasting to predict the number of user requests in the next prediction interval. The result of this forecast is visualized by the list U, where each entry $u_i \in \mathbb{R}^{m_i}$ represents the number of user requests to the m_i endpoints of service i. As s services exist within the application, U contains s lists and $s \cdot m$ numerical entries overall, where m is the sum over all m_i. If no load forecasts with that granularity are available, a simple workaround is proposed. If forecasts for every service are available, but no distribution across the endpoints, the requests can be equally or arbitrarily distributed across the endpoints. However, the possibility to model user requests

[1] Let f and g be functions, then the composite function $g \circ f$ is defined as $g \circ f = g(f(x))$

Table 4.1.: Variables Used in the Request Propagation Algorithm

Name	Type	Description
s	Non-zero integer	Number of services
m_i	Non-zero integer	Number of endpoints of service i
m	Non-zero integer	Sum of all m_i
M	List (Size: s)	Ordered set of all m_i
U	List of Lists (Size: s)	Predicted user requests in the next prediction interval for each service
ϵ	Non-negative real value	Hyperparameter, lower bound for request values
x_i	List (Size: m_i)	Total number of incoming requests to service i per endpoint in the next prediction interval
X	List of Lists (Size: s)	Ordered set of all x_i
$\bar{x}_i$	List (Size: m_i)	Number of requests to service i per endpoint to forward in the current iteration of the algorithm
$\bar{X}$	List of Lists (Size: s)	Ordered set of all $\bar{x}_i$
$\hat{x}_i$	List (Size: m_i)	Number of requests to service i per endpoint generated in the current iteration of the algorithm
$\hat{X}$	List of Lists (Size: s)	Ordered set of all $\hat{x}_i$
0_a	List (Size: a)	List, which contains a zero entries

to specific endpoints, enables us better debugging and testing capabilities and preserves the generality of our model.

Request Propagation Algorithms. Given the service dependencies captured by D and the predicted user requests U, we want to predict how the requests are forwarded through the application. Depending on the model type, different algorithms might be used here. We first describe an iterative algorithm, which works with any regular request propagation model, and hence can be seen as a universal algorithm. Algorithm 1 shows the proposed iterative algorithm, while Table 4.1 shows all used variables. In the following, we describe the proposed algorithm and its properties in a more detailed way.

The request propagation algorithm takes the user request list U, which has been generated in the previous step, as an input and returns the request list X, which contains the total number of predicted incoming requests in the next prediction interval for all services and endpoints. The list X can be seen as the sum of the user requests U and generated internal calls. Hence, at the beginning of the algorithm, the values of U are assigned to X. The while loop then calculates the internal calls iteratively. The list $\bar{X}$ represents the requests, which need to be forwarded in the current iteration of the algorithm, and gets the values of U at the beginning as well. The while loop terminates if there are no requests to forward.

Algorithm 1: Request Propagation Algorithm

Input: Propagation matrix D, user requests U, threshold ϵ, no. of services s, endpoint list M

1 $X \leftarrow U$;
2 $\bar{X} \leftarrow U$;
3 **while** *not all numerical entries in* $\bar{X}$ *equal* 0 **do**
4 $\quad \hat{X} \leftarrow 0_s$;
5 $\quad$ Set all entries $\hat{x}_i$ in $\hat{X}$ to 0_{m_i};
6 $\quad$ **foreach** $\bar{x}_i$ *in* $\bar{X}$ **do**
7 $\quad\quad$ **if** $\bar{x}_i \neq 0_{m_i}$ **then**
8 $\quad\quad\quad$ **foreach** $d_{i,j}$ *in i-th row of* D **do**
9 $\quad\quad\quad\quad \hat{x}_j \leftarrow \hat{x}_j + d_{i,j}(\bar{x}_i)$;
10 $\quad\quad\quad$ **end**
11 $\quad\quad$ **end**
12 $\quad$ **end**
13 $\quad X \leftarrow X + \hat{X}$;
14 $\quad \bar{X} \leftarrow \hat{X}$;
15 $\quad$ Set all numerical entries in $\bar{X}$ lower than ϵ to 0;
16 **end**
17 **return** X;

In the inner loops, the propagation functions $d_{i,j}$ are called for each service i, which has incoming requests to forward, and all target services j. The newly generated internal requests are stored in the temporary support list $\hat{X}$, which gets filled with zeros at the beginning of each iteration (line 4 and 5). After iteration over all services, the newly generated requests $\hat{X}$ are added to the total numbers of requests X and need to be forwarded in the next iteration. Hence, $\bar{X}$ gets the value of $\hat{X}$. Note that, the requests do not have to be integers, as a service can also call another service with a probability of only 80% or similar. This leads to the fact, that the functions $d_{i,j}$ can both receive and output any positive value. To prevent an infinite loop, the algorithms hyperparameter ϵ is used. It represents a lower bound, which is applied to $\bar{X}$, and all numerical entries which are smaller than ϵ will be set to 0. This guarantees, that at some point there will be no requests to forward and the loop will end. One advantage of the algorithm is the ability to handle cyclic topologies as long as cycles generate fewer requests than received in one iteration. This fits our definition of regular request propagation models and is a natural assumption, as the application itself would fall to an infinity loop in such a case. Another advantage is that the algorithm can be parallelized easily, as the only synchronized calls are the commutative addition in line 13 and the assignment in line 14. In the proposed version of the algorithm, the resulting list X represents the total number of requests for every service and endpoint. There is no information about the arrival rate course within a prediction interval, so it is assumed that the service performance depends only on the total number of incoming requests in a prediction interval. This is a valid assumption in particular for small-sized intervals, where the variations of the arrival rate can be neglected. For larger intervals, the arrival rate might be replaced by arrival rate distributions in order to integrate information about the load variations within a prediction interval.

For special instances of the request propagation model, more efficient algorithms might be used to calculate the resulting request list X. As an example, we consider linear acyclic models. As defined earlier, every propagation function $d_{i,j}(x)$ within a linear model can be written in the form of $d_{i,j}(x) = c_{i,j} \cdot x$ with a constant matrix $c_{i,j}$. If we now construct the composition $d_{j,k} \circ d_{i,j}$ of two linear propagation functions $d_{i,j}$ and $d_{j,k}$, a new function

$d_{i,k}(x) = c_{i,k} \cdot x$ emerges, which is still a linear propagation function. This can be proven by the following equivalent transformation:

$$\begin{aligned} d_{i,k}(x) &= d_{j,k} \circ d_{i,j} && \mid \text{Definition of function composition} \\ &= c_{j,k} \cdot (c_{i,j} \cdot x) && \mid \text{Associative property} \\ &= (c_{j,k} \cdot c_{i,j}) \cdot x && \mid \text{Substitution: } c_{i,k} = c_{j,k} \cdot c_{i,j} \\ &= c_{i,k} \cdot x \end{aligned}$$

If a service i receives calls from multiple origins, the total number of incoming requests is the sum of all inbound request flows. Hence, the vector of incoming requests x_i can be written as the sum of multiple linear propagation functions. We further know that the resulting request list X is the sum of the user requests U and the internal calls, while every internal call is originated by a user request. From these properties follows that every vector x_i is a linear superposition of the entries u_i of the user request list U. With that, we are able to calculate every x_i with a single matrix multiplication. For complex application topologies, this can bring some numerical advantages because the iterative calculations from Algorithm 1 are summarized. Additionally to this, the parallelization is more simple. We waive a more detailed description of this and other request propagation algorithms as we use only the described universal algorithm in our evaluations.

Performance Inference Model. As mentioned before, we assume that the performance $\tilde{Q}_i \in \mathbb{R}^{m_i \times r}$ of a service i is dependent on the number of incoming requests to the service. We obtain these requests from the request list X calculated in the previous step. To describe the performance, we use r metrics, e.g., average response time or the number of exceptions. These metrics are calculated for all m_i endpoints of service i. As a consequence, the service performance matrix $\tilde{Q}_i$ of service i has $m_i \cdot r$ entries overall. We assume that for each service i there is a performance function $q_i : \mathbb{R}^{m_i + r \cdot z_i} \to \mathbb{R}^{m_i \times r}$ which maps the requests list $x_i \in \mathbb{R}^{m_i}$ and $r \cdot z_i$ additional input values to the service performance $\tilde{Q}_i$. We define the list Q, which contains all s performance functions.

$$Q = \begin{pmatrix} q_0 \\ q_1 \\ \vdots \\ q_s \end{pmatrix}$$

For considering additional dependent factors on the service performance, we consider the position of the service in the topology graph. We define z_i as a variable, which represents for a given service i the sum of all endpoints of all services, which receive calls from service i. A backend service only responds to incoming requests. As a consequence, for a backend service b the value z_b equals 0. We assume, that its performance $\tilde{Q}_b$ only depends on the number of incoming requests x_b. As a consequence, the performance function q_b maps m_b input values to $r \cdot m_b$ performance metrics. For all other services i, which are no backend services, we assume that their performance $\tilde{Q}_i$ depends on the performance measures of all endpoints, which receive calls from service i. An overview of all variables used for performance inference is given in Table 4.2.

We want to approximate the performance functions by using supervised machine learning and choose r performance metrics, which are measurable. Moreover, we can gather the request list X by using simple request tracing. As a consequence, we are able to train

Table 4.2.: Variables Used for Performance Inference

Name	Type	Description
r	Non-zero integer	Number of performance metrics
z_i	Integer	Sum of all endpoints of all services called by service i
q_i	Function ($\mathbb{R}^{m_i+r \cdot z_i} \rightarrow \mathbb{R}^{m_i \times r}$)	Maps incoming request list and additional parameters to performance metrics for service i
Q	List (Size: s)	Ordered set of all p_i
$\tilde{Q}_i$	Matrix (Dimension: $m_i \times r$)	Predicted performance values of service i

regression models, which approximate the performance function list Q. Therefore, it is possible to use one common or different prediction models for each performance metric. It is likely that the performance functions change over a time period, therefore we aim to update Q on a regular basis using measured performance data. Possible extensions, like deployment information, might be included in this model in future work.

Performance Inference Algorithm. We propose an iterative algorithm to calculate the performance $\tilde{Q}_i$ for each service i, which starts with the backend services and goes through the whole application topology. Therefore, we identify all backend services in the first step. In our model, a service i is a backend service if and only if the i-th row of D contains only null functions. This means, i does not send any calls to any other service. As mentioned earlier, the performance of these backend services does only depend on the request list X, which we obtain from the request propagation algorithm. In the next step, we select those services i, where all dependent services have assigned performance matrices, and predict their metrics. This procedure is repeated until all services have assigned performance matrices. As a possible implementation feature, the calculation of the performance metrics can be parallelized.

Finally, the performance $\tilde{Q}_i$ for every service i is rated using a three-grade system, which is described in Table 4.3. The categorization of the service performance into different classes can be found also in a recent journal article by Bianchini et al. [22]. In our case, the system administrator defines target ranges for every performance metric and every endpoint in advance, which determine the rating. In this step, a simple extension is possible, which takes deployment dependencies into account. We propose an heuristic $h(x_{host}, \tilde{Q}_{host})$ for every host, which depends on all request lists x_{host} of all services deployed on the host and their performance matrices $\tilde{Q}_{host}$. If this heuristic is used, we define a fourth grade, which represents a potential service performance degradation caused by an overloaded host. Such a performance degradation of service i can be especially caused by too many requests to the host (network overload) or by another faulty service j deployed on the same host as i, which, e.g., consumes all computing resources.

The proposed version of the performance inference algorithm does not support cyclic topologies. However, it can be applied to such topologies as well by adding heuristics to resolve a cyclic dependency. A possible heuristic is to ignore the dependency $d_{i,j}$, which generates the fewest calls for a given input list X. This means, more roughly spoken, that the performance of a service i does not depend significantly on a service j, which is called by i only infrequently. Another possibility to resolve cycles is to apply contraction, which

Table 4.3.: Service Performance Ratings

Name	Meaning	Recommended Action
GREEN	No performance degradation expected, all performance metrics in desired range	None
YELLOW	No performance degradation expected, but some metrics are close to their limits	Further observations
RED	Performance degradation expected, some or all performance metrics out of desired range	Identify root cause, create new instance etc.
(Optional) BLUE	Performance degradation probable, host is overloaded	Change deployment

means that all services in the cycle are summarized to one macroservice. This might be a suitable option if all services in a cycle provide similar functionalities. An option to minimize the probability of cycles is to replace the services by their endpoints. Thereby, cycles, which exist between different endpoints of two services, are resolved. An example of this approach is given in Chapter 6. The following steps summarize the performance inference algorithm:

1. Identify all backend services
2. Predict the performance matrices of all backend services
3. Create a list L of all services, for which all dependent services have performance matrices assigned
4. Predict performance matrices of all services in L
5. Repeat step 3 and 4 until all services have performance matrices
6. Rate the predicted state of all services (Optional: with deployment heuristic)

Root Cause Reporting (Extension). In this step, the root cause for a performance degradation is reported to an administrator. Even if this step is optional, the basic version of our approach gives a rough idea of the root cause implicitly. If the performance ratings are calculated, a human can identify root cause candidates, if the application topology is known (see also [33]). For example, a service is a root cause candidate if its rated red but all dependent services are rated green. This means, that the failure is likely to be caused by that service. However, we do not propose an explicit algorithm to identify the root cause for performance degradations.

Self-Improvement (Extension). By comparing the forecast performance and request matrices with actually measured values, we can assess the prediction quality at runtime. A mechanism could be included, which identifies causes of prediction errors (e.g., request propagation, load forecasting, etc.). In a second step, the prediction technique or underlying model could be changed to achieve a better prediction result.

Example. We want to demonstrate our approach on a plain example. Figure 4.2 shows a simple application topology. The three boxes represent one service each and are numbered

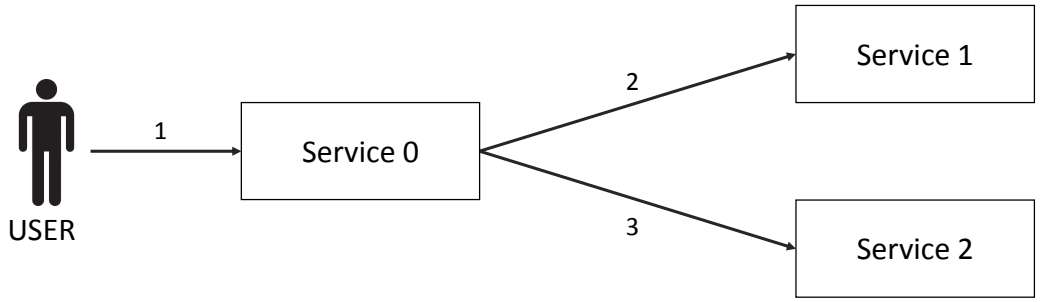


Figure 4.2.: A Simple Request Propagation Model

from 0 to 2. We assume, that each service has only one endpoint. This means, that the value m_i equals 1 for every service i and each input x_i can be represented using a scalar. Moreover, each entry $d_{i,j} : \mathbb{R} \to \mathbb{R}$ in D is an one-dimensional mapping. The numbers on the edges shown in Figure 4.2 mean, that one incoming request on service 0 creates two calls to service 1 and three calls to service 2. One possible propagation matrix, which represents these relationships using linear functions is:

$$D = \begin{pmatrix} 0 & 2x_0 & 3x_0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix}$$

In this case, $d_{0,1}$ and $d_{0,2}$ are set, while all other functions are set to 0, which means that no other dependencies are known. In the next step, we assume that our forecasting mechanism predicts two requests to service 0 and no other requests in the next prediction interval. This means that the user request list U is given by:

$$U = \begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}$$

In the next step, we forward the user requests through our application. Therefore, the values of U are assigned to the request list X first. In the first iteration, we generate $2 \cdot 2$ requests to service 1 and $2 \cdot 3$ requests to service 2. In the second iteration, no further requests are generated, which means that the algorithm terminates with the following result:

$$X = \underbrace{\begin{pmatrix} 2 \\ 0 \\ 0 \end{pmatrix}}_{U} + \underbrace{\begin{pmatrix} 0 \\ 4 \\ 6 \end{pmatrix}}_{\hat{X}\text{Iteration 1}} + \underbrace{\begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}}_{\hat{X}\text{Iteration 2}} = \begin{pmatrix} 2 \\ 4 \\ 6 \end{pmatrix}$$

As a performance metric, we use the average response time in milliseconds (ms). As this is the only metric we use, the performance $\tilde{Q}_i \in \mathbb{R}$ of a service i is a scalar. For simplicity, we assume that for every service the response time increases linearly with the number of incoming requests in the prediction interval. The performance of service 0 is additionally dependent on the performances of service 1 and 2. These relationships are gathered in the performance function list Q:

Table 4.4.: Classification Table for Performance Rating

Service	Green	Yellow	Red	Predicted Response Time	Result
0	< 200 ms	< 250 ms	≥ 250 ms	150 ms	Green
1	< 40 ms	< 60 ms	≥ 60 ms	30 ms	Green
2	< 40 ms	< 60 ms	≥ 60 ms	40 ms	Yellow

$$Q = \begin{pmatrix} 20 + 5x_0 + max(2\tilde{Q}_1, 3\tilde{Q}_2) \\ 10 + 5x_1 \\ 10 + 5x_2 \end{pmatrix}$$

In the first iteration, the performance inference algorithm calculates the performance measures of all backend services. In this case, services 1 and 2 are backend services, as their respective rows in D contain only null functions. In the second iteration, the performance of service 0 is determined. The final result $\tilde{Q}$ is displayed in Eq. 4.1. Table 4.4 shows the rating rules for this example. By applying these rules to the predicted response times, both service 0 and service 1 get green ratings, while service 2 is assigned a yellow rating.

$$\tilde{Q} = \begin{pmatrix} \tilde{Q}_0 \\ \tilde{Q}_1 \\ \tilde{Q}_2 \end{pmatrix} = \begin{pmatrix} 150 \\ 30 \\ 40 \end{pmatrix} \tag{4.1}$$

Summary and Discussion. The approach relies on two fundamental models, which influence the prediction power of the algorithm: the request propagation model, an extended application topology, and the performance inference model. In the first phase, predicted user requests are forwarded through the application. In the second phase, the performance of each service starting with the backend services is determined. The described mechanism is a general approach, which has several extension points and the models make no requirements on the prediction technique, which means that the way how to build D and Q is neither prescribed nor restricted. This enables the possibility to evaluate different prediction models. Moreover, the approach makes no critical assumptions about the application and does not need prior knowledge. Furthermore, the output is readable and interpretable by humans and the algorithm produces explainable results. We choose a comprehensive three-grade rating system, which can be adjusted to the use case. For example, the yellow rating can be used as a buffer, which takes the accuracy of the framework on a specific application into account. Each rating includes an implicit recommendation for possible degradation prevention actions. Another advantage of the algorithm is that it requires only common measurable quantities and can be implemented lightweight and fast, which is necessary to work within online environments. More strengths and weaknesses of the approach are discussed in Chapter 7.

Reference Architecture. In the last paragraph of this chapter, we introduce a reference architecture for the realization of our approach. This architecture is an enhancement of the approach of Mueller [50]. The previously described models are universal and abstract. Hence, an important requirement for the architecture is flexibility and extendability. An implementation must be able to support different types of request propagation models, performance inference models, and forecasting techniques, for example in order to enable self-improvement mechanisms. An overview of the proposed architecture is given in Figure 4.3. One central component is the core, which coordinates and configures all other

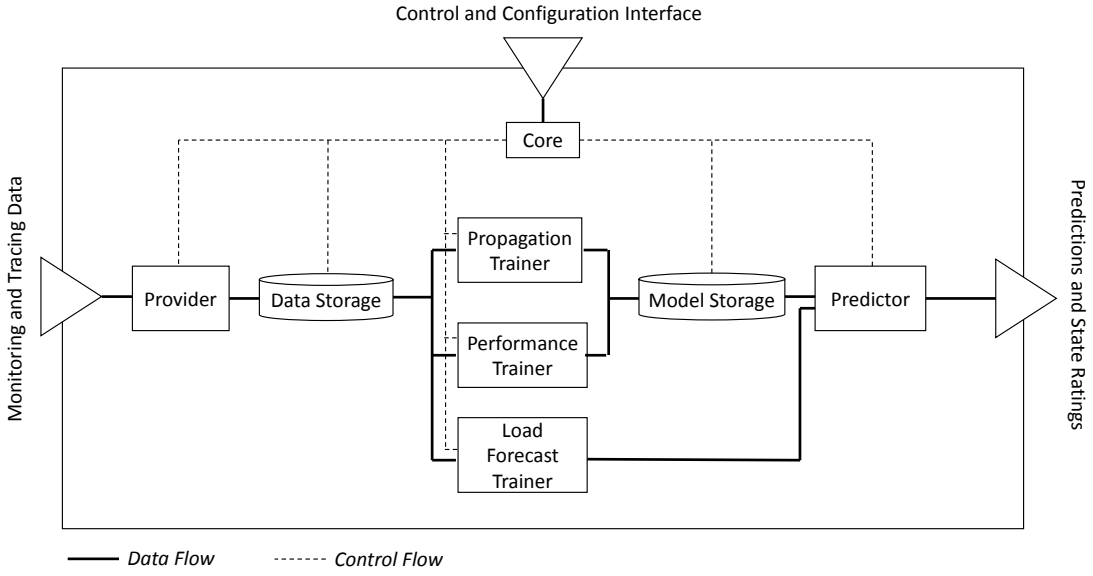


Figure 4.3.: Reference Architecture

components. It might also manage the integration of new components. The core processes all control commands at the start and runtime of the framework and should be able to start or stop the prediction process. The core receives the commands via a control and configuration interface.

All other components work as a pipeline, which transforms monitoring and tracing data to performance predictions and service state ratings. At the beginning of this pipeline, the provider component processes monitoring data and brings them into a defined format. In practice, multiple instances or implementations might be needed to support different monitoring systems and data formats. Similarly, it remains open whether the monitoring data represent realtime measurements or recorded traces. This leaves the possibility to use the approach online or offline open. The provider sends transformed monitoring data to a data storage. This storage is used by other components to extract model training data. As an example, the load forecaster uses the load statistics and courses to predict the load in the next prediction intervals and sends its forecasts to the predictor component.

The monitoring data are also used for the training of the request propagation and performance inference models. Therefore, we suggest the usage of two separate trainer components. This increases the exchangeability and enables individual configurations. For example, it could be useful to train the models at different intervals. The trained models are persisted in a model storage. This storage enables the possibility to analyze and eventually reuse past models. The last component in the pipeline is the predictor component. It computes performance predictions and state ratings based on the recent request propagation model, load forecasts, and performance inference model. More detailed descriptions of the components and their implementations are given in the next chapter.

5. Implementation

This chapter focuses on the implementation and realization of the described theoretical approach. Therefore, Section 5.1 delivers an overview of the implemented framework *PPP* (*P*ropagation *P*erformance *P*rediction), while Section 5.2 provides a detailed description of each component and its configuration options. Finally, Section 5.3 introduces the operation modes of the PPP framework, which facilitate the evaluation process of this work.

5.1. The PPP Framework

The PPP framework consists of ten components overall, which are implemented as microservices and run inside dedicated Docker containers. We provide configuration files both for a local and distributed deployment. Figure 5.1 shows the components of the PPP framework. The design is an enhancement of the architecture proposed in the approach of Mueller [50]. In the following, we summarize the main functionalities and operation of the implemented software.

To gather monitoring data of the test application, we instrument all or some of its microservices by attaching Pinpoint agents at startup. These agents send data to the Pinpoint Collector, which processes the data and saves them into an Hbase database. By using the Pinpoint Web API, we can query all monitoring data and use them in our framework. The workflow of PPP can be split into data querying and processing (provider component), model generation and training (performance and propagation trainer), data prediction (predictor and load forecaster), coordination and controlling (core) and data persistence (propagation, performance, model and prediction persistor). Inter-service communication within the PPP framework is realized by a Docker network and payloads are formatted using the JSON standard. As shown in Figure 5.1, the provider component receives the data from Pinpoint and divides them into performance and propagation data. These data are used by the trainer components to train the associated models. We decided to separate the performance and propagation data flows as far as possible, to increase configuration options and enable easier debugging. The degree of this separation is also dependent on the format of the monitoring data and consequent preprocessing steps. We describe related issues in Section 5.2.

The trained models are stored in databases and used by the predictor to make predictions according to the proposed method from Chapter 4. The resulting predictions are stored into databases as well and compared with the measured data for the evaluation. The

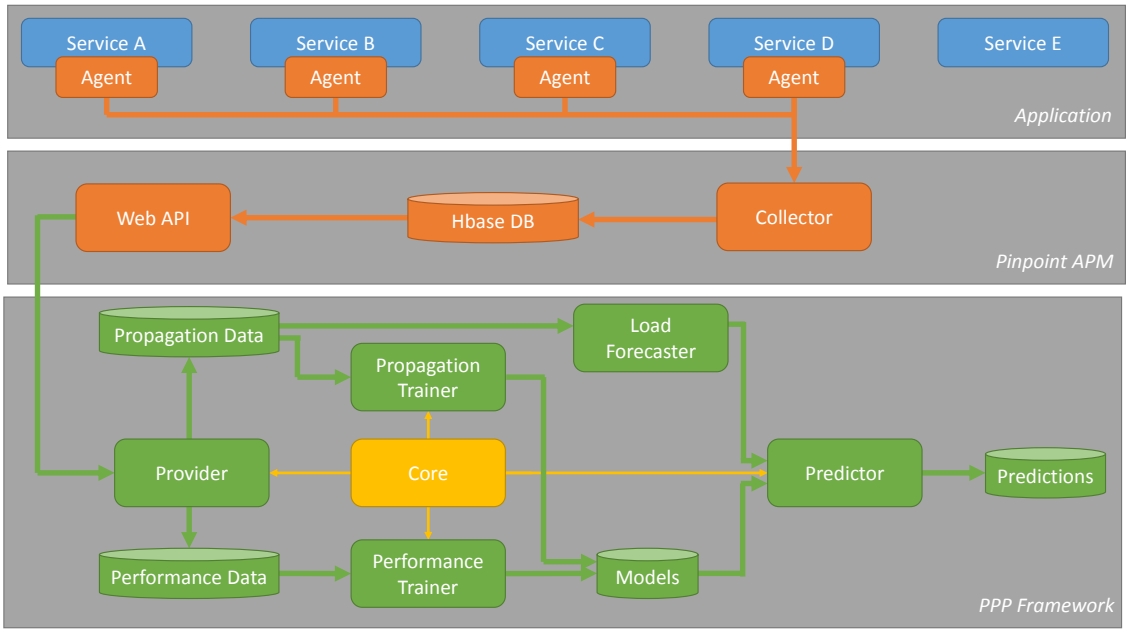


Figure 5.1.: Overview and Data Flows

predictor additionally needs the forecast of the user requests U, which is calculated by the forecaster component. In this work, we use Telescope as our forecasting tool. Note that both forecaster and provider components can be easily substituted in case an other forecasting mechanism or APM tool is used.

5.2. Description of Components

In the following section, we describe the functionality and interaction of all components in the framework in a more detailed way. Therefore, we illustrate outputs and intermediate results on the example shown in Figure 5.2. Here, we observe an incoming user request at endpoint A of service A in our monitoring interval. This request creates 24 new requests from service A to endpoint B of service B. We assume, that no other data have been monitored in the same interval.

Core. The core component takes the role as the central coordinator of the PPP framework. Its major responsibility is the initiation of actions executed by other components, like gathering new monitoring data, requesting a performance prediction, and retraining of models. Moreover, the core processes and forwards the framework configuration. It

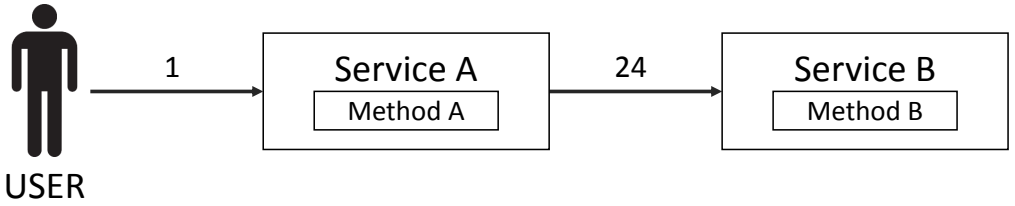


Figure 5.2.: Running Example in this Chapter

Table 5.1.: Core Configuration Parameters

Name	Type	Runtime Change	Description
intervalSize	Non-zero integer	No	Length of monitoring interval in milliseconds
requestDelay	Non-zero integer	No	Threshold for communication and processing time of Pinpoint
evaluationMode	String	No	See Section 6.1
startTime	Integer	No	Start time of trace if evaluation mode is *replay*
endTime	Integer	No	End time of trace if evaluation mode is *replay*

provides the endpoints `start` and `stop` for starting and stopping the framework. As an input, the start mechanism needs a JSON-formatted configuration object. This object contains all settings for the framework, including parameters for the core itself and other services. A central configuration parameter for the core is the size of the monitoring interval. The provider is contacted with that frequency and requests all monitoring data from Pinpoint from the previous interval. Moreover, the predictor is contacted concurrently and predictions are made for future intervals. The core supports different evaluation modes, including the *replay* mode, which allows analyzing recorded traces. Therefore, start and end times can be provided in the configuration. The evaluation modes are described in Section 5.3. All configuration parameters which affect the core are listed in Table 5.1.

For the communication with different components, the core manages multiple threads, which are activated if a message must be sent. The core provides an endpoint `reportError`, where all other services can report unexpected failures, which creates a central logging mechanism. These error descriptions are the only messages, which the core receives from other services. The core does not forward or receive any monitoring or forecasting data.

Provider. The provider is responsible for requesting and processing the monitoring data gathered by Pinpoint. As an input, the provider receives the interval from which monitoring data should be transmitted. This information is forwarded to the Pinpoint Web API. First, the provider requests all service and agent information, including names, start times, and other stats. A knowledge base is built incrementally containing the information of the agents and services. Second, performance and tracing data need to be requested. In Pinpoint version 1.8.4, the interface `transactionInfo` provides that functionality. It requires a `transactionId` as an input. If a user sends a request to a service monitored by a Pinpoint agent, this request is labeled with such an ID. It contains the agent name, its start time, and a sequence number, which represents an incrementing positive integer. These entries are divided by the hat character `^`. For example, the first user request to the agent named `agent1` and started at timestamp `100` would be labeled with the transactionId `agent1^100^1`. The next user request to this agent would receive the id `agent1^100^2` and so on. All inter-service calls initiated by that user request are labeled with the same transactionId. The provider uses caching to save both the agent information and last sequence number, which accelerates the request process. In future work, a plugin for Pinpoint, which provides an interface where all transactionIds generated within a specific interval, could be developed, which would speed up this process even further.

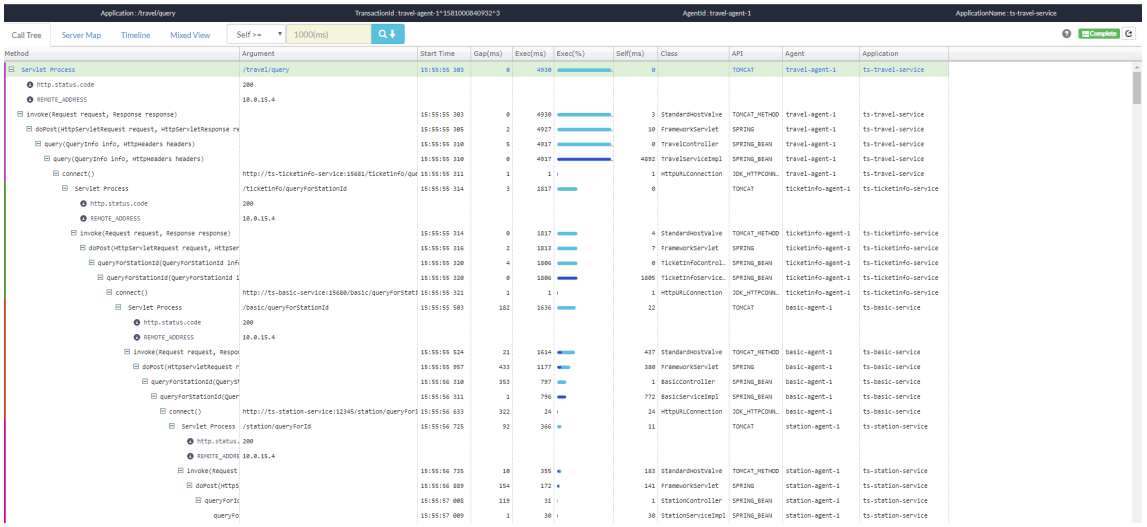


Figure 5.3.: Transaction View and Call Tree of Pinpoint

This kind of labeling enables the possibility to build a call tree for each user request. In the Pinpoint Web GUI, such a call tree can be visualized by using the `transactionView`. An example of that view is shown in Figure 5.3. The call tree contains all information that is needed to build our performance and propagation models, including performance metrics like response times and exceptions. Moreover, the start and end time of the time spent processing the user request is available. We say that a user request and its associated internal calls belong to the monitoring interval $[a; b]$ if the start time of its call tree is bigger than a and smaller or equal b. This definition ensures that the statistics and gathered data represent the true state of the application in this interval. We hereby assume that the monitoring interval is larger than the processing time of all or most user requests. If this assumption is violated to a significant extent, statistics and training data get distorted. This is because the processing of the user request consumes computing resources in future monitoring intervals and therefore influences performance metrics. This relationship is not recorded in the data of these intervals, which influences the training data of our performance models.

In the next step, the provider extracts the relevant information on the response of `transactionInfo` and calculates the performance metrics. For example, the inter-service calls are filtered and average response times are calculated. An example output of the provider is shown in Listing 5.1. It is shown that this output contains both propagation and performance data. The separation of these data is done by the performance and propagation persistors, which receive these data frames from the provider. The Pinpoint Web URL must be set as a parameter in the system configuration. Moreover, the user of the framework can specify a set of regular expressions, which shorten and filter the monitored application URLs. For example, we usually want to remove the parameters of a GET request if they are specified as URL extensions, e.g., in `http://example.com/getNumber?x=2`, we want to remove the parameter `x`. If we do not perform this step, we would record separate statistics and performance metrics for each different parameter `x`. This solution is obviously very application-specific. Problems occur especially in cases when no simple regular expression can be designed. In future work, this problem can be targeted, e.g., by an additional preprocessing step before training, where knowledge from different monitoring intervals can be included.

Table 5.2.: Provider Configuration Parameters

Name	Type	Runtime Change	Description
pinpointWebHost	Hostname or IP Address	Yes	Network address of Pinpoint Web
pinpointWebPort	Non-zero integer	Yes	Port number of Pinpoint Web
methodShortener	Set of Strings	No	Set of regular expressions to filter endpoint names

```
{
  "userRequests": {
    "serviceA:/endpointA": 1
  },
  "transactions": {
     "serviceA:/endpointA serviceB:/endpointB": {
       "number": 24,
       "averageResponseTime": 7.25, ...
     },
     "user serviceA:/endpointA": {
       "number": 1,
       "averageResponseTime": 232, ...
     }
  },
  "from": 1581001986984,
  "to": 1581002006984
}
```

Listing 5.1: Example Output of Provider Component

Performance Persistor. The performance persistor manages access to a connected database, which stores the training data for the performance inference model. In the PPP framework, all databases use the database management system *CouchDB*[1]. This is a document-oriented NoSQL system, which allows to store and query JSON-formatted documents without a large configuration overhead. At the start-up time of the performance persistor, the connection to the database is established and all tables are initialized. Moreover, the so-called design document is transmitted, which defines the functions for storing and querying a document. The component also provides interfaces for emptying and recreating all tables.

The main responsibility of the performance persistor is the extraction and processing of the performance data contained in traces sent by the provider component. First, all transactions are grouped by their destination endpoint because we assume that the caller of an endpoint does not influence the endpoint's performance metrics. Second, the performance metrics for every endpoint, such as average and maximum response time, are calculated. Afterward, the resulting performance trace is saved into the database. The resulting document for our running example is shown in Listing 5.2.

[1] https://couchdb.apache.org/

```
{
  "performanceValues": {
    "serviceB:/endpointB": {
      "number": 24,
      "averageResponseTime": 7.25, ...
    },
    "serviceA:/endpointA": {
      "number": 1,
      "averageResponseTime": 232, ...
    }
  },
  "from": 1581001986984,
  "to": 1581002006984
}
```

Listing 5.2: Example Performance Trace

Propagation Persistor. The propagation persistor is the counterpart of the performance persistor and is responsible for extracting and processing the propagation data from incoming provider traces. Moreover, it stores the user request time series, which are mandatory for the load forecaster service. It contains the same interfaces for initializing, accessing, and managing the connected database. The propagation persistor removes the performance data from the provider trace and focuses on the connections between the services. For every endpoint, the number of incoming and outgoing requests as well as the destination endpoint for every outgoing request is determined. Again, we assume that the caller of an endpoint does influence neither the performance metrics nor the propagation behavior. Therefore, the amount of incoming requests is sufficient to save and the caller endpoint does not need to be reported. The resulting propagation trace also contains an entry named `user`, which contains all user requests and has zero incoming requests, and time stamps, which are used for querying the trace. An example propagation trace is shown in Listing 5.3.

```
{
  "propagationValues": {
    "serviceB:/endpointB": {
      "incoming": 24,
      "outgoing" : {}
    },
    "serviceA:/endpointA": {
      "incoming": 1,
      "outgoing": { "serviceB:/endpointB": 24 }
    },
    "user": {
          "incoming": 0,
          "outgoing": { "serviceA:/endpointA": 1 }
    }
  },
  "from": 1581001986984,
  "to": 1581002006984
}
```

Listing 5.3: Example Propagation Trace

Table 5.3.: Forecaster Configuration Parameters

Name	Type	Runtime Change	Description
timeSeriesSize	Non-zero integer	No	Maximum length of time series to be used for load forecasting
autoInterpolation	Boolean	No	Determines whether linearly interpolated values should be inserted
interpolationMargin	Fractional number	No	Determines in which cases interpolation is applied

The propagation persistor also supplies the data needed in the load forecaster component. The persistor uses caching so that the database does not have to be queried every time a load forecast has to be made. The user requests are therefore saved in a separate data structure, which represents a time series containing the incoming user requests to a specific endpoint. The maximum length of this time series can be configured in the system configuration. Some load forecasters require nearly equidistant data points in the time series. To ensure this even in cases when some data points are lost, we automatically insert linearly interpolated data points if two consecutive data points are too far apart. Via the system configuration, the minimum distance as a factor of the monitoring interval can be configured. Moreover, this step can be deactivated completely as well. All configuration parameters for the user request time series are summarized in table 5.3.

Load Forecaster. The load forecaster has the task to predict the incoming user requests for every endpoint for the next intervals. The number of values to predict is determined by the predictor's forecast horizon. The forecaster receives the measured time series as an input from the propagation persistor. As mentioned before, we use GluonTS as our load forecaster. The GluonTS Python library and its dependencies are installed at the build time of the load forecaster image. The output of the load forecaster is a mapping, where the forecast values are assigned to their respective endpoints. The predictor component queries this information before executing the request propagation algorithm.

Propagation Trainer. The propagation trainer uses a set of propagation traces from different monitoring intervals to create a propagation model. The time interval from which training data are consumed during training can be adjusted in the system configuration. This is useful because the propagation behavior can change over time and long past values should not influence the resulting model. If the trainer module does not receive any training data, the training process is suspended.

As the first step, the received propagation traces are converted into a directed graph, where each node is a specific endpoint and an edge from A to B means that A sends requests to B. An edge from A to B is also associated with the propagation function $d_{a,b}$. A propagation function is characterized by a set of parameters ψ and an expression $(\psi, x) \to y$, which represents the calculation specification, how the output y is calculated from given parameters and input x. In the current implementation, every propagation function $d_{i,j}$ is modeled as a linear function in the form $d_{i,j}(x) = c \cdot x$, where c is non-zero positive number. This parameter represents the number of requests, endpoint i sends

to j per incoming request at endpoint i. During the training process, the parameter c for every edge in the graph is trained. We hereby calculate the arithmetic average of c of all monitoring intervals. The following example illustrates this process. In the first monitoring interval, we observe one incoming request at endpoint i. This request creates six requests from endpoint i to endpoint j. In this case, c would be equal to 6. In the second monitoring interval, we observe three incoming requests at endpoint i and 12 requests sent from i to j. In this interval, c would be $12/3 = 4$. If we now take the average of these two intervals, the resulting parameter c, which would appear in the model would be equal to 5. This form of averaging is sensitive against outliers, which is good for our usage scenario, especially when we want to detect changing communication patterns over time. Other forms of propagation functions can be specified if needed.

The result of this process is a connected graph, which contains a node named `user` from which all nodes can be reached. The propagation model tries to predict the internal calls caused by user requests. Hence, the user himself does not play any role in the propagation model and we delete this node and all outgoing edges. The resulting graph can now be interpreted as a dependency graph, where an edge from A to B means *A's performance depends on B*. For every node, we collect the targets of its outgoing nodes and save them into a dependency map. This dependency map is mandatory for the performance trainer to calculate the training vectors and thus transmitted to this component.

After this transmission, we examine whether the dependency graph contains any cycles. If the graph is acyclic, we utilize a dependency resolution algorithm to retrieve an ordered list of all endpoints. At the beginning of this list, all endpoints with no incoming edges appear, which means that they are only called by the user. The total incoming requests of these endpoints can be calculated first in the request propagation algorithm. At the end of this list, those endpoints, which do not send requests to other services appear. In the performance inference algorithm, this list can be reversed to calculate the performance metrics. This is why this list is named `calculationOrder` and by providing this list, we shorten the runtime of the predictor component. If the graph contains cycles, we cannot calculate such a list. Note that we did not observe any cycles on this level in any of our test applications. We do also expect these cycles to appear only in very exceptional cases in practice too, as two endpoints of different microservices should not call each other. On the contrary, the case, where two services call different endpoints of each other, does not cause any problems here, as every endpoint is modeled as an independent node in the graph.

If these processing steps are done, the resulting model is submitted to the model persistor. The model is labeled with a timestamp, which represents the time when the last training data have been recorded. Moreover, the resulting model contains the dependency functions, the calculation order for acyclic graphs, and a list of all endpoints, which are contained in the model. The JSON-formatted propagation model for our running example is shown in listing 5.4.

Model Persistor. The model persistor service manages access to a connected CouchDB, where propagation and performance inference models are stored in separate tables. The persistor provides interfaces for saving and querying these models individually or in pairs. For evaluation and debugging purposes, models can also be queried by their timestamp. As the predictor wants to use the most recent models during normal operation, we keep both the most recent propagation and performance model cached. Moreover, the latest timestamp of the models can be queried separately to ensure that every model instance is only transmitted once. The model persistor has the responsibility that the predictor always receives two consistent models, which means that all endpoints and services in the

performance model are also covered in the propagation model and vice versa. If no valid pair of models is available, no model instance will be transmitted and the predictor will suspend the prediction.

```
{
  "nodes": ...,
  "functions": [
    {
      "source": "serviceA:/endpointA",
      "target": "serviceB:/endpointB",
      "function": {
        "type": "Linear",
        "parameters": "24.0"
      }
    },
  ],
  "calculationOrder": [
    "serviceA:/endpointA",
    "serviceB:/endpointB"
  ],
  "dependencyMap": {
    "serviceA:/endpointA": [
      "serviceB:/endpointB"
    ],
    "serviceB:/endpointB": []
  },
  "time": 1581002006984
}
```

Listing 5.4: Example Output of Propagation Trainer

Performance Trainer. The performance trainer transforms a set of performance traces, which are queried from the performance persistor, and a dependency map, which is calculated as the first step in the propagation trainer, to a performance model. As mentioned before, we consider the prediction of performance metrics as a regression problem as we aim to predict continuous metrics like response times. We assume that the performance of an endpoint i is dependent on the number of requests to the endpoint itself and all other endpoints of the same service, as well as on the performance metrics of all endpoints, which receive requests from i. The performance trainer receives the dependency relations for every endpoint from the propagation trainer in the form of a dependency map as shown in listing 5.4. Hence, the training process is started as soon as a dependency map is received. This guarantees that for each propagation model an associated performance model is created and consistent predictions can be made.

As the first step in the training process, the sample vectors are created. An example training vector for an endpoint, which belongs to a service with n endpoints and depends on z endpoints from other services, looks like:

$$\Big(\underbrace{x}_{\substack{\text{no. of requests}\\ \text{to endpoint}}} \quad \underbrace{x_2 \quad x_3 \quad \dots \quad x_n}_{\substack{\text{no. of requests}\\ \text{to other endpoints of same service}}} \quad \underbrace{p_1 \quad p_2 \quad \dots \quad p_z}_{\substack{\text{performance of dependent}\\ \text{endpoints}}} \Big)$$

Table 5.4.: Trainer Configuration Parameters

Name	Type	Runtime Change	Description
retrainInterval	Non-zero integer	Yes	Retraining interval in milliseconds for propagation and performance model
dataInterval	Non-zero integer	Yes	Time interval in minutes, which determines the data to be used for training
metric	String	Yes	Name of performance metric to be predicted
modelConfig	Configuration	Yes	Type and parameters for the machine learning model to be used for performance predictions

The order of the features is saved in the performance model to guarantee valid predictions. We prepare these kinds of sample vectors for every endpoint and every monitoring interval and use them for the training of the machine learning model. In the current version of the PPP framework, we support those four algorithms, which are also mentioned in [20], namely Bayesian, random forest, k nearest neighbors (KNN), and support vector regression (SVR). For this purpose, we use the model implementation from the wide-spread Python library `scikit-learn`[2]. The model type and settings, as well as the metric to predict, can be set in the system configuration. Table 5.4 shows all parameters for the trainer components, while all model settings are named similarly to the associated parameters specified in the scikit-learn documentation[3]. If the training process is completed for all endpoints, the resulting models are serialized as strings to be transmitted to the model persistor. The predictor can deserialize the models to use them for prediction. The performance model receives the same timestamp as the associated propagation model. An example output of the performance trainer is shown in Listing 5.5.

```
{
  "performanceModels": {
    "serviceA:/endpointA": { "\"py/object\"... },
    "serviceB:/endpointB": { "\"py/object\"... }
  },
  "services": {
    "serviceA": [ "serviceA:/endpointA" ],
    "serviceB": [ "serviceB:/endpointB" ]
  },
  "time": 1581002006984
}
```

Listing 5.5: Example Output of Performance Trainer

[2] `https://scikit-learn.org/`

[3] See documentations for classes `sklearn.svm.SVR`, `sklearn.neighbors.KNeighborsRegressor`, `sklearn.linear_model.BayesianRidge` and `sklearn.ensemble.RandomForestRegressor`

Table 5.5.: Predictor Configuration Parameters

Name	Type	Runtime Change	Description
forecastHorizon	Non-zero integer	Yes	Forecast horizon for performance and load forecast prediction as a factor of the monitoring interval
epsilon	Positive number	Yes	ϵ value of the request propagation algorithm

Predictor. The predictor component uses associated propagation and performance models as well as a load forecast to predict the performance metrics for every endpoint and service of the monitored application. The forecast horizon is specified by a configuration parameter. To calculate the predicted values in accordance with the procedure presented in Chapter 4, the load forecast is requested from the associated component as the first step. Afterward, the predictor controls whether a new pair of models has been generated. If so, the models are queried from the model persistor. After a validity check of the models and forecasts, the request propagation algorithm is executed. The hyperparameter ϵ is specified in the predictor configuration, which is described in Table 5.5. As a result of this algorithm, we retrieve a list where all endpoints and services are mapped to their predicted number of requests within the next forecasting interval. This list serves as an input to the performance inference algorithm. By using the precomputed calculation order for acyclic topologies, we guarantee that the performance metrics can be determined iteratively. The current implementation does not support cyclic topologies, as they are not evaluated in this work. Possible approaches are discussed in Chapter 4. As a result of the performance inference algorithm, we retrieve the predicted performance values for the next monitoring intervals. For evaluation purposes, the predictions are stored together with the load forecast and calculated internal requests into a data structure which is labeled with the time at which the predictions were made. In production environments, the services would be now rated based on the predicted performance metrics according to a user-specified scheme. We execute this rating during the evaluation step to test different rating schemes on the same results. Our rating procedure is described in Chapter 6. The predicted values are finally transmitted to the prediction persistor, which stores the values for evaluation permanently.

Prediction Persistor. The prediction persistor is the main interface used for the evaluation in this work. The component has the responsibility to store all predictions made by the predictor component. It provides an interface where all predictions sorted by the time at which they were calculated can be queried. These values are compared with the measured performance values stored in the performance persistor during the evaluation of the framework.

5.3. Evaluation Modes

The PPP framework supports different execution modes in order to facilitate the evaluation process. An overview of these modes is shown in Table 5.6. In the *live* mode, every component of the PPP framework, Pinpoint as well as the test application is running. In this scenario, the performance and tracing data of the application are gathered by Pinpoint in real-time. This information are queried, processed, and persisted in regular intervals by the provider. The performance and propagation models are created and the predictions are computed in parallel. This mode is in particular intended to be used in scenarios

Table 5.6.: Evaluation Modes and Active Components

Mode	Active Components						
	Test Application	Pinpoint	Core	Databases	Provider	Trainer	Predictor
Live	✓	✓	✓	✓	✓	✓	✓
Record	✓	✓	✓	✓	✓		
Replay			✓	✓		✓	✓

where predictions need to be done online and available at runtime. For the evaluation process in this work, this mode comes with some disadvantages. First, we can train and evaluate only one type of performance model at the same time, which makes it hard to compare different approaches and find the best one. Moreover, there is an increased risk that an arbitrary error propagates through the framework and results in failures of single components or anomalous data, which finally causes the experiment to be invalid. Last but not least, we need a high amount of computing resources to guarantee that the test application, Pinpoint, and our framework do not influence each other and still have enough resources to work properly. For those reasons, we decided to split the evaluation process into several steps.

In the *record* mode, measurement data are recorded and processed but neither models nor predictions are created. Hence, the predictor, load forecaster, and trainer components of the PPP framework are not active in this mode and do not need to be deployed. Only the provider, core, and database components are mandatory. In this setting, Pinpoint gathers performance and tracing data from the test application and the provider queries and parses them. This has the advantage that we save a significant amount of computing resources and can record a trace that can later be used for the evaluation of different model types. A variant of the record mode is the *parse* mode, which enables the possibility to process data from Pinpoint, which have been recorded in the past. Thereby, also externally recorded traces can be used for evaluation. The counterpart of the record mode is the *replay* mode, which is used for training models and creating predictions on a recorded trace. Therefore, a starting and ending time of the trace has to be provided at the framework startup. The trace is replayed, which means that a virtual time is used within the framework and the models receive only training data that are known at this time. Thus, we can run different experiments and settings on the same data and compare the performance of different models in the same circumstances. In the replay mode, Pinpoint and the test application are not running. The functionality of the record and the replay mode together is equal to the live mode.

6. Evaluation

In this chapter, we describe how the previously described approach has been evaluated. Therefore, Section 6.1 explains the technical setup of our experiments. Section 6.2 describes our test scenarios and workloads, while Section 6.3 introduces metrics to quantify the quality of our approach.

6.1. Technical Setup

In this section, we describe the technical setup for our experiments. For the major part of the evaluation, we decided to use TrainTicket as it is a state-of-the-art microservice test application and can be monitored with Pinpoint. The complexity of our demo application gives us a wide range of evaluation options for our approach. In the following, we describe our evaluation setup and workflow, which is visualized in Figure 6.1. The TrainTicket microservices are deployed on a single machine. By setting resource constraints, we limit the maximum resource usage for every container. This is mandatory in our setting as we want to minimize the resource sharing between the services. Moreover, we observed that network connections are the limiting factors without setting CPU and memory constraints. This led to problems with the monitoring agents as they were not able to transmit their data. We assigned 0.6 cores[1] and 1GB RAM to every service, in non-critical scenarios the average CPU consumption of the services lay between 0.05 and 0.15 cores. The full hardware and software settings for our application server are shown in Table 6.1. In future work, we plan to deploy TrainTicket in a computing cluster. By setting deployment constraints, we can influence the placement of the services manually and analyze different configurations. Moreover, also the influence of service replications can be analyzed.

Those Java services, which need to be monitored, are started with an associated Pinpoint agent. In general, all Java services can be monitored using Pinpoint. However, in some scenarios, it could be useful to disable monitoring for specific services. Moreover, some TrainTicket components cannot be monitored as they are not written in Java. In Figure 6.1, Service C represents such an unmonitored service. The agent can be configured as desired for any application, e.g., by adjusting the sampling rates for regulating the monitoring overhead. In our evaluation, we try to monitor every request, hence the sampling rate is 1. However, we observed that in high load scenarios a small proportion of the tracing information is dropped and not available for our evaluation. All other components

[1] The indication x *cores* means that x out of one second computing time of one core is the maximum, which can be assigned to a service.

Table 6.1.: Hardware and Software Settings of the Application Server

Hardware	
CPU	Intel Xeon CPU E5-2640 v3
RAM	2x 16GiB DIMM DDR4 RAM
Disk	500GB HP HDD MB0500GCEHE
Software	
Operating System	Ubuntu 18.04.3 LTS
TrainTicket Version	30 Aug 2019, commit `fa1fc90` from repository [44]
Teastore Version	1.3.7
Pinpoint Version	1.8.4
Docker Version	18.09.7
Docker Compose Version	1.23.1

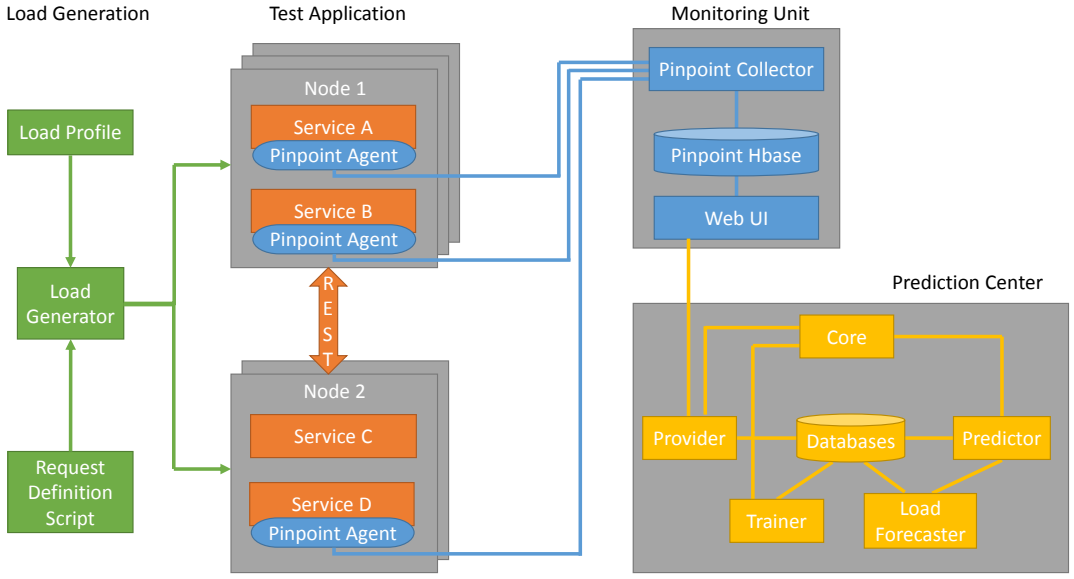


Figure 6.1.: Evaluation Setup

of the Pinpoint monitoring system, such as the Pinpoint Collector, Web UI, and Hbase storage run on a dedicated (virtual) machine and do not influence the test application. Before the start-up of TrainTicket, the IP address of the monitoring machine is set in the agent configuration. Similar to the monitoring unit, the performance prediction center will be deployed on a dedicated host and receives data from the Pinpoint Web component.

For the generation of different load profiles and request rates, we use the HTTP load generator described in Section 2.3. The load generator runs depending on the desired load intensity on one or more dedicated (virtual) machines and is able to use multiple threads on one machine to generate requests. We can vary the request definition script to simulate different user behaviors and stimulate selected services only. In combination with different load intensities, we are able to overload certain services and generate performance degradations. Finally, we are able to test different setups and dependencies by varying the placement of the services.

Figure 6.2.: Workload Overview for TrainTicket

The evaluation workflow starts with the generation of different load profiles and the export of them as CSV files. We further define different realistic user behaviors in request definition scripts (see Section 6.2). These two inputs are used as parameters for the load generator, which sends requests to the test application. Different performance metrics, such as response times and the number of exceptions, are measured by the Pinpoint agents. In our experiments, all Java services are instrumented with Pinpoint agents. The gathered data are sent to the Pinpoint Collector, which saves the data into the database. The Pinpoint Web UI can be used for a first plausibility assessment of the data. This workflow and the technical setup remains nearly unchanged for the Teastore, our second test application, which is used for a minor part of the evaluation. The Teastore application consists of fewer microservices than TrainTicket. This allows us to increase the resource limits. For our evaluation, we assigned 1 core and 4GB RAM to every service.

6.2. Test Scenarios

The major part of our evaluation is performed with the microservice test application TrainTicket, which has been described in Section 3.5. TrainTicket provides a large functionality in the area of online booking of train journeys, which reaches from essential functions like user login through to vouchers and complex queries. For our experiments, we defined a sequence of user requests, which is oriented towards real user behavior. In the following, we describe this sequence of calls in more detail, an overview is shown in Figure 6.2.

As the first step, we simulate the user login on the TrainTicket website. Therefore, two requests are needed. Prior to the actual login, where the user credentials, e-mail address, and password, are transmitted, a captcha image is generated. Before starting the experiments, we create an account and credentials for every single user in the thread pool of the HTTP load generator. This is necessary, as TrainTicket allows only one active session per account. If a thread enters the website with the same credentials as another one, the session of that user, which has entered the website first, is closed. After the login, the users are searching for train connections. Therefore, we defined a pool of eight routes, where each user picks one randomly. We further scripted that the first request of a user always results in an empty result list, which means that no trains are found on that route. This is possible as TrainTicket contains different train types, which operate along different routes. On the second try, the user receives a non-empty result list, as the correct train type is

selected for his route. Moreover, the probability that a user sends an invalid request can be specified. In this case, the customer enters an invalid station name, which leads to an exception in a backend service. Per default, this probability is set to 0.2. If a user performs an invalid request, he retries the call, which has again the specified failure probability. If this parameter is small enough, every user should receive a valid result list for his route in the long term.

In the next step, we simulate the ticket booking on the website. Therefore, several requests have to be sent. First, the contact data of the customer is queried from a database. For each booking, valid contact data have to be specified. Second, the user queries information about available travel assurances. We defined that 30 percent of the customers select an assurance for their journey. Third, the user queries the food offerings onboard his train. We defined that the customers order food on the train if it is available on their route. This is the case in three of our eight routes. Afterward, the actual ticket booking takes place where all this information is transmitted. Analogously to the route query, we defined that the first booking request fails because the train type is wrong. In contrast to the previous scenario, no exception takes place here. The response only contains the information that the booking has failed because the train could not be found. With the second try, every user performs a valid booking. In order to prevent overbooking of trains in high load scenarios, we delete the bookings for the trains periodically. After the ticket booking, the user pays his order and it is simulated that he collects his ticket and checks in at the station. In total, we thereby defined a realistic user behavior for that test application.

All in all, 57 endpoints across 26 microservices are used to process these user requests. Figure 6.3 shows the resulting call graph on the service level. We observe the relations described in our approach. Some services do not send any requests to other services, while other components receive queries from and send requests to many other services. Moreover, we see several cycles between different services, for example between `travel2` and `seat`. Figure 6.4 shows the same call graph on the endpoint level. Closely apposed nodes hereby mean that these endpoints belong to the same microservice. The position of the services is the same as in Figure 6.3. We can see that the described cycles disappear here, which validates our assumption of an acyclic graph on that level. By using the `randomize-users` flag of the HTTP load generator, we can now generate a realistic request mix, which stresses the application to an adequate extent. In order to force exceptions and performance degradations, we vary the load intensity and stimulate some single endpoints and services directly.

A minor part of our evaluation is performed on the Teastore application. For this application, we use a predefined load script[2]. This script encapsulates the behavior of a customer, who searches for different items in the webshop but does not buy anything. This results in the fact that the backend databases are not modified. In the following, we describe the user requests more detailed. First, the user accesses the main page of the webshop and then logs in with his credentials. In the second step, he searches for different random items in the shop and adds two in his virtual shopping cart. Afterward, he accesses his profile and finally logs out again. The detailed experimental settings are described in Chapter 7.

6.3. Evaluation Metrics

In this section, we define the metrics which are the base of our model evaluations and comparisons. As described in Chapter 4, we use regression models to predict continuous performance metrics. In terms of this work, we choose the average response time to

[2]Download available at: `https://github.com/DescartesResearch/TeaStore/tree/master/examples/httploadgenerator`

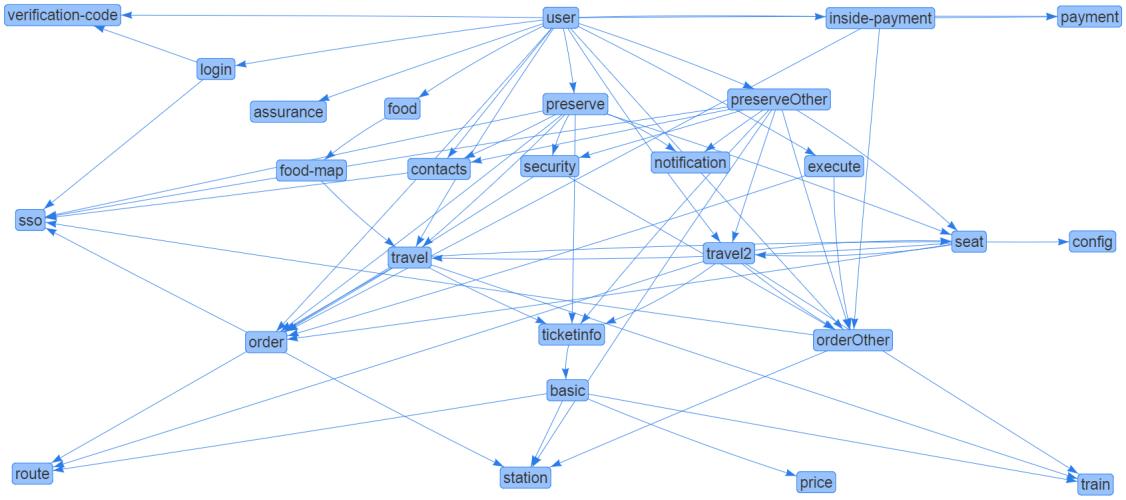


Figure 6.3.: Service Call Graph for Described Workload

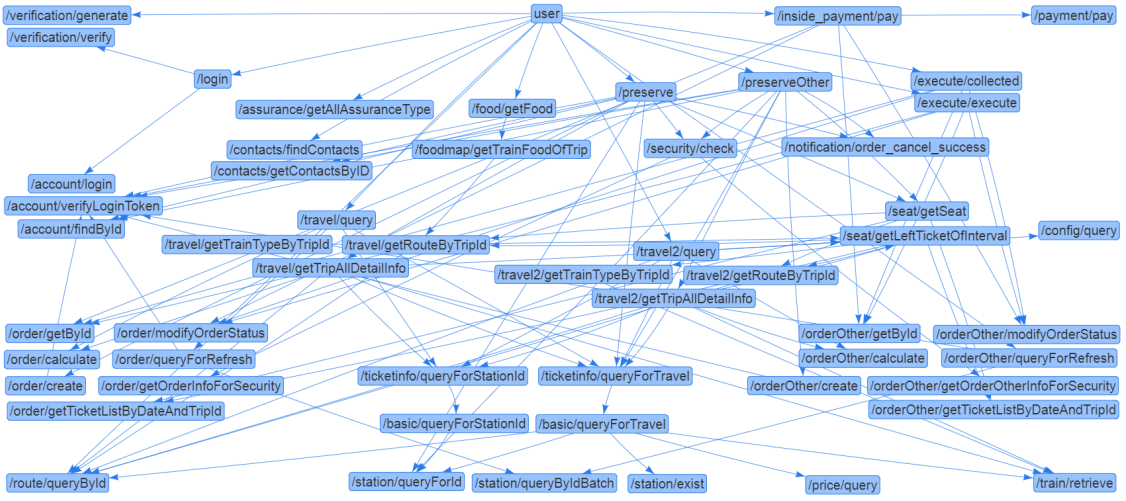


Figure 6.4.: Endpoint Call Graph for Described Workload

requests, which arrive in a dedicated measurement interval, as our performance metric. The PPP framework supports beyond that also the metrics maximum response time and number and proportion of exceptions. Based on the numeric values of the performance metrics, we grade the state of an endpoint and assign one of the ratings green, yellow, or red. In this work, this assignment is based on the average response time only. Therefore, the response time of an endpoint under low load is considered as a base value for the green rating class and the limits for the yellow and red ratings are set accordingly. We discuss the influence of these limits and give recommendations on how to set them in Chapter 7. Our rating schemes for every endpoint are given in Appendices B and C. In practice, multiple metrics might be included in the calculation of the rating and the schemes and limits might be part of service level objectives. The PPP framework assigns the ratings based on the predicted response times. Analogously, the measured data are labeled based on the measured response times. Hence, we obtain a set of true and predicted labels for every endpoint and can apply standard classification metrics. These are based on the confusion matrix, which contains the amounts of true positives (TP), false positives (FP), true negatives (TN) and false negatives (FN) for every rating class.

First, we define those metrics, which assess the prediction quality on the dataset of one dedicated endpoint. In our evaluation, we have one predicted and one measured label for an endpoint per measurement interval. Based on this set, we calculate the *precision* for every rating class, which is a measure of the prediction quality. It describes the proportion of the true positives on the set of all predicted positives. Hence, the precision reaches its maximum value of one, if all predicted positives are true positives. Equation 6.1 shows how to calculate the precision p_x for a rating class or label x.

$$p_x = \frac{TP}{TP + FP} \tag{6.1}$$

In addition to the precision, the *recall* is another important classification metric. In contrast to the precision, the recall takes quantitative aspects of the results into account. It describes the proportion of the true positives on the set of all measured positives. Hence, the recall reaches its maximum value of one, if all measured samples of one class have been predicted. This value is especially interesting in our use case for the red ratings, as we can evaluate how many of the measured performance degradations we are able to predict. Equation 6.2 shows how to calculate the recall r_x for a rating class or label x.

$$r_x = \frac{TP}{TP + FN} \tag{6.2}$$

In practice, both precision and recall are of importance as one wants to predict as many true positives as possible without generating too many false positives. In our case, we want to predict as many performance degradations as possible, but, if we give too many false alarms concurrently, the benefit of our approach is limited. Moreover, in our setting a false positive rating in one class is a false negative rating in another class simultaneously. Consequently, we need a metric that combines precision and recall. We use the *F1 score*, which is the harmonic mean of precision and recall and gives equal weights to both metrics. Equation 6.3 shows how to calculate the F1 score F_x for a rating class or label x.

$$F_x = \frac{2 p_x r_x}{p_x + r_x} = \frac{2TP}{2TP + FP + FN} \tag{6.3}$$

The previously defined metrics support only binary classification. In our case, we have three potential ratings/labels for one endpoint and measurement interval: green (G), yellow

(Y), and red (R). To grade the overall prediction quality of our approach on one endpoint, we use the macro-averaged F1 score. It represents the arithmetic mean of the F1 scores for the three labels green, yellow, and red. With this, we ensure that every rating has an equal influence on the resulting score. This is important in our use case, as we are interested in accurate predictions for all rating classes. Another advantage of the macro-averaged F1 score is that it is independent of the number of samples for every label and does not require a balanced dataset. This is suitable for our application case, as we expect both in our evaluation and in practice imbalanced datasets. More precisely, we expect that at an endpoint has a good response time and stays in a non-critical state most of the time. Performance degradations are anomalies and consequently yellow and red ratings appear seldom in the dataset. This results in an imbalanced dataset. Equation 6.4 shows how to calculate the macro-averaged F1 score F for an endpoint based on the F1 scores F_G, F_Y and F_R for the ratings green, yellow and red.

$$F = \frac{1}{3} \cdot (F_G + F_Y + F_R) \tag{6.4}$$

In our evaluation, we provide the overall accuracy value of the prediction for one endpoint in addition to the macro-averaged F1 score. The overall accuracy is a simple and comprehensible metric, which represents the proportion of the correctly predicted states on all predicted states independently of the rating class. For example, an accuracy of 0.6 means that 60% of all predicted labels are equal to the measured ones. In our use case, we can further say that the framework would issue correct ratings in 60% of the time. The accuracy reaches its maximum value of one if all predicted labels equal the measured ones. Equation 6.5 shows how to calculate the overall accuracy A for one endpoint. Hereby, TP_x and FN_x describe the true positives and false negatives of the rating class x.

$$A = \frac{TP_G + TP_Y + TP_R}{TP_G + TP_Y + TP_R + FN_G + FN_Y + FN_R} \tag{6.5}$$

All previously described metrics are defined for the data of one endpoint only. For our evaluation, we further need global metrics, which assess the performance of our approach on all endpoints and thus on the application as a whole. We stick to the combination of the accuracy, as a simple and comprehensible metric, and the macro-averaged F1 score, which takes the results related to all rating classes into account explicitly, here as well. To determine these measures, we first calculate the confusion matrices for all endpoints. In the next step, we sum the values of true positives, false positives, true negatives, and false negatives up and obtain the global confusion matrix. This matrix contains $e \cdot n$ samples overall, where e is the number of endpoints and n the number of measurement intervals, which is also the number of samples of one endpoint. Based on the global confusion matrix, we calculate the measures analogously to the definitions above. Hence, the global (macro-averaged) F1 score F_{global} is calculated as shown in Equation 6.6. Hereby, $\sum TP_x$, $\sum FP_x$, and $\sum FN_x$ represent the summed values of the true positives, false positives, and false negatives of all endpoints.

$$\begin{aligned} F_{global} &= \frac{1}{3} \cdot (F_{G,global} + F_{Y,global} + F_{R,global}), \\ \text{where } F_{x,global} &= \frac{2 \sum TP_x}{2 \sum TP_x + \sum FP_x + \sum FN_x} \end{aligned} \tag{6.6}$$

Analogously, the global accuracy is calculated with the following formula:

$$A_{global} = \frac{\sum TP_G + \sum TP_Y + \sum TP_R}{\sum TP_G + \sum TP_Y + \sum TP_R + \sum FN_G + \sum FN_Y + \sum FN_R} \tag{6.7}$$

7. Results

In this chapter, we describe the results of our evaluation. All in all, we discuss five scenarios, where four describe different workloads and their influence on the performance of the TrainTicket application and the last one describes the results of our framework on the TeaStore application. Each section provides a description of the scenario and discusses different aspects of our approach. Section 7.6 points out the findings of all scenarios and summarizes the strengths and weaknesses of our approach.

7.1. Scenario 1: Periodic Load

Scenario Description. In the first scenario, we evaluate the performance of our framework on the TrainTicket application, which is stressed with a regular periodic load. The load intensity varies from 3 requests per second (rps) up to 22 rps. In preliminary tests, we observed that some services crash or have a response time well above ten seconds with our hardware restrictions under a load of 24 rps. The trace for this scenario has a total length of about 60 minutes. The peak loads occur 10 times during this period. The measurement data in all scenarios are gathered and aggregated in intervals of five seconds. Figure 7.1 visualizes the rating of all endpoints stimulated by our workload at the fourth load peak, which appears 1345 seconds after the beginning of the trace. The positioning of the endpoints and the grouping of the services in similar to Figures 6.3 and 6.4. We can see that multiple endpoints and services experience performance problems and some issue performance warnings. For this evaluation, we rate the state of an endpoint based on its average response time only. In practice, more metrics could be included the calculate the rating. The rating schemes and thresholds can be found in Appendix B. The rating thresholds have been set as follows. For all endpoints which have an average response time in the low load scenario greater or equal to 30 milliseconds, the yellow rating threshold is about 1.5 to 2 times and the red rating threshold is about 3 times the average response time under low load. For all other endpoints, we defined a default threshold of 40 ms for the yellow rating and 80 ms for the red rating. Otherwise, the ranges for the different rating classes would be too small and not practicable.

The evaluation and figures in this and the following scenarios focus on the data and performance of the endpoint `/travel/query`, which has been chosen as a representative example within the TrainTicket application. This endpoint is managed by the service `ts-travel-service` and is called by the user whenever he searches for a train connection between two cities on the website. Hence, the endpoint needs the origin and destination station,

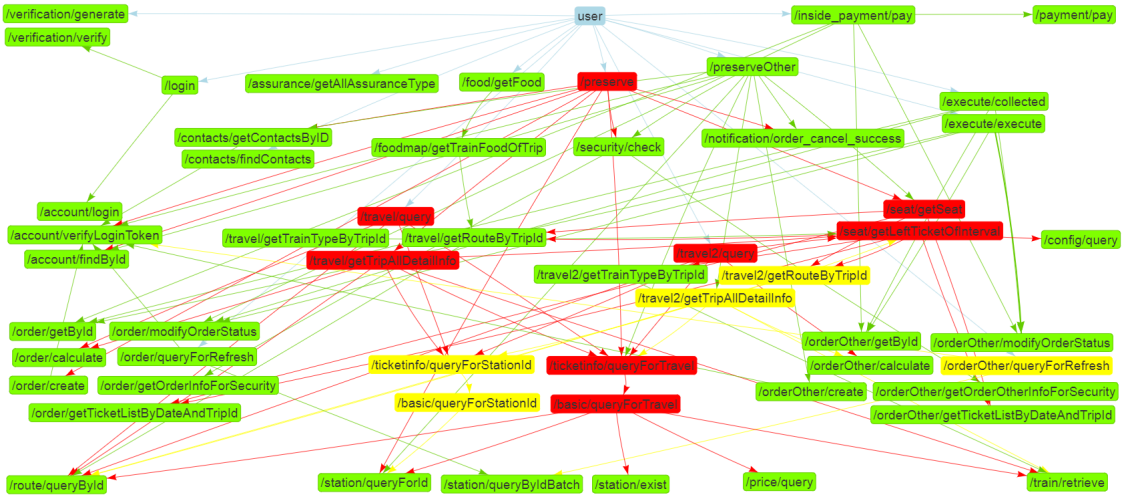


Figure 7.1.: Situation of All Endpoints Stimulated by the Workload

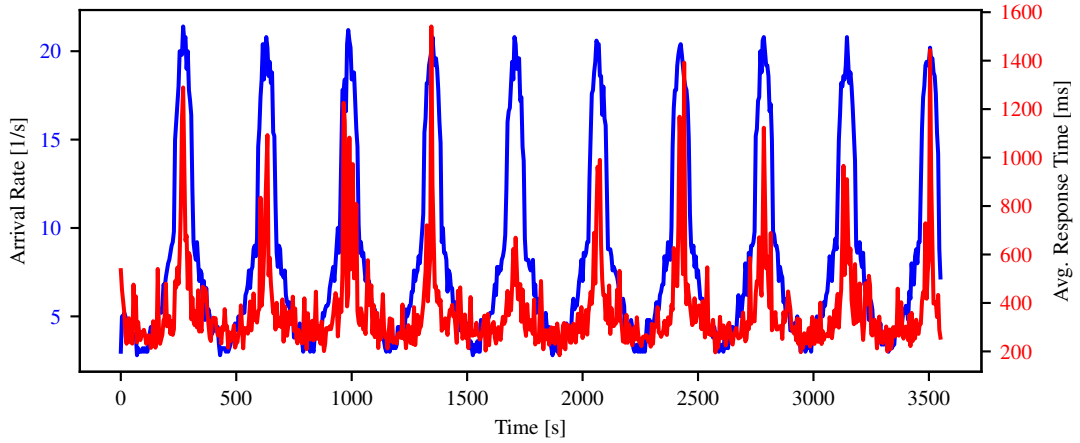


Figure 7.2.: Load and Average Response Time of the Endpoint `/travel/query` over Time

as well as the travel date as parameters. As a response, the user receives a list of train connections that matches his input. We choose this endpoint as the running example for our evaluation for a variety of reasons. First, this endpoint would be directly called by the user also in real-world scenarios and would have a direct influence on the user experience. Moreover, the function of this endpoint is easy to understand. From a software architecture and performance point of view, it is suited for evaluating our framework as it depends on many other services and its response time is sensitive to load variations. Furthermore, the endpoint is stateless and no login is required to access it, which makes testing easier. The average response time of `/travel/query` in a low load scenario under the given hardware constraints varies between 250 and 300 milliseconds. With the maximum load, the average response time increases up to 1500ms. The correlation between the load intensity and the response time of this endpoint is shown in Figure 7.2. For this scenario, we defined that this endpoint gets a green rating if it has an average response time up to 400ms and a yellow rating up to 600ms. If an average response time above 600ms, which is twice the low load response time, is measured, the endpoint gets a red rating. Given this rating scheme, the endpoint is rated green in 534, yellow in 127, and red in 51 out of 712 total measurement intervals.

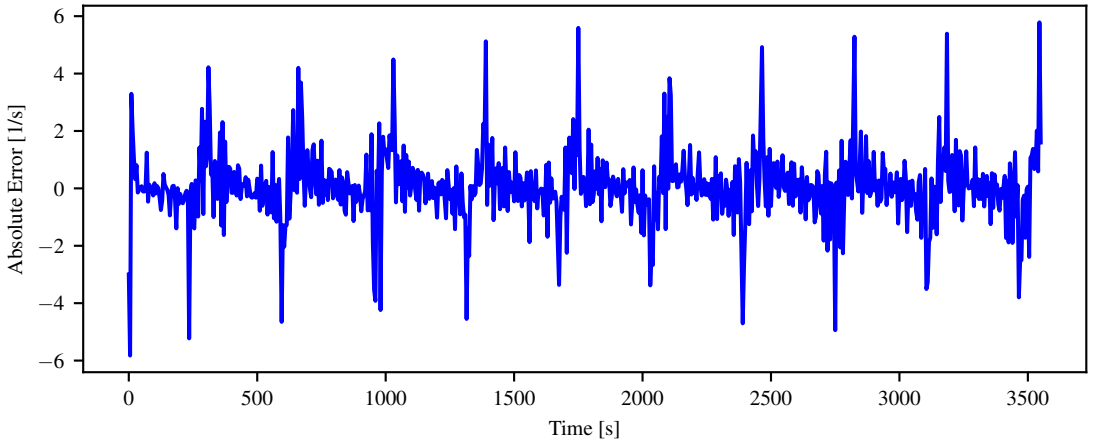


Figure 7.3.: Absolute Arrival Rate Forecast Error of the Endpoint `/travel/query` over Time

Load Forecast Quality. Before assessing the quality of our performance predictions, we want to evaluate the quality of the load forecast and how well our propagation model can predict the number of internal calls caused by the user requests. An accurate load forecast and request propagation are important for the performance prediction as the features for the machine learning model are derived from them. As our load forecaster, we selected the `SimpleFeedForwardEstimator` from the GluonTS library, which is based on a multilayer perceptron model (MLP), a feedforward neural network. We choose this estimator because it has a much lower training and prediction time than other estimators from the library, from which the majority are based on recurrent neural networks. Moreover, it produces results of equal quality in our cases. We use two epochs for the estimator training, this results in a training and prediction time which is acceptable also in the live operation mode. The estimator receives the time series of all endpoints, which received user requests, containing all values recorded in past intervals. Given this input, the estimator calculates a forecast for the given forecast horizon. GluonTS is a probabilistic forecaster and therefore returns an interval, within its range the next value will be most likely. As our framework requires a single value, we always extract the mean value of the forecast. However, in future work, an extension could be developed, where the framework receives minimum and maximum value of user requests to calculate a range of predicted performance values as well.

Figure 7.3 shows the absolute errors over time for the arrival rate of the endpoint `/travel/query` with a forecast horizon of one interval (5 seconds). We can see that the forecast has a good quality in general and only single values have deviations of 3 rps or more to the measured value. These high differences occur whenever the load raises or decreases strongly before or after a peak load. In all other cases, the error sways around its optimal value of zero. Positive and negative deviations occur equally, which means that the arrival rate is neither constantly under- nor overestimated.

Propagation Model Quality. Given this load forecast, we evaluate how well our request propagation model is able to predict the internal calls of the test application. Therefore, we stick to our running example `/travel/query` and look at the part of the propagation model, which contains those services and endpoints, which are called in the succession of a user request to `/travel/query`. Figure 7.4 shows this model section. The blue boxes hereby represent an endpoint. Every endpoint is part of a service, which is visual-

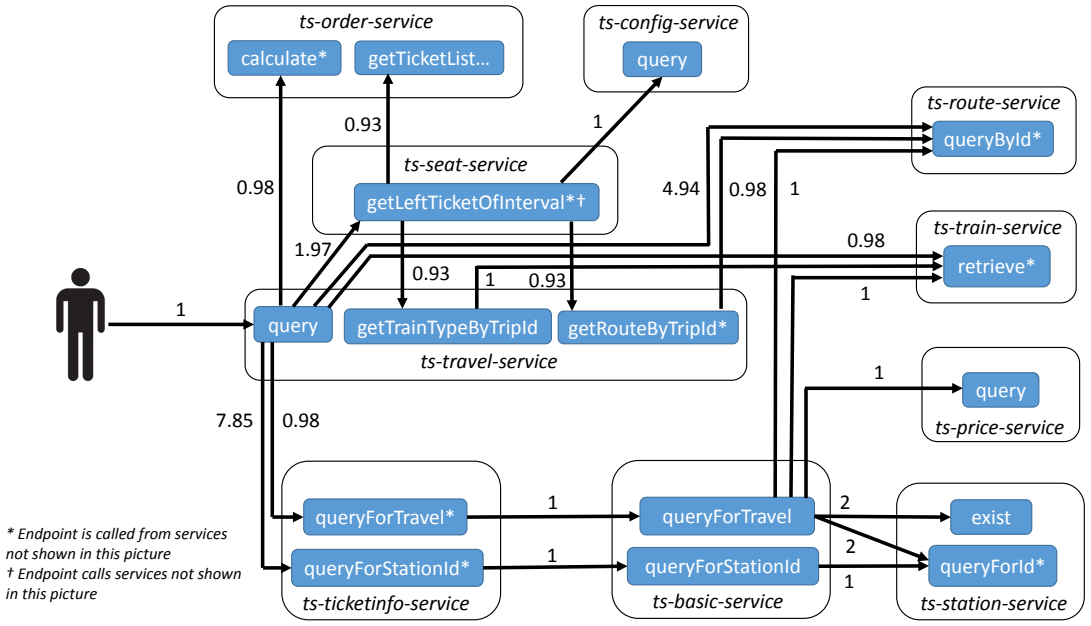
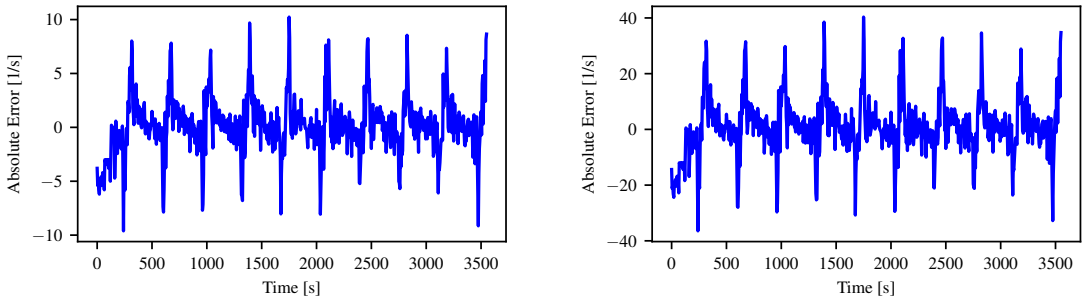


Figure 7.4.: Section of Propagation Model Related to a User Call to `/travel/query`

ized by black frames. An edge between two endpoints signalizes a request flow between these endpoints. Each edge is labeled with the parameter c from the propagation model, which represents the number of requests sent per incoming request at the origin endpoint. The values in the figure are taken from the propagation model calculated 1500 seconds after the beginning of the trace. We can see that different values of c appear in the model. Many edges are labeled with the value one, which can be interpreted as request forwarding. These kind of communication pattern can be seen for example between the services `ts-ticketinfo-service` and `ts-basic-service`. In this case, the endpoints of `ts-ticketinfo-service` forward every request to the endpoints of `ts-basic-service` with the same name. If the parameter c is smaller than one, not every request causes a new request to be sent. This happens due to a variety of reasons. First, the generation of a new request can depend on a condition specified in the application logic. Examples for this case are given by the requests sent from `ts-seat-service` to `ts-travel-service`. The endpoints of `ts-seat-service` are called from another service named `ts-travel2-service` as well. This service manages other train types than `ts-travel-service`. `ts-seat-service` can distinguish the trains by their train id and calls only the service, which is responsible for the current train type. A side effect here is that the propagation model can also show some business values. In this example, the model states that 93 percent of the calls to `ts-seat-service` are related to train types managed by `ts-travel-service`. As a consequence, the users are obviously more interested in these trains. However, to perform such an analysis, application-specific knowledge is needed. Another reason for low values of c can be exceptions or errors, which influence the information flow as well. For example, an exception causes that other requests, which would be sent later in the processing of a user request, are not sent, and, as a consequence, the value of c is lowered. Last but not least, also measurement errors or numeric deviations can influence the value of c. For example, as described earlier, we observed during our tests that Pinpoint is not able to trace all requests when the arrival rate is high. Such effects can also affect c especially if c is not constant.

High values of c mean that an incoming request causes multiple new requests to be sent. One example of this is the relationship between `ts-basic-service` and `ts-station-service`. Here, the value of two means that every incoming request causes two requests



(a) Endpoint `/price/query` (b) Endpoint `/train/retrieve`

Figure 7.5.: Absolute Errors of Arrival Rate Forecasts for Selected Backend Services

to `ts-station-service`. This is really reasonable in this case, as for querying a travel always two stations have to be checked: the origin and destination station. Another example for this phenomenon can be seen in the relationship of `ts-travel-service` and `ts-ticketinfo-service`. We can see that for querying a travel, an average of 7.85 different stations is queried. This could be also useful information for developers, for example, to evaluate different search algorithms. Moreover, a significant change of this value over time can signalize a change in the user behavior, for example, if the users request different routes. All in all, we conclude that the propagation model contains architectural and business information, which promotes the understanding of the application itself and the way it is used.

In the following, we want to discuss how well the linear propagation model predicts the load intensity on backend services with given load forecasts. Therefore, we consider the endpoints `/price/query` and `/train/retrieve` as examples. The absolute errors of the arrival rate predictions with a prediction horizon of one interval are shown in Figure 7.5. The graphs have the same shape as the one in Figure 7.3. Concretely, only single values have a high deviation from the optimal value of zero, while the majority of predictions are near to the measured value. The high deviations occur whenever the load increases or decreases strongly. Hence, we can see that the propagation model is able to predict the internal calls well. However, the errors of the load forecast propagate within the model. This can be also concluded from the analysis of the relative errors. The maximum relative errors of the arrival rate prediction for the frontend `/travel/query` (75%) is nearly equal to the relative errors of the backend services (78%). We conclude that the linear propagation model is well suited for this application and workload and the prediction quality mainly depends on the quality of the user forecast. We discuss this dependency later in more detail. To improve the accuracy of the arrival rate prediction, minimum and maximum values might be inserted as described earlier. As a result, we would obtain a range of arrival rates rather than a single value.

Evaluation of Different Machine Learning Models. For our performance inference models, we tested four different machine learning model types: the k nearest neighbor algorithm (KNN), bayesian ridge regression, support vector regression (SVR), and random forest regression. We used the implementations provided by the Python library `scikit-learn`. To optimize the hyperparameters of the algorithms, we performed a grid search. Therefore, we split the time series into six parts with a length of about ten minutes and execute a cross-validation based on the scheme of an out-of-sample forecast evaluation [51]. As the scoring function and optimization metric, we use the macro-averaged F1 score, which has been described in Section 6.3. The resulting parameters and all tested

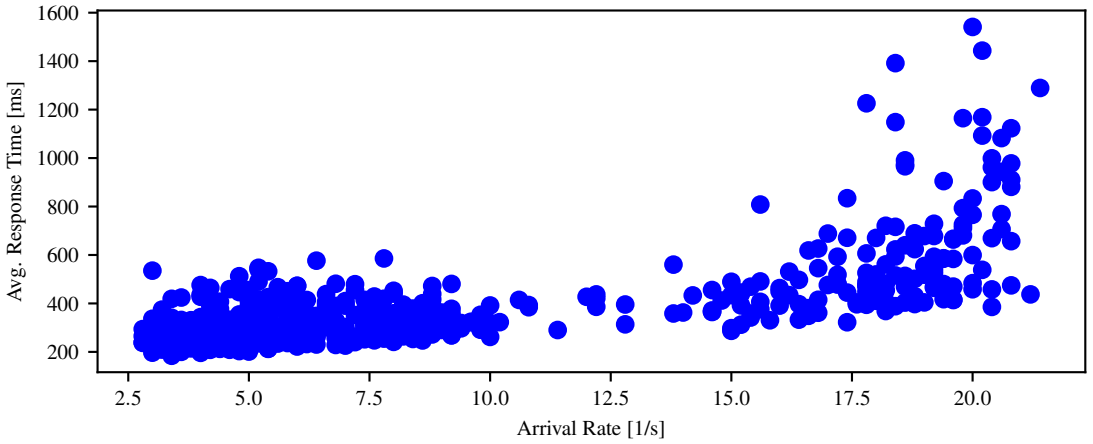
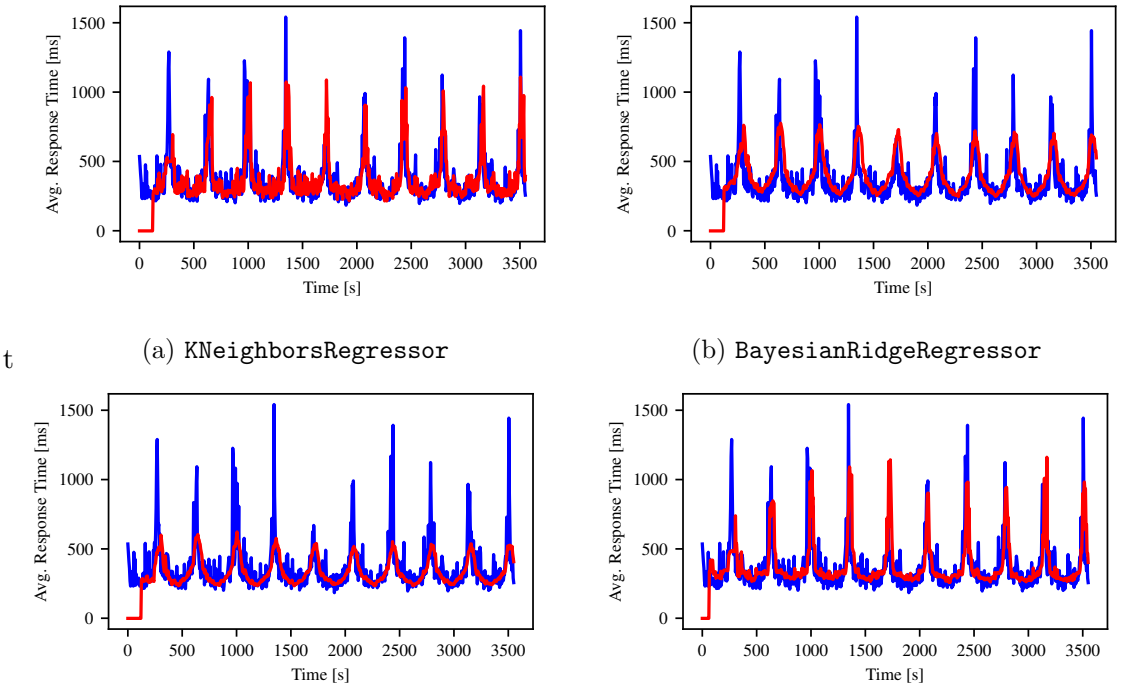


Figure 7.6.: Average Response Time as a Function of the Arrival Rate on the Endpoint `/travel/query`

configurations for every model optimized for the endpoint `/travel/query` are shown in Appendix A. During the hyperparameter search, we took also time series of other endpoints into account. However, it turned out that nearly the same optimal parameters had been found because the training data were of similar nature.

The dependency between the average response time and the arrival rate of the endpoint `/travel/query` is shown in Figure 7.6. The arrival rate to an endpoint itself is one of the features used for predicting its performance metrics. In this scenario, this feature is the main reason for performance degradations. This can be seen in the figure, which foreshadows the shape of a classical latency curve. In the range up to 17 rps, the average response time increases slowly and near linearly, while a higher arrival rate causes a much faster increase of the average response time. Besides this, we see that the majority of training data is located in the non-critical area below 10 rps, while only a few training data are available in the high load area. This imbalance appears in real-world applications as well. The problem is even more challenging for our framework and models especially at the beginning of the traces, where no training data is available and especially none in the critical area.

In the following, we want to evaluate the results of different machine learning models. The models have been retrained every 60 seconds and capture the training data known at the training time. Any pretraining or advance information was not needed and has not been used. Moreover, we neglect any further influencing factors at runtime, for example, the transmission delay of the models and training data. This investigation is enabled by the replay mode of the PPP framework. Figure 7.7 shows the predicted courses of the average response time for different model types for the endpoint `/travel/query` in comparison with the measured data. Table 7.1 shows the quality of the derived ratings for every model. Therefore, the values of the true positives (TP), false positives (FP), and false negatives (FN) for every rating class as well as the metrics accuracy and macro-averaged F1 score are listed. In general, we can see that all models are able to approximate the shape and periodicity of the measured graph. Moreover, all models are not able to predict the first increase and peak of the response time because no training data in this area is known at this time and the load forecast is not good as well because the time series has only data points. Similarly, we see that all models overestimate the response time shortly after the first peak, which leads to false red ratings, and adapt their predictions after this initialization phase.



(a) `KNeighborsRegressor` (b) `BayesianRidgeRegressor`

t

(c) `SVR` (d) `RandomForestRegressor`

Figure 7.7.: Prediction Results (red) and Measured Data (blue) for Different ML Models

The prediction of the `KNeighborsRegressor` is characterized by a higher noise compared to the other model types. The reason for this is that the hyperparameter k, which determines the number of neighbored data points consulted for prediction, has been set to two, which is a really small value. As a result of this, the estimator is strongly oriented towards the training data. In this case, the prediction is the average value of the two nearest known samples. This is beneficial in our application case, as nearly equal feature vectors yield nearly equal response times. The figure shows that the amplitudes of the peak loads are estimated well compared to other model types. In the non-critical areas, the prediction is really noisy at the beginning of the trace but the graph gets smoother over time when more training data are available. An advantage of the KNN model is the small training time and model size, which makes it applicable in live scenarios and reduces the computation power. The overall performance metrics accuracy and F1 score show that the KNN produces results of medium quality compared to its competitors.

The Bayesian regressor is also a relatively lightweight linear prediction model. Compared to the KNN, the prediction graph of the Bayesian regressor is much smoother. The course of the curve fits the one of the measured data qualitatively. However, the amplitudes are not as prominent as the ones of the KNN or random forest prediction. Moreover, the Bayesian regressor predicts slightly higher values of the response time than the measured in the non-critical area. This leads to a higher number of false yellow ratings and an increased false negative value of the green ratings. A similar effect can be seen between the yellow and the red rating classes. The estimator issues far more performance alerts than the other model types, which leads to an increased true positive value on the one hand but a higher false-positive value in the red class on the other hand. All in all, this overestimation leads to the worst accuracy in the competition. However, the macro-average F1 score is the second-highest in this scenario. The reason for this is that the model types receive much smaller F1 scores in the red rating class.

Table 7.1.: Results of Different Machine Learning Models

Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
k Neighbors	486	71	48	24	62	103	26	43	25	0.753	0.517
Bayesian	431	**44**	103	37	94	90	**37**	69	**14**	0.709	0.537
SVR	**491**	68	**43**	**59**	92	**68**	1	**1**	50	0.774	0.454
Random Forest	**491**	71	**43**	32	**50**	95	31	37	20	**0.778**	**0.574**

The SVR receives the smallest F1 score in the competition. This is because it underestimates the peak response times clearly. The figure shows that it is able to predict an increase in the response time but the amplitude is significantly smaller than the measured value and the values of the other estimators. As a consequence, far more yellow ratings are issued than red ones. The SVR only issues two performance alerts in this scenario, while all other model types issue at least 68. Moreover, the SVR rates the condition of the endpoint in 151 intervals as yellow, which is the highest number in the competition. With this behavior, the estimator receives the second-highest accuracy in the competition with a small distance to the highest score. The reason for this is that the majority of data points lie in the green or yellow area and the SVR receives high true positive values in these rating classes. However, the macro-averaged F1 score is much lower compared to the other model types because the SVR receives a very low score in the red rating class. All in all, this behavior is counterproductive in our use case, as the estimator is not able to predict performance degradations and give correct alerts.

The random forest regressor performs best in this scenario. The peak response times are estimated as well as by the KNN, but the predictions in the non-critical areas are much less noisy. All in all, the prediction quality is balanced across all rating classes, which leads to the best accuracy and F1 score in the competition. When using a random forest estimator in a realtime use case, one has to be aware of an increased model size and training time when using a large ensemble of decision trees. A good balance between the computational effort and prediction quality has to be found. In the following, we perform a further investigation of the results from the random forest estimator, as it makes the most accurate predictions in this scenario. But it has also been shown, that the KNN and Bayesian estimators can be good alternatives in use cases, where training time and computational resources are critical factors.

The previous results refer to the endpoint `/travel/query`. Besides that, we analyze the prediction quality on the application as a whole. Therefore, Figure 7.8 shows global quality measures for all model types together with the measures of primitive classification approaches. These measures allow us to assess the overall prediction quality of the different model types by taking data from all endpoints into account. We see that the values for both the unfiltered and filtered measures of our model types are greater than the ones achieved by the primitive approaches. The high accuracy of the all green classifier, which issues always green ratings, shows that the vast majority of measured response times are in a non-critical area and yellow or red ratings are seldom. The reason for this lies in the nature of the dataset. The majority of the endpoints in the application do not experience any performance degradation in this scenario. Their average response times remain nearly constant all the time and, so, their rating does as well. This makes the prediction task much easier and yields a strong increase in accuracy compared to the values of `/travel/query`.

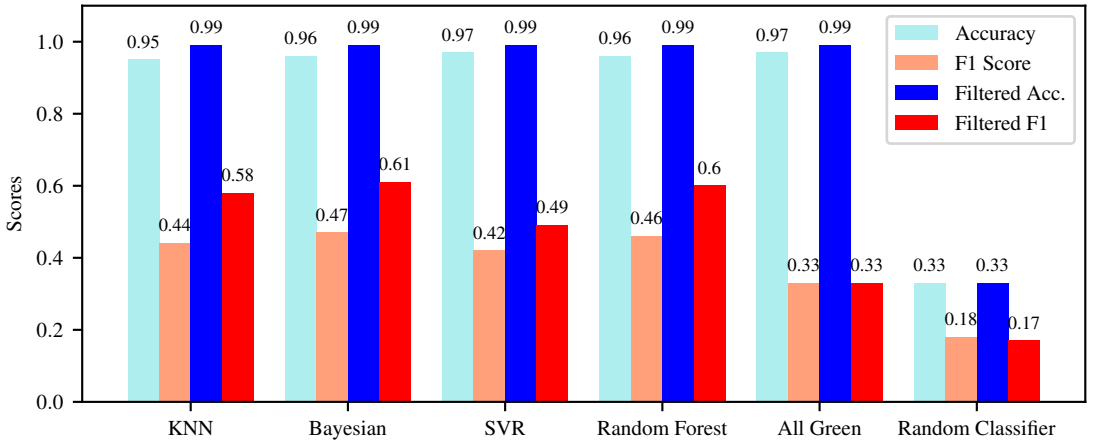
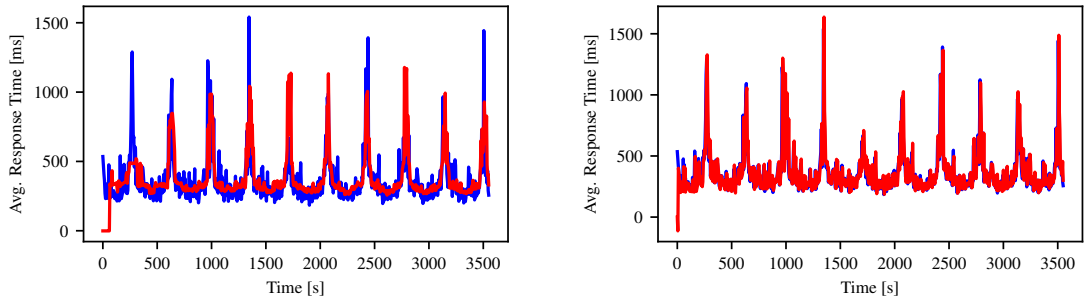


Figure 7.8.: Global Prediction Quality Measures of Different Model Types

Besides that, many endpoints of the application experience only a small arrival rate most of the time. This leads to the fact that only a few requests are included in the calculation of the average response time. This makes the average response time prone to outliers and the measured rating of this endpoint relates to only a few requests. Such outliers with a small arrival rate should be ignored in practice as the rating in such cases is not meaningful. Therefore, we change the labeling of the measured and predicted values. In all intervals, where the measured arrival rate at an endpoint is smaller than 10 rps, the measured rating is green independently from the measured average response time. Analogously, the PPP framework assigns a green rating to an endpoint, if the predicted arrival rate at this endpoint is lower than 10 rps. This can be implemented by a simple conditional statement and is a fair adjustment also for evaluation purposes, as the models still have to predict the arrival rate correctly. The figure shows that this adjustment causes an increase of the accuracy, but especially of the F1 score. This shows that many of the falsely predicted yellow and red ratings in the unfiltered case are caused by single outliers and a small arrival rate. Hence, we conclude that the filtered global measures are more meaningful for our evaluation. We stick to these measures in the following scenarios as well because their datasets show similar characteristics.

However, we see that the F1 score does not rise as strongly as the accuracy compared to the values of `/travel/query`. This is because the global F1 score does not take the imbalance of the dataset into account. In our case, we have many samples and a high score in the non-critical green rating class and a few samples and a lower score in the yellow and red rating classes. This applies to all endpoints, which have varying response times. Consequently, wrong predictions in the yellow or red rating class are penalized sharply by the global F1 score. However, the global scores confirm our results from the endpoint `/travel/query` qualitatively. We see that the random forest model and the Bayesian model receive the highest F1 score in the competition. Furthermore, the SVR has a higher accuracy but a lower F1 score compared to the random forest model. As discussed earlier, this is because the SVR predicts nearly all of the green ratings, which are the majority in the dataset, correctly. All in all, the high accuracy shows that our models are able to predict the performance correctly for a large number of services and endpoints. Moreover, they provide significantly better predictions than primitive approaches. The global scores include results from the whole application, but nevertheless one has to take the special role of endpoints like `/travel/query` into account, which are a direct measure for the user experience as their response time is directly noticeable by a customer.



(a) Random Forest Model with Optimal Input

(b) GluonTS Time Series Forecast

Figure 7.9.: Predictions (red) and Measured Data (blue) for Best Case Prediction and Time Series Forecast

Table 7.2.: Theoretical Best Case vs. GluonTS Simple Forecast

Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
GluonTS	473	58	61	**49**	75	**78**	28	**29**	23	0.773	0.599
Random Forest with optimal input	**504**	**56**	**30**	37	**44**	90	**33**	38	**18**	**0.806**	**0.606**

Best Case Analysis and Comparison to Time Series Forecasting. In the following, we want to discuss how our models and framework perform in the best case. As the optimal case, we consider the situation when the models receive an exact load forecast. As discussed earlier, we are able to construct a better input for our propagation and performance inference models, if we receive an accurate load forecast. We have seen in the previous section that the random forest estimator performs best in this scenario. Hence, we utilize the same models again, but now use the measured load instead of a load forecast for the feature construction. This can be interpreted as a prediction with a horizon of zero and enables the possibility to evaluate the prediction quality separated from the load forecast quality. The results of this investigation are shown in Figure 7.9a and Table 7.2. It is shown that this changed input leads to an increase in the accuracy of about three percent and the F1 score rises over the value of 0.6. The remaining deviations are caused by the fact that at the beginning of the trace only a few training data are available and the predictions at this time have high variabilities and uncertainties. Furthermore, due to the lack of training data the first peak of the curve can still not be predicted well.

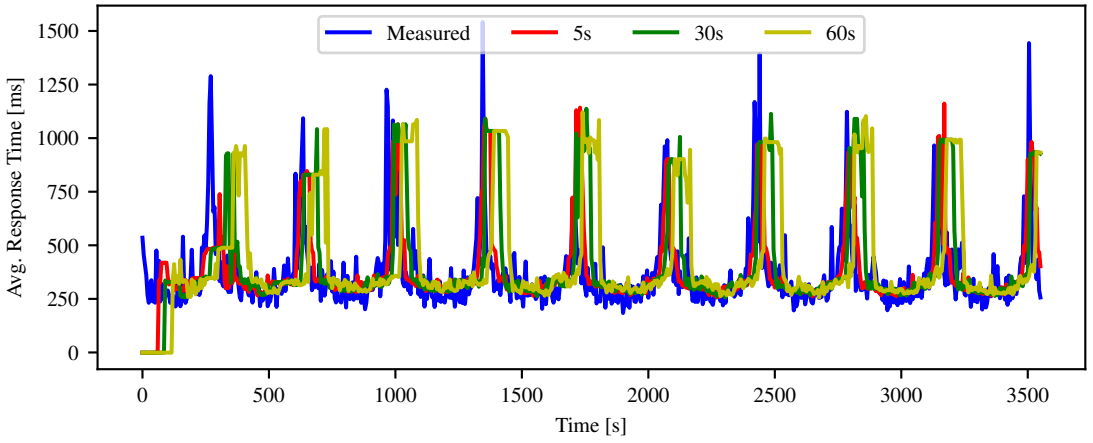
In the following, we want to compare these prediction results to the performance of a time series forecaster, which takes only characteristics of the time series, like trends and seasonal patterns, for his estimations into account. Therefore, we use the `SimpleFeedForwardEstimator` from the GluonTS library again. We use the time series of the average response times of the endpoint `/travel/query` as the input for the forecaster, whose task it is to predict always the next data point. The results of this forecast are shown in Figure 7.9b. We can see that the forecaster is also able to approximate the course of the curve well and gives better numeric predictions than our models. However, when consulting the metrics accuracy and F1 score, the time series forecast performs slightly worse than the random forest model. The reason for this is that the forecaster captures the noise in the non-critical areas, which leads to overestimations. Consequently, the estimator has more

false-positive yellow ratings and fewer true positive green ratings. This yields a smaller accuracy and F1 score compared to the random forest model.

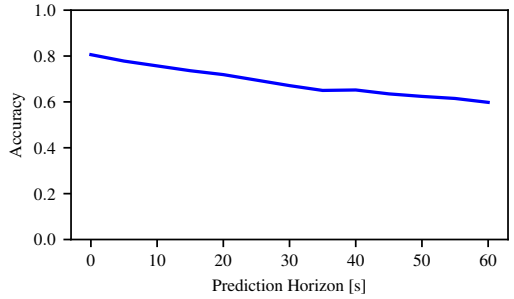
However, the time series forecaster gets a higher F1 score than all models shown in Table 7.1. Hence, it can be concluded that in this scenario our approach does not deliver the hoped-for improvements. Moreover, it needs a higher data preprocessing effort. The reason, that the time series forecaster performs equally well, lies in the nature of this test case. Here, the decisive factor for performance degradations are periodically appearing load variations exclusively. Other possible triggers for performance degradations do not occur here. This one-dimensional relationship can be captured well by the time series forecaster. In general, we conclude that in the current version of our framework the prediction quality depends strongly on the load forecast. Even if we receive a really good forecast for a specific load profile, a time series forecaster can likely make also good performance forecasts because the time series of the load intensity and performance metrics have equal characteristics. This behavior can be seen in Figure 7.2 also.

The full potential of our approach is developed, if a higher number of features and features from different sources are considered in the performance inference model. Moreover, in the test scenarios, also other factors must influence the performance significantly. Such information or factors could be, for example, deployment information and hardware data, which would allow us to quantify the dependencies between services deployed on the same physical host and the influence of the hardware usage. Additionally, content information of the requests can be useful to improve performance predictions. For example, if we analyze and predict the parameters of a request, we would be able to model the response time and generated internal calls of an endpoint depending on the input. A concrete use case would be the number of items in a virtual shopping cart on a website. The more items are in the cart the longer the generation of the invoice would take. The architecture of the PPP framework makes the inclusion of such further dependencies easy. Another advantage of our approach is that all of our results are explainable and interpretable. In this scenario, our models match a higher load directly to a higher response time automatically. However, we can see that seasonal patterns and trends of the time series, which are used by the time series forecaster, can also be valuable inputs for the prediction. Hence, in future work, a combination of our approach and time series forecasting could be evaluated. In this way, both time series characteristics and reasoning based on load and other factors can be taken into account for predicting performance degradations.

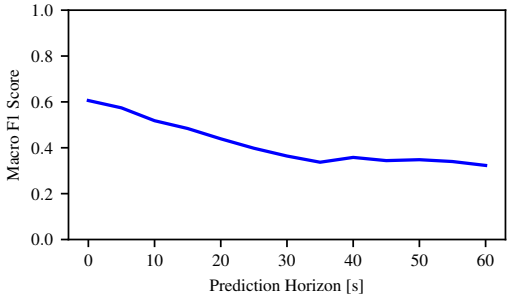
Influence of Different Prediction Horizons. In the last paragraph of this scenario, we want to discuss the prediction quality of our approach with an increasing prediction horizon. This is interesting for real-world use cases, in which performance degradations should be predicted some time in advance to enable proactive acting to mitigate these degradations. The previous investigations were made with the minimal prediction horizon of one measurement interval, which is in our cases five seconds. Figure 7.10a shows the numeric prediction results with different prediction horizons, while Figures 7.10b and 7.10c displays the evolution of the accuracy and macro-averaged F1 score with increasing prediction horizons. Similar to the previous section, the data represent the results of the endpoint `/travel/query` produced by the random forest model. In Figures 7.10b and 7.10c, the metrics of the optimal case, where the measured load is used as an input for the models, is labeled with a prediction horizon of zero. As expected, we see that the prediction quality decreases with an increasing prediction horizon. The numerical results show that the measured peaks appear earlier as predicted by the model. This limits the possibility of mitigating performance degradations in a proactive way. However, when choosing a large prediction horizon useful information can still be retrieved. For example, if we predict a degradation in 60 seconds, but in reality, it happens in 50 seconds, we do



(a) Numeric Prediction Results



(b) Accuracy



(c) Macro-Averaged F1 Score

Figure 7.10.: Influence of the Prediction Horizon on Performance Prediction

not have an exact prediction, but we could still use the 50 remaining seconds for proactive acting.

When analyzing the effect of an increasing prediction horizon in the current version of the framework, we actually analyze the quality of the load forecast with an increasing forecast horizon. The reason for this is that both the request propagation model and the performance inference model do not consider the time horizon. This means that, if we had an optimal load forecast, we would get the same results as shown in Figure 7.9a assuming that the dependency between load and performance does not change significantly over the prediction horizon. Consequently, also the prediction quality over a larger horizon mainly depends on the load forecast quality. As discussed earlier, this dependency could be reduced by including features from other sources. We further discuss the influence of the prediction horizon on the prediction quality in scenario four.

7.2. Scenario 2: Load Peaks on a Frontend Service

Scenario Description. In this scenario, we want to look at the effects of a disproportionately large load on a frontend service and how well we are able to predict them with our models. Therefore, we use the TrainTicket application with the same hardware constraints as in the previous scenario and stress it with eight requests per second constantly. Moreover, we generate a high load of 22 rps every four minutes for 30 seconds at the endpoint `/travel/query`, which is only called by the user. The main difference to the first scenario is that the load on the majority of all other services and endpoints does not increase at these times. In real-world applications, this situation can occur, when customers change

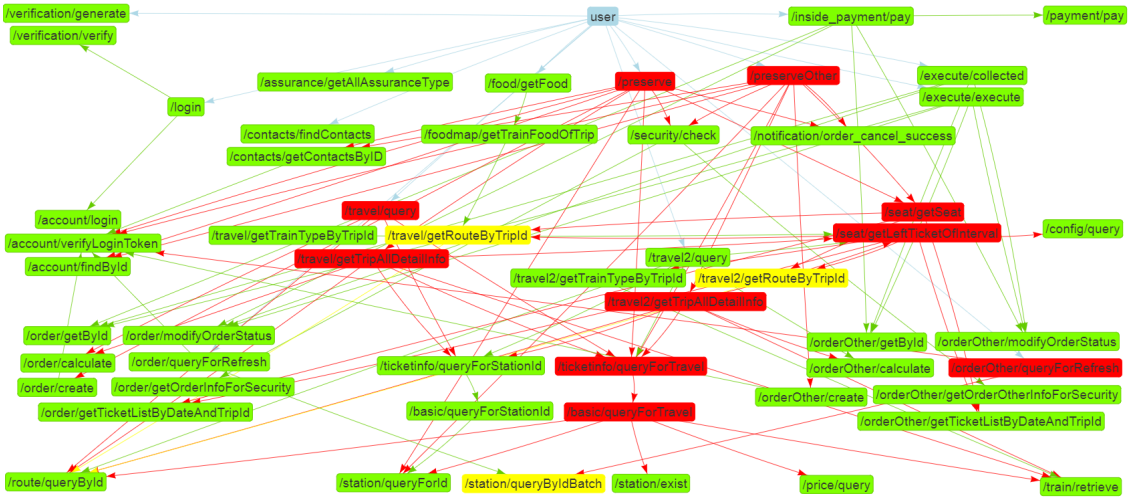


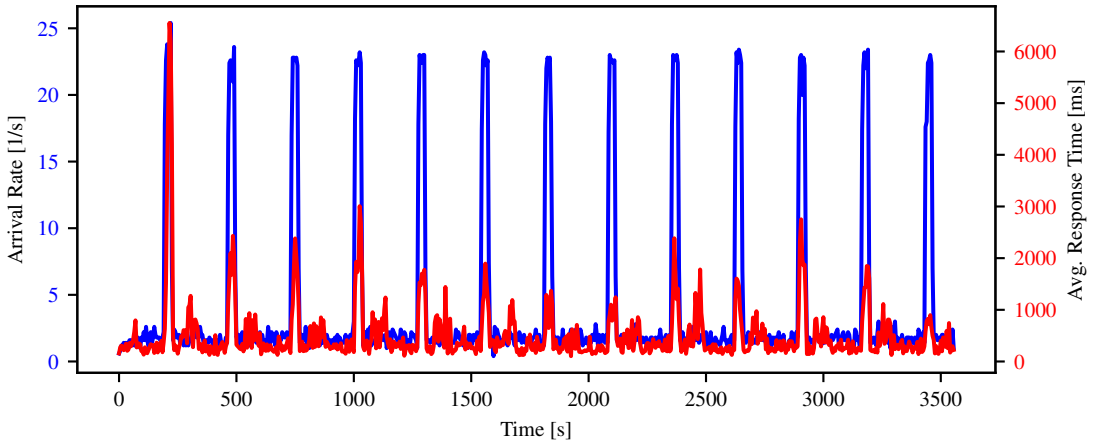
Figure 7.11.: Situation of All Endpoints Stimulated by the Workload

their behavior significantly in a short time, for example, if a limited sale of a popular item starts in a webshop. Another possible reason for a disproportionately large load on one service in real-world applications can be denial-of-service or any other load-driven attacks. Similar to the first scenario, the trace has a total length of about 60 minutes and all measured data are aggregated and queried in intervals of five seconds.

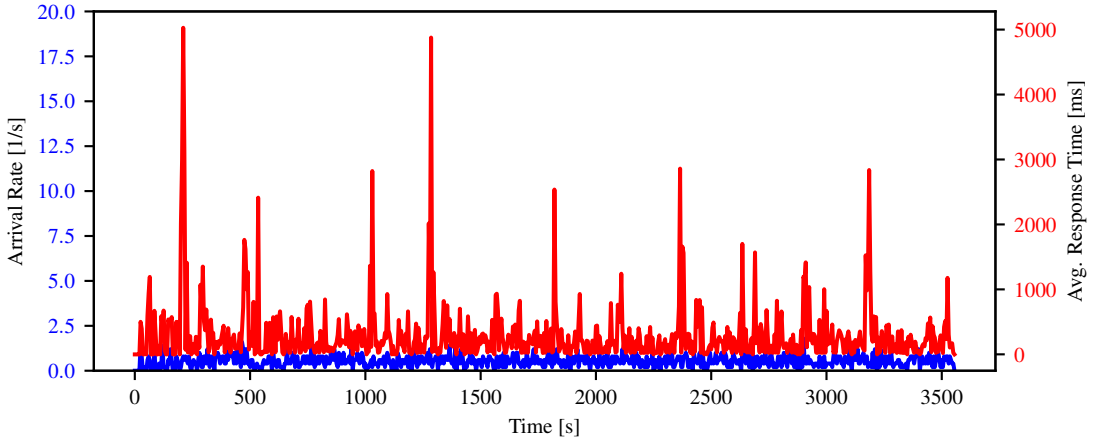
Figure 7.11 visualizes the global situation of the TrainTicket application 2908 seconds after the start of the trace, a time at which the endpoint `/travel/query` experiences a peak load. As mentioned before, this endpoint belongs to the service `ts-travel-service`. We see that multiple other services give performance alerts as well, as a consequence of this large load on the frontend service. As an example, we see that the endpoints `/preserve` and `/preserveOther`, which are called by the user when he wants to book a ticket, have red ratings. Apart from that, other important functions of the application are not affected, for example, the login, visualized in the upper left corner of the figure, and the payment, displayed in the upper right corner.

In the following, we want to investigate the consequences of the large load on the frontend for other selected endpoints. Therefore, Figure 7.12 shows the dependency between load and average response time for three different endpoints. First, Figure 7.12a visualizes the measured data of the endpoint `/travel/query`. We see the expected behavior that whenever the load suddenly increases strongly and average response time ascends as well. Hence, similar to the first scenario, the load on the endpoint itself is the decisive factor of the performance degradation and we observe a direct proportionality between the arrival rate and the response time. The foregoing stands in opposition to the measured values at the endpoint `/preserve`, which are shown in Figure 7.12b. In this case, the load remains at a constant, low level the whole time. However, we observe high peaks of the response time. These variations can obviously not be explained by the load on the endpoint itself. We do not see a direct dependency here. The reasons for the high response times are weak performances of endpoints which are called by `/preserve` in order to process a user request. These weak performances then again are caused directly or indirectly by the high load on `/travel/query`.

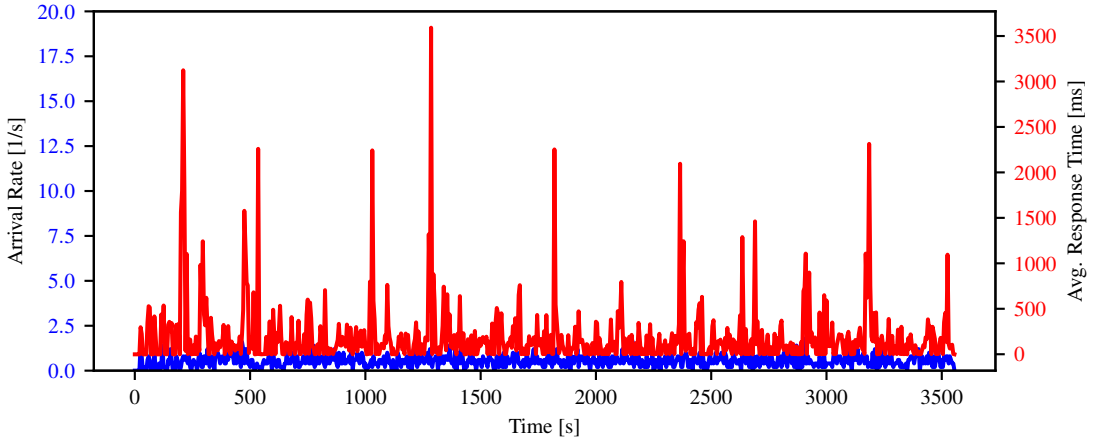
An example of this relationship is given by the endpoint `/travel/getTripAllDetail-Info`, which belongs to the service `ts-travel-service` and is called by the endpoint `/preserve`. The dependency between the arrival rate and the average response time for this endpoint is shown in Figure 7.12c. We see a behavior, which is similar to the one



(a) Endpoint `/travel/query`



(b) Endpoint `/preserve`



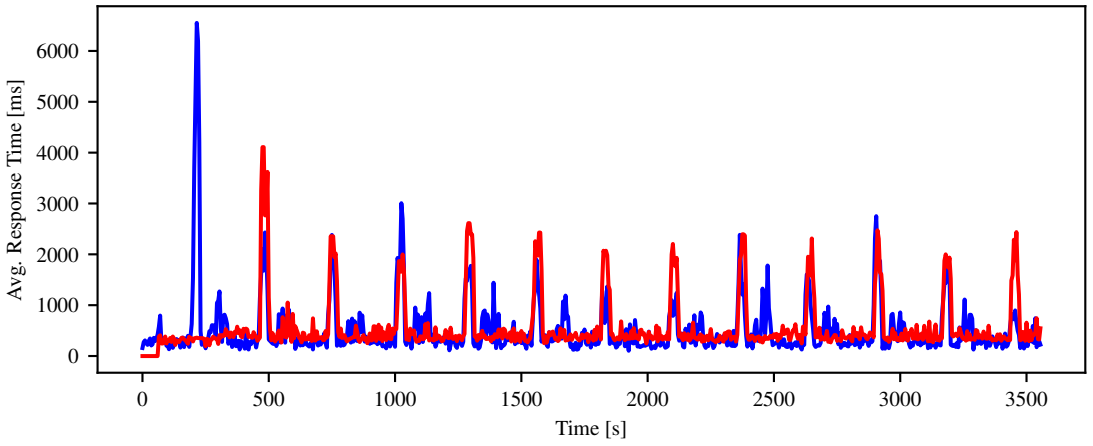
(c) Endpoint `/travel/getTripAllDetailInfo`

Figure 7.12.: Dependency Between Load and Response Time for Different Endpoints

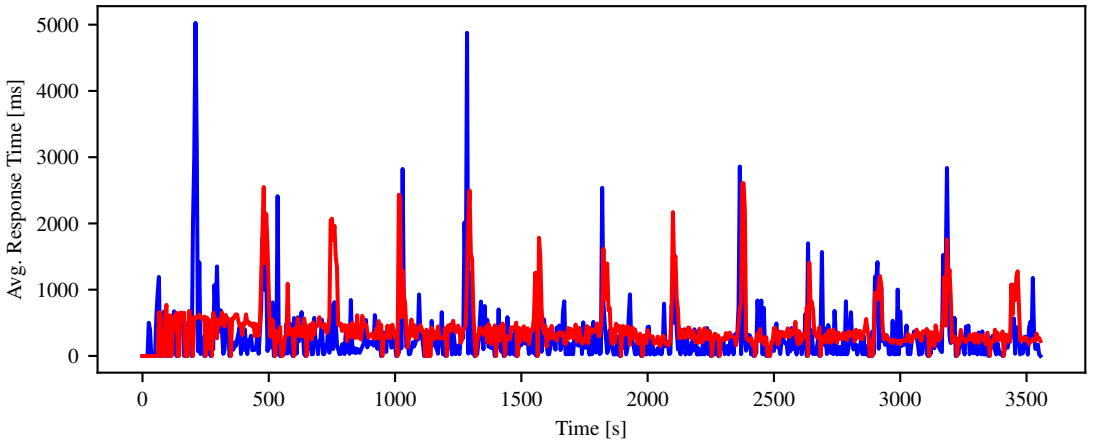
observed at the endpoint `/preserve`. The arrival rate stays at a constant and low level, while the response time has high variations. There are two potential reasons for the performance degradations observed at this endpoint. One possible reason is, similar to the argumentation above, that the endpoint `/travel/getTripAllDetailInfo` depends on services, which have high response times. Hence, requests to these endpoints take much longer than normal and the total response time of `/travel/getTripAllDetailInfo` rises as well. For example, Figure 7.11 shows that `queryForStationId` of the `ts-ticketinfo-service` experiences performance problems and is called by `/travel/getTripAllDetailInfo`. Another possible reason is that the endpoint belongs to the same service as `/travel/query`, which experiences a high load at those times when the response time of `/travel/getTripAllDetailInfo` rises as well. The service `ts-travel-service` has many active threads at these times and, caused by the limited hardware resources, the execution time of its routines increases. Which of the two proposed reasons is the decisive one here is not captured by the data. However, our performance inference model includes both factors in the predictions. More precisely, we both consider the performance of the dependent endpoints and the load of all endpoints of the same service for the performance prediction. This means that both reasons are theoretically covered by our approach. In the following, we want to evaluate whether our framework is able to predict the performance degradations of the three suggested endpoints.

Quality of Performance Prediction. In this section, we want to investigate the quality of the performance prediction for the three endpoints `/travel/query`, `/preserve` and `/travel/getTripAllDetailInfo`. Therefore, we tested and optimized the four model types known from the first scenario. Similar to the previous scenario, the random forest model performed best when considering the prediction quality on all three endpoints and rating classes. The filtered global accuracy of the random forest model in this scenario with a prediction horizon of five seconds amounts 98.9%, while the filtered global F1 score is 0.576. The Tables 7.3, 7.4 and 7.5 show the rating results and metrics on the different endpoints. Compared to the first scenario, we see that the metrics accuracy and macro-averaged F1 score take other values. The accuracy is slightly higher, while the F1 score is lower than in the previous scenario. The main reason for this lies in the nature of the training and evaluation data. Especially the yellow values occur only very seldom in the trace, for `/travel/query` in 82, for `/preserve` in 27 and for `/travel/getTripAllDetailInfo` in 51 out of 712 total intervals. Hence, only a few training data in this rating class are available and a single wrong prediction causes a significant decrease of the F1 score. Meanwhile, the vast majority of data points are in the non-critical green area and many true positive arise in this rating class. Consequently, the overall accuracy increases. Further reasons for wrong predictions are inaccurate load forecasts again. The load forecaster is not able to predict the timing and amplitude in the sudden strong increase of the arrival rate accurately. To suppress this factor, the metrics for the random forest model with optimal inputs are given in the tables.

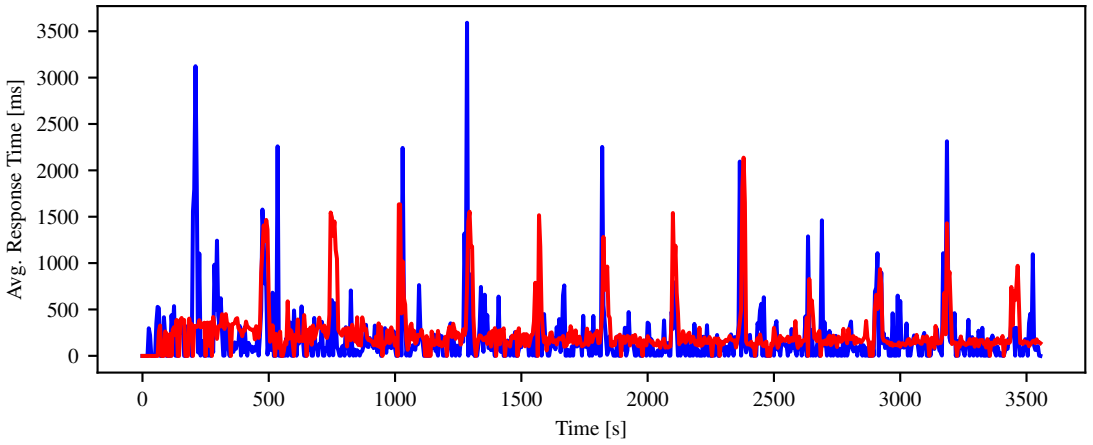
Additionally to this, Figure 7.13 shows the numerical results of the performance prediction of the random forest model with optimal input. We see that the models are able to predict the increase of the response time at all three endpoints whenever the arrival rate at the endpoint `/travel/query` is high. Consequently, we conclude that our approach can model the dependencies between the different endpoints and the effects of the high load on the frontend in this concrete scenario. We further see that the low scores for the predictions are mainly caused by response time peaks which appear between the periodic load peaks. These variations and alerts are not predicted by the model. We assume that these peaks mainly have statistical reasons. The arrival rate at these times is really low, which means that only a few requests come in the system in these intervals. This means that the average



(a) Endpoint `/travel/query`



(b) Endpoint `/preserve`



(c) Endpoint `/travel/getTripAllDetailInfo`

Figure 7.13.: Predicted (red) and Measured Performance Values (blue) for Different Endpoints

Table 7.3.: Rating Results for Endpoint `/travel/query`

Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
Random Forest	507	96	45	**4**	18	**78**	46	41	32	0.782	0.504
Random Forest with optimal input	**525**	**86**	**27**	3	**13**	79	**56**	**29**	**22**	**0.820**	**0.550**

Table 7.4.: Rating Results for Endpoint `/preserve`

Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
Random Forest	600	40	51	**1**	29	**26**	8	34	26	0.855	0.392
Random Forest with optimal input	**611**	**33**	**40**	**1**	**22**	**26**	**13**	**32**	**21**	**0.878**	**0.438**

Table 7.5.: Rating Results for Endpoint `/travel/getTripAllDetailInfo`

Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
Random Forest	**552**	68	**58**	**4**	**32**	**47**	15	41	36	0.802	0.423
Random Forest with optimal input	550	**63**	60	3	41	48	**20**	**35**	**31**	**0.804**	**0.447**

response time is calculated based on the data of only a few requests. Consequently, the outliers have a huge impact on the resulting average response time. When considering only a few requests, statistical and random errors play a more important role compared to intervals with a high arrival rate. In general, the task of predicting the response time for a single or few requests is quite more difficult than predicting the average performance of a large number of requests. Concurrently, the performance prediction of only a few requests is less important in practice. We see that our models underestimate the response time in these intervals. This is because the feature vectors do not differ significantly from the ones with low response times. This means that the reason for these high response times is not captured by our models.

Despite this, this scenario shows that considering and modeling architectural information is important for a good performance prediction. This is the only way to understand the interactions and dependencies between the services and endpoints. Our approach takes these dependencies into account explicitly by including both information of the endpoints of the same service and performance measures of the dependent endpoints. The user of our framework can derive qualitative and quantitative statements on how overloads affect its

application and the customer experience. We further investigate inter-service dependencies and their effects on the performance in the next scenario, where the reason for performance degradations is an overloaded backend service.

7.3. Scenario 3: Load Peaks on a Backend Service

Scenario Description. In this scenario, we continue the investigations regarding the dependencies between different services, their influence on the response time and capabilities of our approach to include this information to predict the performance measure. In contrast to the previous scenario, we generate a disproportionately large load on a backend service, which responds only to calls from other services, does not send requests itself, and is usually not called by a user directly. As an representative example, we selected the endpoint `/train/retrieve` from the service `ts-train-service` for our evaluations. This endpoint receives a train number as an input and returns information about the corresponding train. Under a low load, the average response time of this endpoint is only about two milliseconds in our setup. Hence, to produce a significant performance degradation, we need a high arrival rate. We stress this endpoint every four minutes for 30 seconds with a load of 250 requests per second the first time and with 300 rps all other times. Moreover, we generate constantly 8 rps, which simulate users following our behavior defined in Section 6.2. To investigate the effects of the performance degradation at the backend service on the user experience, we evaluate the prediction quality also on our running example `/travel/query`, which depends directly and indirectly on `/train/retrieve` as shown in Figure 7.4. To avoid the previously described statistical problems with low arrival rates, we further send 8 rps to `/travel/query` directly to generate a bigger pool of requests in every interval. However, the loads on the endpoint `/travel/query` and all other endpoints of the TrainTicket application do not scale with the load of `/train/retrieve` and remain nearly constant at all times. Similar to the previous scenarios, the trace used here as a total length of about 60 minutes, and all measured data are aggregated and queried in intervals of five seconds.

Figure 7.14 shows the ratings 953 seconds after the beginning of the trace of all endpoints of the TrainTicket application stimulated by our workload. At this time, the endpoint `/train/retrieve`, which is shown in the lower right corner of the figure, is stressed with a high load of 300 rps. We see that this high arrival rate leads to a red rating. The average response time rises from 2 ms to about 600 ms. This significant increase has obviously also consequences for services that send requests to `/train/retrieve`. As an example, we see that the endpoint `/basic/queryForTravel` of the service `ts-basic-service` gives a performance alert. This example makes also clear, why it is useful and important to differentiate between the different endpoints of a service. In this case, we see that the endpoint `queryForStationId` of the same service is not affected by the degradation and keeps its green rating. This is because that endpoint does not depend on `/train/retrieve` and can still maintain its normal response time. Besides this, we see that also a customer which calls `/travel/query` would notice the performance degradation as its response time is in a critical area as well.

Figure 7.15 shows the arrival rates and response times of the endpoints `/travel/query` and `/train/retrieve` in more detail. In Figure 7.15a, we see that large arrival rates at the endpoint `/train/retrieve` cause a significant increase of the response times. Moreover, it is shown that the arrival rate varies at the peak times. This is because the load generator struggled to generate a constantly high arrival rate over 30 seconds with our resources. In Figure 7.15b, we see that the endpoint `/travel/query` receives an average of nine requests per second the whole time. The arrival rate for this endpoint has only small variations, which are caused by the dynamic number of users who search for tickets. In contrast to this, the response times show huge spikes. Compared to the previous

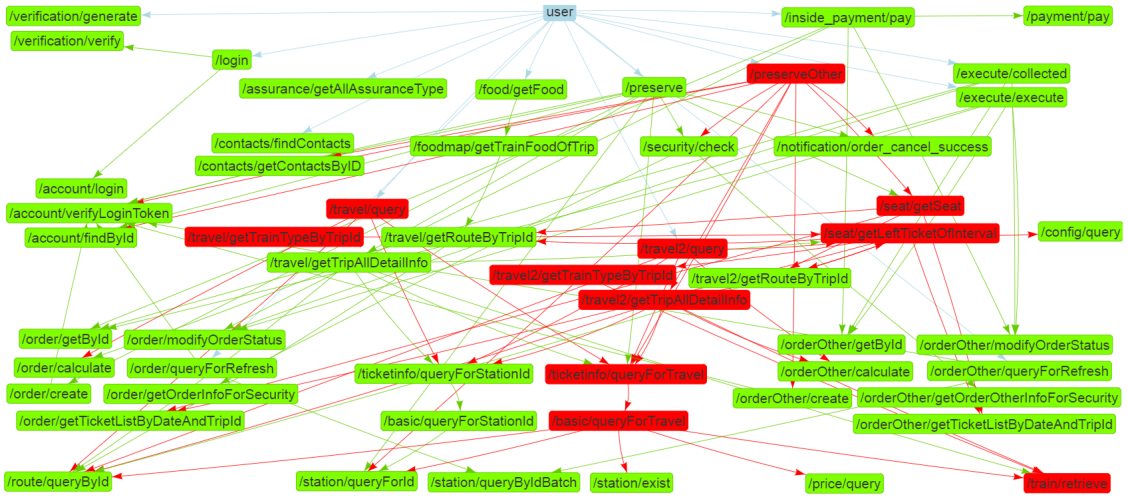
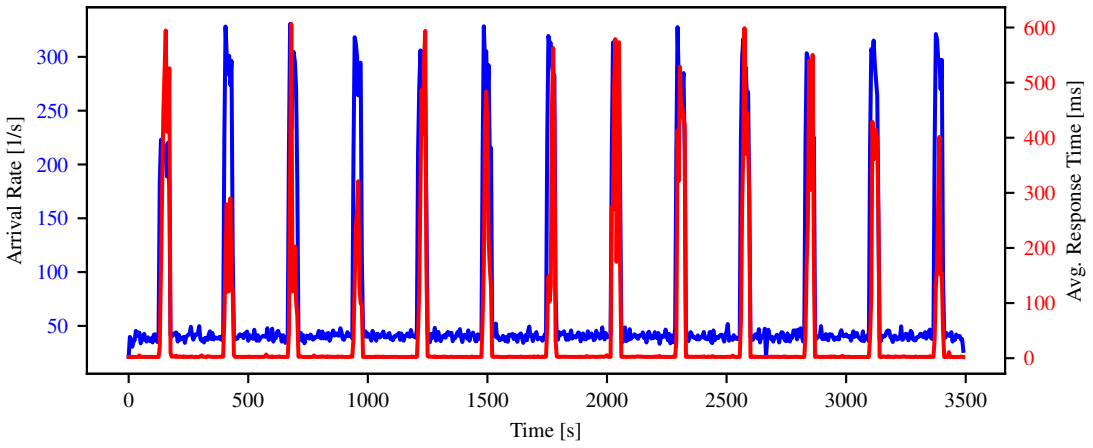


Figure 7.14.: Situation of All Endpoints Stimulated by the Workload

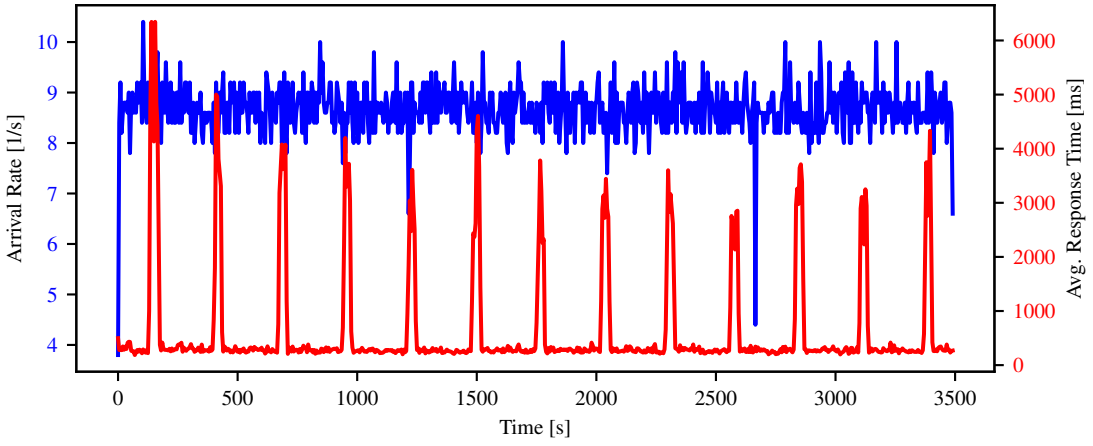
scenarios, the amplitudes of the peaks are even higher. The reason for this is that the endpoint `/train/retrieve` is called multiple times via different paths in order to process a user request to `/travel/query`. This can be seen in Figure 7.4 as well. Consequently, a high response time of `/train/retrieve` causes multiple requests sent by `/travel/query` to have a high response time as well. Hence, the total response time of `/travel/query` increases by a multiple. In the following, we want to evaluate the capabilities of our approach to predict these performance degradations.

Quality of Performance Prediction. In order to assess the prediction quality of our framework in this scenario, we test the four model types known from the previous scenarios as bases for our performance inference models. The hyperparameters of the algorithms are optimized for the endpoint `/travel/query` again and can be found in Appendix A. The Tables 7.6 and 7.7 show the rating results for the endpoints `/train/retrieve` and `/travel/query`. Similar to scenario two, we see high accuracies for all model types. One reason for these high scores is the big imbalance of the datasets. We have more than 600 samples in the green rating class and only about 90 in the red one. Even fewer samples exist in the yellow area, only 13 for `/travel/query` and 9 for `/train/retrieve`. As a consequence of this, we see that our models are not able to make correct predictions in this range. As discussed earlier, this lowers the macro-averaged F1 score significantly. Besides this, we see that our models can predict slightly more than two thirds of all performance degradations measured at these two endpoints. If we received an optimal load forecast, we would predict an even bigger proportion, 86.5% for `/travel/query` and 88.9% for `/train/retrieve`. In general, we see that an optimal load forecast increases the prediction quality significantly. The values in the tables display the performance of the support vector regressor, as this model type performs best among the models when using an optimal user forecast.

Further on, we infer from the tables that all model types perform almost equally well in this scenario. The higher F1 score of the random forest model for the endpoint `/travel/-query` is caused by the correct prediction in the yellow rating class. The reason why all models have almost the same quality measures is situated in the underlying measurement and training data. Figure 7.15 shows that the decisive factor for increasing response times for both endpoints is the high arrival rate at the endpoint `/train/retrieve`. We see that the arrival rate at this endpoint is either really high or low and the increases and decreases happen in a very short time. Similarly, the response times rise and fall sharply.



(a) Endpoint `/train/retrieve`



(b) Endpoint `/travel/query`

Figure 7.15.: Arrival Rate and Average Response Times of Different Endpoints

Table 7.6.: Rating Results for Endpoint `/train/retrieve`

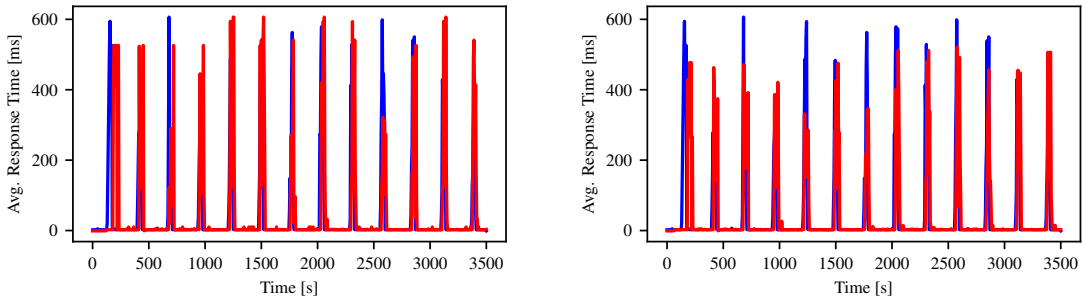
Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
k Neighbors	561	30	40	**0**	**3**	**9**	66	40	24	0.896	0.538
Bayesian	552	28	49	**0**	8	**9**	69	47	21	0.887	0.535
SVR	556	28	45	**0**	14	**9**	63	39	27	0.884	0.532
Random Forest	560	30	41	**0**	**3**	**9**	65	42	25	0.893	0.533
SVR with optimal input	**588**	**16**	**13**	**0**	**3**	**9**	**81**	**12**	**9**	**0.956**	**0.620**

Table 7.7.: Rating Results for Endpoint `/travel/query`

Model Type	GREEN			YELLOW			RED			Overall	Macro F1
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	Score
k Neighbors	555	33	43	0	6	13	62	44	27	0.881	0.524
Bayesian	544	32	54	0	10	13	64	50	25	0.869	0.519
SVR	553	32	45	0	15	13	58	42	31	0.873	0.516
Random Forest	553	34	45	**1**	8	**12**	61	43	28	0.879	0.552
SVR with optimal input	**585**	**20**	**13**	0	**2**	13	**77**	**16**	**12**	**0.946**	**0.606**

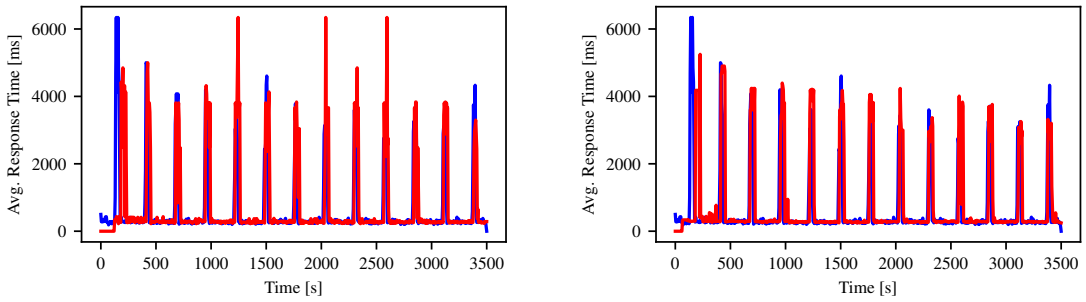
Consequently, there is a firm and unambiguous boundary between the non-critical and critical areas within the training data. All model types are able to capture this clear differentiation. Additionally, we guarantee that the performance degradations are forwarded through the application by using the performance of dependent services as parts of the feature and training vectors.

The recommendation, which model to use in this scenario, depends on the use case and role of the PPP framework. To illustrate this, we want to compare the numeric predictions of the KNN model and the random forest model, which are shown in Figures 7.16 and 7.17. We see that the `KNeighborsRegressor` often predicts higher response times than the `RandomForestRegressor`. When considering the results of the endpoint `/travel/query`, we see that the random forest model provides numerically better results than the KNN model, which often overestimates the response times. Hence, if one focus of the use case is the numeric prediction quality or the prediction of response times, which are noticed by the users, the random forest model would be the better choice here. The main advantages of the `KNeighborsRegressor` are the reduced computational effort and lower training and prediction times. So, when taking the resource usage into account and focusing on the predicted rating only, one might prefer the KNN model. In previous scenarios, we have seen that the KNN model received smaller scores than the random forest model. In this scenario, this model type benefits from the large variations of the arrival rates and response times, as well as from the clear boundary within the training dataset. This supports the base assumption of the KNN model, that similar feature vectors yield similar outputs. In



(a) `KNeighborsRegressor` (b) `RandomForestRegressor`

Figure 7.16.: Numeric Prediction Results for Endpoint `/train/retrieve`



(a) `KNeighborsRegressor` (b) `RandomForestRegressor`

Figure 7.17.: Numeric Prediction Results for Endpoint `/travel/query`

such scenarios, it can be useful to set the hyperparameter k to a really small value. On the one hand, this makes the model sensitive to outliers even if only a few of them are in the training dataset. On the other hand, the numeric quality of the prediction decreases as only a few data points determine the prediction.

All in all, we see that our approach is able to model the effects of a high-loaded backend service on other services. This is also verified by the global quality measures. As an example, the filtered global accuracy for the random forest model amounts 98.8%, while the filtered global F1 score is 0.51. The values for the other model types are in the same range. As discussed in scenario 2, our approach has the advantage that it considers architectural information, which we extract automatically from tracing data, when predicting the performance of a service. This enables the possibility to assess the effects of a bad-performing service or endpoint. The global scores show that we can both determine those endpoints, which are affected by a performance degradation, and those, which are not. This could enable effective resource management in practice. Moreover, by looking into the arrival rates and response times of single services, one can get indications for possible root causes.

7.4. Scenario 4: Study with Realistic Workload

Scenario Description. In the previous sections, we mainly discussed the functionality and capabilities of our approach and the PPP framework. Therefore, we generated appropriate load intensity courses to highlight several effects. In practice, workloads and load intensities are more variable and complex. In this scenario, we want to perform a first test to evaluate whether our approach can bring a clear added value in practice. Therefore,

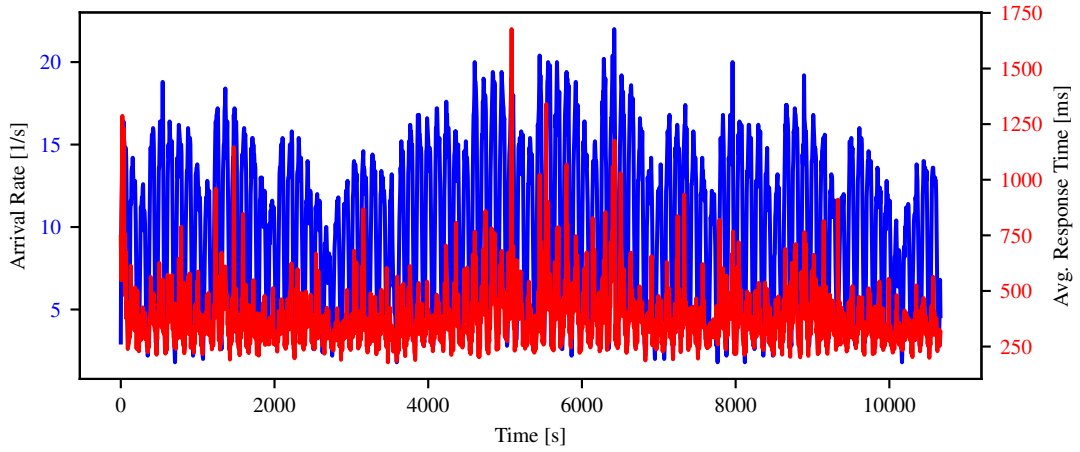
we stress the TrainTicket application with a real-world load intensity course[1]. The data are taken from the Wikipedia page view statistics repository [52]. To use this trace in our setup, we re-scaled the load intensities so that the maximum arrival rate is 22 rps. The measurement data are again queried and aggregated in intervals of five seconds. Hence, we generated a constant arrival rate for a duration of five seconds based on one load intensity value from the re-scaled Wikipedia trace. The load is split on different endpoints of the TrainTicket application corresponding to the user behavior defined in Section 6.2. To avoid the statistical problems discussed in scenario two, we use a separate load generator instance to generate the same load intensity course for the endpoint `/travel/query`, which is used again as a representative example for our evaluation. The total length of the resulting trace is about three hours. The hardware constraints remain unchanged compared to the previous scenarios.

Similar to the previous scenarios, we stick to the data of our running example `/travel/query` for evaluation and comparison purposes. The dependency between load and response times measured at this endpoint is shown in Figure 7.18. Figure 7.18a shows the temporal courses of the two quantities. We see that both measures increase and decrease periodically. The amplitudes of the peaks vary over time. It is shown that between 5000 and 7000 seconds after the beginning of the trace both the arrival rates and response times reach their maximum values. Figure 7.18b highlights the dependency between the two quantities by eliminating the temporal dimension. It is displayed that the response times increase with ascending arrival rates. Moreover, the number of outliers, which have average response times over one second, rises starting from the value of 15 incoming requests per second.

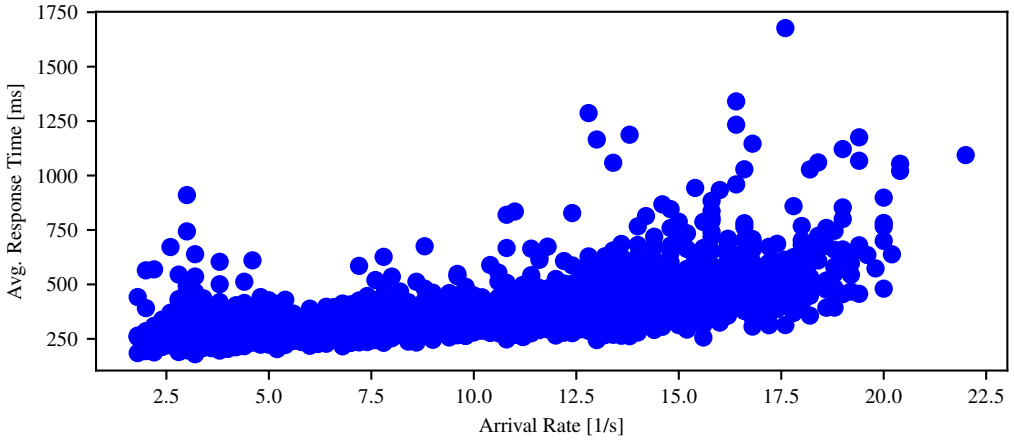
Quality of Performance Prediction. In this scenario, we again use the four model types presented earlier for our evaluations. The hyperparameters have been optimized based on the values of the endpoint `/travel/query`. The search space and resulting parameters can be found in Appendix A. In contrast to the previous scenarios, the models are retrained in intervals of five minutes and consume only data of the last 60 minutes for training. This means that data measured early in the trace are not included after a certain time in the training anymore. We expect that such a sliding window mechanism would be used in practice for various reasons. First, the measurement data can require, depending on the application size and usage, a lot of memory, and also the model training takes longer with a rising number of samples. Especially when memory and computing resources are critical factors, old data have to be excluded from the model training. Second, if old samples are included in the training process, they influence the performance prediction of future values. This can be counterproductive as the application and user behavior might have changed over time. At which time data are considered as outdated depends on the application case. We use the value of 60 minutes in this scenario in order to create a good ratio to the total trace length of three hours for our evaluations.

Table 7.8 shows the rating results for the endpoint `/travel/query` of different model types with a prediction horizon of five seconds. Considering the comparison between the different machine learners, we see qualitatively equal results as in the previous scenarios. The random forest model produces the best results in the competition based on the macro-averaged F1 score. This model type reaches a filtered global accuracy of 99% and a filtered global F1 score of 0.523. We see that the random forest model can predict much more red ratings correctly than the other models. However, the proportion of correctly predicted performance degradations is lower compared to the previous scenarios. The random forest

[1]Download available at: `https://github.com/joakimkistowski/LIMBO/tree/master/DLIM_examples/trace`



(a) Temporal Courses of Arrival Rates and Response Times



(b) Response Time Values Depending on Arrival Rates

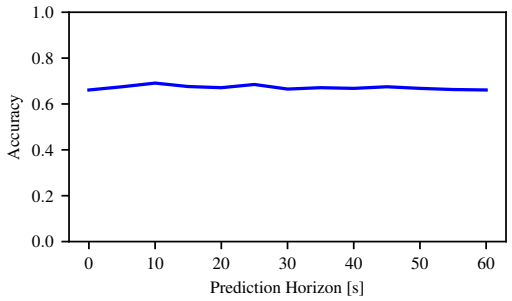
Figure 7.18.: Arrival Rates and Average Response Times of the Endpoint `/travel/query`

Table 7.8.: Rating Results for Endpoint `/travel/query`

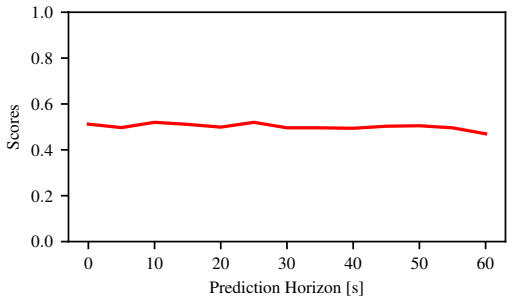
Model Type	GREEN			YELLOW			RED			Overall	Macro
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	F1 Score
k Neighbors	1270	321	215	235	292	283	5	12	127	0.707	0.446
Bayesian	1031	**198**	454	**352**	553	**166**	0	1	132	0.648	0.418
SVR	**1340**	364	**145**	212	**219**	306	0	**0**	132	**0.727**	0.429
Random Forest	1136	242	349	283	405	235	23	46	109	0.675	0.497
Random Forest with opt. input	1082	203	403	294	444	224	**35**	77	**97**	0.661	**0.512**

model predicts 17.4%, with optimal input 26.5%, of all red ratings. The prediction of these performance degradations is even harder in this scenario than in the previous ones. In general, as discussed earlier, we have only a few samples in the red rating class available for training. Here, this problem is aggravated by two factors. First, as only values from the last hour are considered for the model training, the number of available samples in the red rating class is further lowered. Second, freshly gathered measurement data appear with a higher delay in the models, as the retraining interval has been increased from one to five minutes compared to the first three scenarios. Both of these factors impede the prediction of performance degradations.

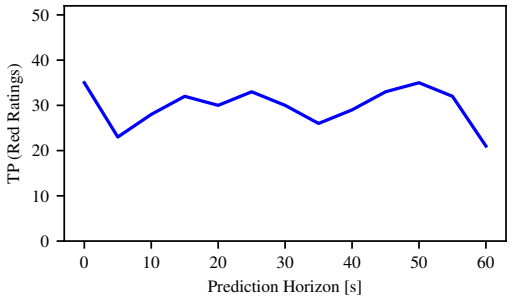
Additionally, we see that the overall accuracy is lower than the values from the previous scenarios. The reason for this is that the load intensity fluctuates more compared to the homogeneous artificial intensities, which have been used in the previous scenarios. These variations lead to varying ratings as well. The state of the endpoint `/travel/query` changes 641 times in the three hours recorded. This means that the endpoint remains only between 15 and 20 seconds on average in one state. Thereby, most state changes happen between the green and yellow ratings. The boundary between the classes is not as clear as in the previous scenarios. This is supported by the small numeric intervals on which the ratings are based. We use the same rating scheme for the endpoint `/travel/query` as in scenario one, which means that an average response time under 400 ms is considered as good, while a response time over 600 ms results in a red rating. We use this labeling to ensure that enough samples exist in all rating classes. However, many samples in this scenario are close to the boundaries of 400 and 600 ms. As a result, also small prediction errors for these samples lead to wrong ratings. In practice, the rating schemes should be defined appropriately for the use case. With a tight rating scheme with small intervals, the rating captures also small increases in the response time. This could be suitable for an application case, where the limit for the red rating is a hard limit and it is better to be warned early. In such cases, preventive actions, like upscaling, might be performed earlier and more often than in other use cases. For other cases, a rating scheme should be chosen where the rating does not fluctuate too often and alerts or warnings are given only if absolutely necessary. Here, any adaptations or other actions could be expensive and one might consider the predictions for multiple prediction horizons before performing an action because a short, temporary performance degradation might be acceptable. The current version of the PPP framework predicts only the performance measures of services and endpoints. As future work, the predictions, also with different prediction horizons, could be used to make advice on whether to perform preventive actions for some services.



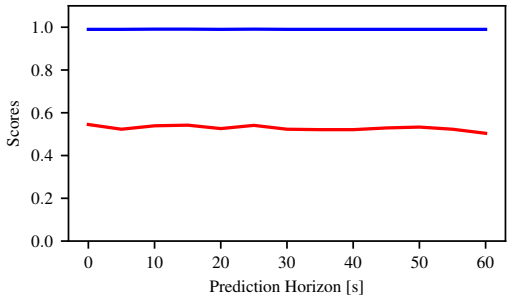
(a) Accuracy on Endpoint `/travel/query`



(b) F1 Score on Endpoint `/travel/query`



(c) Correct Red Ratings on Endpoint `/travel/query`



(d) Filtered Global Accuracy (Blue) and F1 Score (Red)

Figure 7.19.: Prediction Quality Measures Depending on Prediction Horizon

Influence of Different Prediction Horizons. Besides the ability to predict performance degradations, a main requirement for the PPP framework and similar tools is the ability to them sufficient time in advance. Depending on the use case, this interval must be large enough to perform preventive actions or at least to trigger them. We use this trace to make a similar analysis as in scenario one, where we investigate the influence of different prediction horizons on the prediction quality. Figure 7.19 visualizes multiple quality measures of the random forest model and their evolution with ascending prediction horizons. Thereby, the prediction horizon of zero seconds represents the case, when the model receives an optimal load forecast. The Figures 7.19a and 7.19b show that both accuracy and F1 score on the endpoint `/travel/query` stay nearly constant up to a prediction horizon of one minute. This stands in contradiction to Figure 7.10 from scenario one, where a significant decrease of the prediction quality occurs with an ascending prediction horizon. Figure 7.19c shows the number of true positive red ratings for the endpoint `/travel/query`. This metric is important in our use case as it shows how many correct alarms have been given by our models. The accuracy and F1 score do not represent this necessarily. The figure shows that this number sways around the value of 30. Especially remarkable is the fact that the models are able to predict as many red ratings correctly with a horizon of 50s as with an optimal load forecast. By taking the constant accuracy and F1 score into account, we can infer that the prediction quality for the other rating classes does not change significantly as well. Figure 7.19d confirms that also the global prediction quality remains at the same level for different prediction horizons.

In scenario one, we concluded that the quality of the load forecast determines the quality of the performance prediction of our models. This statement holds also in this scenario and we see that the load forecaster maintains its forecast quality also for larger horizons. One reason for this is that the load forecaster is able to use longer time series for its forecasts for the majority of its predictions. After the first hour, the forecaster uses always time

Table 7.9.: Rating Results for Endpoint `/travel/query` (Horizon: 5s)

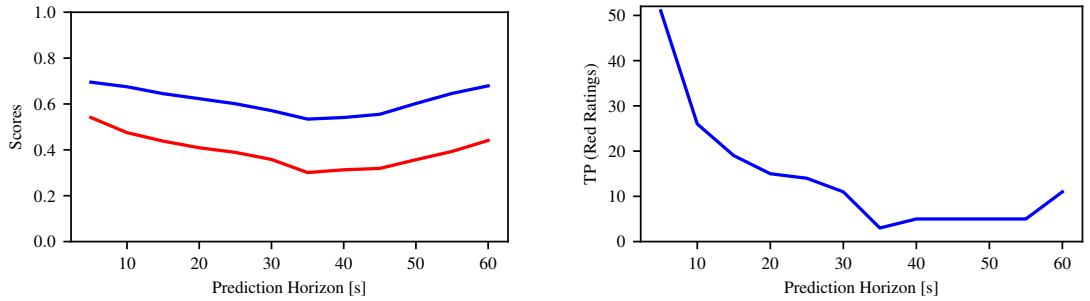
Model Type	GREEN			YELLOW			RED			Overall	Macro
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	F1 Score
Random Forest	1136	**242**	349	**283**	405	**235**	23	**46**	109	0.675	0.497
GluonTS	**1216**	281	**269**	217	**287**	301	**51**	83	**81**	**0.695**	**0.541**

series with a length of 60 minutes for forecasting. In the previous scenarios, the forecaster received many fewer training samples. More training data increase the forecast quality and allows a better analysis of patterns and trends. Moreover, the load intensity courses in this scenario do not contain sharp and abrupt increases or decreases which could lead to large forecast errors.

With these investigations made, the question arises how well our models perform in comparison with the usage of a time series forecaster alone. In scenario one, we saw that the time series forecaster performed slightly better than our models for a horizon of five seconds. In the following, we want to analyze whether this statement holds also in this scenario and how the prediction quality of the time series forecaster evolves compared to our models when increasing the prediction horizon. Similar to scenario one, we use the `SimpleFeedForwardEstimator` from the GluonTS library. In contrast to scenario one, we raise the number of training epochs from 2 to 10 because of the increased number of samples. This increases the training time and iterations for GluonTS which leads to better model parameters and predictions. Table 7.9 shows the rating results for the endpoint `/travel/query` with a horizon of five seconds. We see that the time series forecaster receives better values for both accuracy and F1 score. Hence, we conclude that for this small horizon the time series forecaster performs better than our models and confirm the results from scenario one.

Finally, we evaluate the performance of the time series forecaster with an increasing forecast horizon and compare it to the performance of our approach in this scenario. Therefore, Figure 7.20 visualizes different prediction quality measures of the `SimpleFeedForwardEstimator` for the endpoint `/travel/query`. Figure 7.20a shows the accuracy and F1 score for different forecast horizons. In contrast to our models, the prediction quality decreases significantly with an ascending horizon. Both measures descend up to a horizon of 35 seconds, which equals seven measurement intervals. Beyond this point, the values increase again. But, this rise is only based on more correct predictions in the green and yellow rating classes. This is shown in Figure 7.20b, which displays the correctly predicted red ratings. In this figure, we see that the value decreases from 51 with a horizon of five seconds to single-digit values for horizons greater than 30 seconds. Moreover, the time series forecaster issues less correct red ratings than the random forest model for all horizons bigger than five seconds. Table 7.10 shows the rating results for the endpoint `/travel/query` for the random forest model and the `SimpleFeedForwardEstimator`. It confirms that the predictions of the time series forecaster have much lower quality scores and true positive values for all three rating classes compared to the random forest model. Furthermore, it is shown that the random forest model receives slightly better scores, while the time series forecaster performs much worse, compared to the results with a horizon of five seconds shown in Table 7.9.

All this shows that our models should be preferred for larger prediction horizons. Similar to the time series forecaster, our models include past values of the performance into the prediction by consuming them at training time. In contrast to the time series forecaster,



(a) Accuracy (Blue) and F1 Score (Red)

(b) Correct Red Ratings

Figure 7.20.: Prediction Quality Measures for Time Series Forecaster

Table 7.10.: Rating Results for Endpoint `/travel/query` (Horizon: 50s)

Model Type	GREEN			YELLOW			RED			Overall	Macro
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	F1 Score
Random Forest	**1152**	**271**	**333**	**239**	371	**279**	**35**	67	**97**	**0.668**	**0.505**
GluonTS	1142	450	343	139	**356**	379	5	**43**	127	0.602	0.357

they do not take the temporal dimension into account. Instead of this, they consider one of the main reasons for performance degradations and variations into account: the arrival rate. The models assume that a similar arrival rate, together with the performance of the dependent services of an endpoint, leads to similar response times. The better this statement holds also quantitatively in practice, the better our approach works. In this case, the models are able to infer a range from their training data, where the next values of the performance metrics will be most likely. If more training data are available, this range gets smaller and also the numeric prediction quality increases. Nevertheless, as discussed earlier, the prediction quality depends on the quality of the load forecast. In direct comparison to the time series forecaster, our approach performs better for larger horizons because the load intensity at this time, which is taken into account by our models, is more important for the performance prediction than the temporal course of the performance measures, which is the base of the forecast from a time series forecaster. For small prediction horizons, the time series forecaster can infer accurate values for the performance measures by taking the historical values and their evolution into account, in particular when no abrupt change in the load intensity or performance measures take place. In practice, the probability that the response time changes strongly is much higher when considering long time intervals than short ones. To validate these results also in other use cases and settings, further measurements and investigations will be made in future work.

7.5. Scenario 5: Study with Second Application

Scenario Description. In the previous sections, we claim that one of the advantages of our approach is that it is not application-specific. In this scenario, we want to perform an initial test on a second microservice application in order to evaluate the performance of our approach and compare the results with the previous scenarios. Therefore, we use the Teastore application presented in Section 3.5.2 and stress it with a periodic load, analogous to scenario one. The maximum load intensity is 120 requests per second, while the minimum intensity is 19 rps. We hereby simulate users, who browse for different items

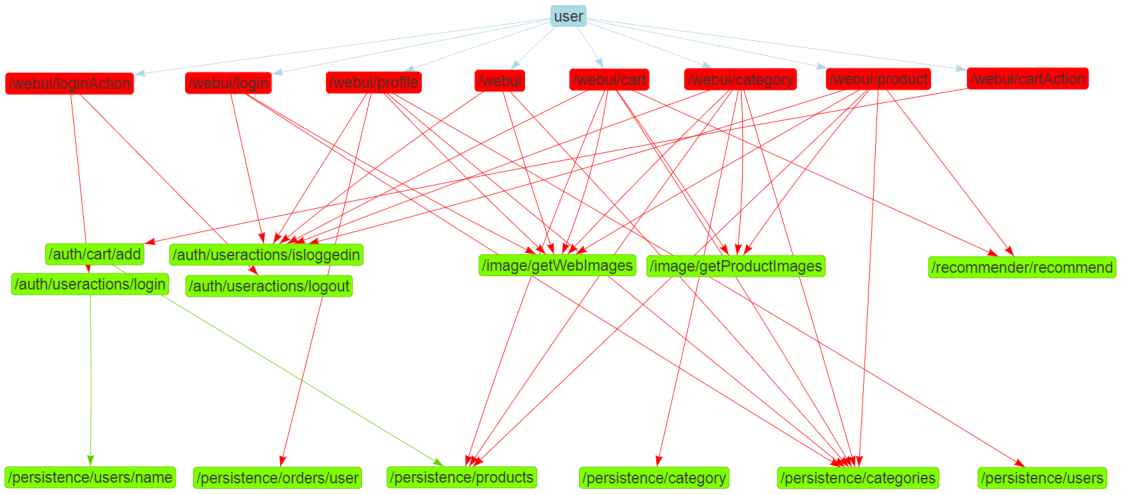


Figure 7.21.: Situation of All Endpoints Stimulated by the Workload

in the webshop, as described in Section 6.2. The recorded trace has a total length of 5000 seconds. The peak load occurs ten times during this period. Similar to the previous scenarios, all measured data are aggregated and queried in intervals of five seconds.

Figure 7.21 shows the rating of all endpoints stimulated by our workload 1253 seconds after the beginning of the trace. At this time, the Teastore is stressed with the maximum arrival rate of 120 rps. We see that far fewer endpoints exist within the application compared to TrainTicket. Moreover, a clear structure is visible. All user requests are sent to the `webui` service, which then sends requests to different backend services. As described in Section 6.1, we assign one CPU core and 4GB RAM to every service. We see that the `webui` service is the bottleneck under these resource constraints when the application is stressed with a large number of requests. Meanwhile, all other services do not experience performance degradations. In contrast to the TrainTicket application, all endpoints of the Teastore application have average response times under low load smaller or equal to 40 milliseconds. As a consequence, we use a fixed rating scheme for all endpoints. An average response time under 80ms is considered as good and yields a green rating, while a response time above 130ms results in a red rating. For the sake of improved clarity, we shortened the name of the endpoints in the following paragraphs and figures. An overview of all endpoints with their full names and response times is given in Appendix C.

As a representative example within the Teastore application, we select the endpoint `/webui` for evaluation purposes. This endpoint is called whenever a user accesses the homepage of the Teastore and consequently, its performance measures could be relevant also in practice. Moreover, the endpoint is sensitive to load variations, as Figure 7.22 shows. Hence, it is well-suited for our evaluation. The figure shows that starting from an arrival rate of 25 rps, the average response time increases strongly and reaches values up to 1200 milliseconds. However, the main reason for the performance degradation is here the total arrival rate to the `webui` service, which is at this time about 100 rps. As mentioned earlier, the `webui` service turns out as the bottleneck under our hardware constraints as all user requests are processed there. Hence, all endpoints of this service experience performance problems when stressed with a high load. In the following, we want to evaluate the prediction quality of the PPP framework in this scenario and compare the results with the previous test cases. In the introduction, we also ask the question, which changes are needed to apply the framework to another application. It turned out that both during the instrumentation with Pinpoint and the use of the PPP framework no major changes had to be performed. Only the patterns, which separate the endpoint name from its parameters, were adjusted.

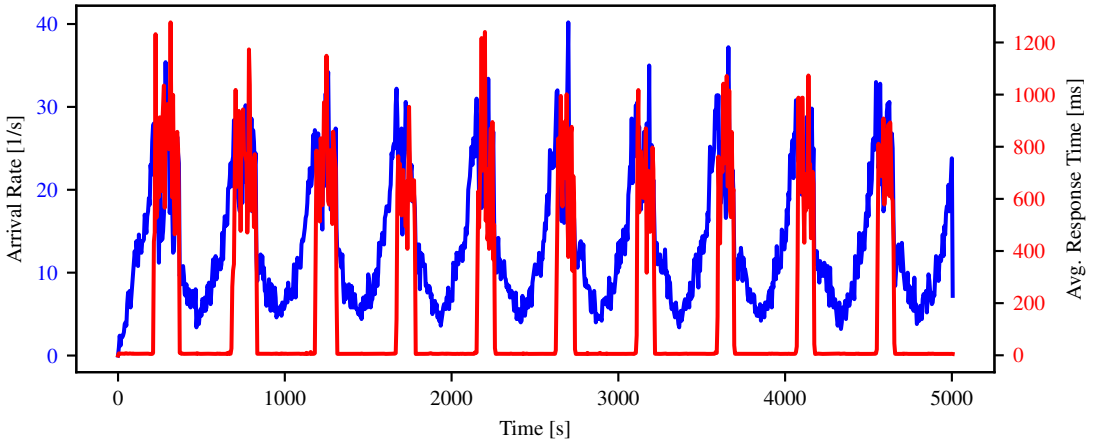


Figure 7.22.: Load and Average Response Time of the Endpoint `/webui` over Time

Table 7.11.: Rating Results for Endpoint `/webui`

Model Type	GREEN			YELLOW			RED			Overall	Macro
	TP	FP	FN	TP	FP	FN	TP	FP	FN	Accuracy	F1 Score
k Neighbors	**726**	91	**42**	**0**	3	**0**	143	**39**	91	0.867	0.535
Bayesian	478	**8**	290	**0**	79	**0**	**226**	211	**8**	0.703	0.479
SVR	495	13	273	**0**	62	**0**	221	211	13	0.715	0.480
Random Forest	699	70	69	**0**	29	**0**	145	59	89	0.842	0.524
KNN with opt. input	725	45	43	**0**	**0**	**0**	189	43	45	**0.912**	**0.918**

As discussed in Chapter 5, these patterns are only prototypical solutions and might be replaced in future versions of the PPP framework.

Quality of Performance Prediction. Similar to the first four scenarios, we want to evaluate the performance of the four well-known model types. The hyperparameters are evaluated based on the data of the endpoint `/webui`, the search spaces remain unchanged compared to the other test cases. The resulting parameters can be found in Appendix A. Table 7.11 shows the rating results for the endpoint `/webui` with a prediction horizon of five seconds. As shown in Figure 7.22, the response time of this endpoint is either really low and close to its minimum or takes high values, which are bigger than ten times the response time under low load. Consequently, only green and red ratings occur in the trace. However, yellow ratings can still be predicted by the models, as we use regression techniques and assign the ratings based on the predicted numeric values. A prediction of a yellow rating is strongly penalized by the macro-averaged F1 score, as the F1 score for the yellow rating class gets zero (see Equations 6.3 and 6.4). We have seen a similar behavior and jumpy response times in scenario three as well. Hence, the values of the macro-averaged F1 scores are in the same ranges. The results of the KNN model with optimal input show that the F1 score rises significantly if no predictions in the yellow rating class are made.

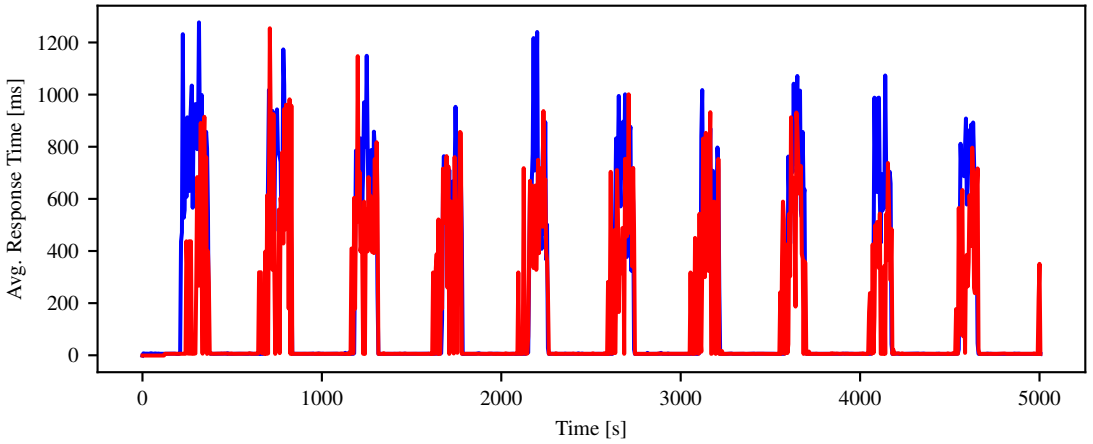


Figure 7.23.: Measured (Blue) and Predicted (Red) Response Times of the Endpoint `/we-bui`

Among the four evaluated model types, the KNN performs best just before the second-placed random forest model. The global filtered F1 score of the KNN model is 0.535 and the global accuracy is 0.947. Similar to previous results, the Bayesian regressor and the support vector regressor overestimate the response time in many cases, which leads to more false positives in the red and yellow rating classes and fewer true positives in the green rating class. In general, we can see that the models achieve high accuracies in this scenario. The reason for this is that almost no samples exist near the rating thresholds and the dataset can be separated well. As discussed in previous chapters, the KNN regressor can handle such datasets well with small computational effort. This statement is confirmed by the results in this scenario. Similar to the previous test cases, we select a small value of two for the hyperparameter k, which leads to the fact that the predictions are strongly geared to the training data. This is supported by the numeric prediction results, which are shown in Figure 7.23. We see that the first peak is not predicted correctly, considering both timing and amplitude. The response time at the second peak is overestimated at the beginning because the training samples from the first peak suggest a higher peak response time. Afterward, the predicted response times under high load stabilize around the mean peak response times.

All in all, we conclude that our results from the TrainTicket application align with the data in this scenario. The Teastore application with our resource constraints has a different behavior as the TrainTicket application under periodic load and consequently, the results from scenario one and scenario five differ significantly. In fact, we see a combination of the results from different scenarios here. While the response times of the TrainTicket application rise continuously with an increasing load intensity (see Figure 7.2), the measurements in this test case are characterized by abruptly increasing response times, so that the results can be compared with those from scenario three most likely. In scenario three, we saw jumpy response times as a consequence of the degradation of a backend service. In contrast to this, the services `auth`, `image`, `recommender` and `persistence` do not experience performance problems in this scenario. This stands also in opposition to scenario two, where high loads and performance degradations at a frontend service led to problems also for other services. Our framework has no difficulties to recognize and model this varied application behavior. The rating results and metrics are in the same range as in previous scenarios and the explainability of the results is preserved. However, the question of whether and how the application and topology size influences the prediction quality can be not answered finally. Therefore, more measurements on different applica-

tions and environments have to be performed. Nevertheless, we have shown that the PPP framework can be applied to different applications easily without requiring major changes.

7.6. Summary and Discussion

In the previous sections, we evaluated the capabilities, pros, and cons of our approach in five scenarios and two test applications. In the following, we summarize the main findings of these investigations. Especially in the first three scenarios, we see that performance degradations in microservice applications can evolve for various reasons. On the one hand, an overload can be the decisive factor for the performance degradation of a microservice. On the other hand, a high response time of a frontend service can be caused by performance degradations in the backend. Both of these reasons are important in practice and modeled within our approach. Hence, we conclude that the features, which we use for the performance prediction, are meaningful and necessary. In particular, we observe that it is indeed important to take architectural information for performance predictions into account. In scenario two and three, we see that our framework is able to detect and model different degradation sources within the application topology with short training times. We further conclude that the models are able to learn dependencies fast and on the fly. The PPP framework requires no prior data and has no information about the application and its architecture at the start-up. In all scenarios, we see a fast convergence of the prediction results and valid predictions even with few training samples.

In the following, we analyze the components of our approach in more detail. The first important part of our prediction process is the request propagation model. In this evaluation, we test an instance of the abstract model presented in Chapter 4, which uses linear propagation functions to model the dependencies between different microservices. We observe that this linearity is a good approximation in our evaluation. The relative prediction errors of the arrival rates increase by only three percent as compared to the user request forecasts from GluonTS. One advantage of the linear propagation model is the low training time and effort. Additionally, this model can be used in other contexts as well besides request propagation. In scenario one, we show that business information can be extracted from the structure of the graph and the propagation coefficients c. In addition to this, the model might be used for software design and testing purposes, e.g., for debugging. Within our approach, the request propagation model contributes a large part for the explainability of our results as it displays qualitative and partially quantitative dependencies between the endpoints and services. The question, how the linear model performs with real user behavior, remains open. We expect a temporarily more volatile behavior, which might lead to larger prediction errors. However, we assume that over a long period the averaging mechanism of the request propagation model should approximate the real values well.

The second important part of our prediction process is the performance inference model. In this evaluation, we test four different machine learning models for this purpose. Considering the results from all five test cases, the random forest model performs best in this competition. It offers fast converging and accurate results. The response times are neither permanently over- nor underestimated. The k nearest neighbors model performs well, whenever the evaluation data are very similar to the training data and the dataset is easily separable in green and red areas. One advantage of this model type is the low training time and effort and the fact that it produces the most explainable results within the competition. The Bayesian regression model overestimates the response times in many scenarios, which leads to many false alarms. In contrast to this, the support vector regressor underestimates the response times in many scenarios and consequently predicts the fewest degradations correctly. In practice, the selection of the model type depends on the requirements, role, and usage context of our framework. Moreover, the available resources and the numeric prediction quality might play an important role.

When evaluating the prediction quality of the different model types, one has to be aware of how to interpret the metrics and results. Some values might appear as relatively small or high at first glance, but there are some effects that influence the calculation and can not be neglected here. First, we use regression models to predict continuous performance metrics and then assign labels based on fixed rating schemes. This leads, in particular in scenarios one and four, where many samples exist near the rating limits, to the fact that small numeric deviations can cause a false rating and therefore a false prediction. Moreover, all test cases are characterized by strongly imbalanced datasets, the total number of yellow and red rating samples are small. The proportion of green ratings, considering the ratings from all endpoints of the application, is always well above 90 percent. Another factor, which lowers especially the F1 score, is our threefold rating system. If we switch to a binary classification, where the green and yellow ratings are summarized in one class, the values would increase significantly. For example, in scenario four, the F1 score of the random forest model would increase from 0.497 to 0.606 and the accuracy would rise from 67.5% to 92.7% without altering the results or models. In general, the metrics represent always the results based on one specific rating scheme, when altering this scheme the metrics will change. Moreover, the numeric results show that the models approximate the courses of the performance metrics well and false red or yellow ratings are either just before and just after a measured one.

In the last part of this section, we summarize the strengths and weaknesses of our approach and the PPP framework. One benefit of our approach is that it does need prior knowledge. The PPP framework extracts information about the application and its topology automatically and brings them in an easily understandable form. The different evaluation modes of the PPP framework allow online or offline usage. Moreover, its components can be exchanged or extended easily. In scenario five, we show that our approach is also not application-specific. In scenario four, we see that also for larger prediction horizons the prediction quality is maintained. The main weakness of the current version of the approach is the strong dependency on the load forecast. In fact, all features are directly or indirectly derived from the load forecast. Hence, errors from the user request forecast propagate through the whole prediction process. We conclude that information and features from multiple other sources should be included in the prediction process. These could be, for example, deployment information or further statistics of the requests, e.g., parameters. Such additional sources and features can be included easily in the PPP framework. More strengths and weaknesses can be pointed out when evaluating the approach in practice under realistic conditions. For a final assessment, longer traces from real-world traffic are needed. The question, how bigger aggregation and measurement intervals influence the prediction quality, remains open. Nevertheless, this evaluation provides a good basis for further investigations.

8. Conclusion and Outlook

Performance prediction for microservice applications is a non-trivial task, as all components have their own characteristics, dependencies, and deployment contexts. In this work, we introduced a new approach for the prediction of performance degradation of microservice applications. The approach takes the load intensity and the load distribution across the application topology as the most important causes of performance degradations into account. We foresee future states of microservices using machine-learning-based regression models, which predict performance metrics based on architectural information and load forecasts. We examined the theoretical aspects and presented a reference architecture for the realization of the approach. With the Propagation Performance Prediction (PPP) framework, we designed a concrete implementation, which is characterized by high extensibility and flexibility. In our evaluation, we used realistic workloads and two state-of-the-art microservice applications. We designed and performed measurements in five test scenarios, which simulate different application states and degradation courses. All in all, we conclude that our approach is suitable for the performance prediction of different microservice applications. The results show that the models are able to learn the performance behavior and architectural dependencies of the applications quickly and without prior knowledge. The predictions are characterized by high accuracies of more than 95% when taking the whole application into account. Depending on the test scenario, up to 72% of the measured performance degradations are predicted correctly. Moreover, the models are able to keep their prediction quality nearly constant even with higher prediction horizons.

Taking all results into account, we conclude that our approach is versatilely applicable. In particular, it is configurable, extensible, and application-agnostic. This is especially important for real-world use cases. For example, our framework can give an overview of the application state and possible performance issues for a service or cloud provider, which manages several microservice or containerized applications. By doing this, actions, which prevent performance degradations, can be recommended or triggered. Another benefit of our approach is the explainability of the results. Every step in the prediction process is comprehensible and the output is easily interpretable. In addition to this, we have shown that also simple machine learning approaches, like the k nearest neighbors algorithm, can be used for performance prediction. This is especially interesting in use cases, where only limited resources are available. A further advantage of our approach is that it does not require prior knowledge and can work based on realtime or recorded data. This opens the possibility to use it online or offline.

The main weakness of the current version is the strong dependence on the load forecast. Whenever the load forecast is not good, the performance inference and request propagation model can not generate a good prediction. Another limitation of the current approach is that only performance degradations, that are caused directly or indirectly by an overloaded service, can be predicted. In practice, other factors might cause high response times of microservices and other degradations, for example, varying network conditions. Both of these limitations can be overcome by including more performance-relevant features from different sources. By doing this, the full potential of our approach can be realized. Such additional features could be, for example, deployment information and hardware measures or more detailed information about the requests, e.g., their parameters. The PPP framework and its modular architecture offer possibilities to integrate such additional features easily.

In our evaluation, we illustrated that time series forecasters can generate predictions of equal quality compared with our models at least for small forecast horizons. Considering this, we conclude that the temporal course of the performance metrics can be another good feature that could be included in the prediction process. This would increase the numeric prediction quality in particular. A further possible enhancement would be the use of performance metric ranges instead of single values. Thereby, we would receive a load forecast consisting of a minimum and maximum arrival rate and would generate, for example, a minimum and maximum response time. By doing this, we can include a measure of prediction uncertainty and confidence. Furthermore, several additional extension points of our approach have been described in this work. For example, our approach could be extended by a root cause localization algorithm, which names possible failure sources. This enables also the possibility to make explicit recommendations on how to prevent future degradations. Moreover, also self-improvement mechanisms might be used to vary the prediction models depending on their prediction quality. From a technical point of view, more interfaces for other monitoring tools and load forecasters can be developed. By doing this, more data formats can be supported, and also a detailed evaluation in a real-world environment is simplified. In terms of such an evaluation, further investigations, for example on the influences of monitoring intervals, request sampling processes, and real-world user behaviors on the prediction quality might be performed.

Nevertheless, we reached our pre-defined goals in this thesis. We presented a new abstract model for the prediction of performance degradations in microservice applications and performed a comprehensive evaluation. Therefore, we designed and developed an extensible framework, that can and will be used in future works.

List of Figures

List of Tables

Acronyms

API Application Programming Interface

CPU Central Processing Unit

CSV Comma-Seperated Values

FN False Negative

FP False Positive

HTTP Hypertext Transfer Protocol

JAR Java Archive

JSON JavaScript Object Notation

JVM Java Virtual Machine

KNN K Nearest Neighbors

ML Machine Learning

RAM Random Access Memory

REST Representational State Transfer

RPS Requests per Second

SLO Service-Level Objectives

SVM Support Vector Machines

SVR Support Vector Regression

TN True Negative

TP True Positive

UI User Interface

Bibliography

[1] J. Lewis, "Micro services - java, the unix way." 33rd Degree Conference 2012 Krakow, Poland, 2012.

[2] J. Lewis and M. Fowler, "Microservices." Online Article, March 2014.

[3] A. Sriraman and T. F. Wenisch, "my suite: A benchmark suite for microservices," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 1–12, Sep. 2018.

[4] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[5] E. Frank, L. Trigg, G. Holmes, and I. H. Witten, "Technical note: Naive bayes for regression," *Machine Learning*, vol. 41, pp. 5–25, Oct 2000.

[6] I. Rish, "An empirical study of the naive bayes classifier," tech. rep., 2001.

[7] G. H. John and P. Langley, "Estimating continuous distributions in bayesian classifiers," in *Proceedings of the Eleventh Conference on Uncertainty in Artificial Intelligence*, UAI'95, (San Francisco, CA, USA), pp. 338–345, Morgan Kaufmann Publishers Inc., 1995.

[8] J. D. M. Rennie, L. Shih, J. Teevan, and D. R. Karger, "Tackling the poor assumptions of naive bayes text classifiers," in *Proceedings of the Twentieth International Conference on International Conference on Machine Learning*, ICML'03, pp. 616–623, AAAI Press, 2003.

[9] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, pp. 5–32, Oct. 2001.

[10] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, Sep 1995.

[11] H. Drucker, C. J. C. Burges, L. Kaufman, A. Smola, and V. Vapnik, "Support vector regression machines," in *Proceedings of the 9th International Conference on Neural Information Processing Systems*, NIPS'96, (Cambridge, MA, USA), pp. 155–161, MIT Press, 1996.

[12] J. Vert, K. Tsuda, and B. Schölkopf, *A Primer on Kernel Methods*, pp. 35–70. Cambridge, MA, USA: MIT Press, 2004.

[13] "Pinpoint - leading open-source apm." `https://naver.github.io/pinpoint/`. Accessed: 2019-11-26.

[14] "Http load generator." `https://github.com/joakimkistowski/HTTP-Load-Generator`. Accessed: 2020-05-17.

[15] A. Alexandrov, K. Benidis, M. Bohlke-Schneider, V. Flunkert, J. Gasthaus, T. Januschowski, D. C. Maddix, S. Rangapuram, D. Salinas, J. Schulz, L. Stella, A. C. Türkmen, and Y. Wang, "Gluonts: Probabilistic time series models in python," 2019.

[16] I. Ilic, B. Gorgulu, and M. Cevik, "Augmented out-of-sample comparison method for time series forecasting techniques," in *Advances in Artificial Intelligence* (C. Goutte and X. Zhu, eds.), (Cham), pp. 302–308, Springer International Publishing, 2020.

[17] D. Salinas, H. Shen, and V. Perrone, "A copula approach for hyperparameter transfer learning," 2019.

[18] S. Becker, L. Grunske, R. Mirandola, and S. Overhage, "Performance prediction of component-based systems," in *Architecting Systems with Trustworthy Components* (R. H. Reussner, J. A. Stafford, and C. A. Szyperski, eds.), (Berlin, Heidelberg), pp. 169–192, Springer Berlin Heidelberg, 2006.

[19] A. Matsunaga and J. A. B. Fortes, "On the use of machine learning to predict the time and resources consumed by applications," in *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, pp. 495–504, May 2010.

[20] A. Andrzejak and L. Silva, "Using machine learning for non-intrusive modeling and prediction of software aging," in *NOMS 2008 - 2008 IEEE Network Operations and Management Symposium*, pp. 25–32, April 2008.

[21] F. Hassan, S. Farhan, M. A. Fahiem, and H. Tauseef, "A review on machine learning techniques for software defect prediction," *Technical Journal*, vol. 23, no. 02, pp. 63–71, 2018.

[22] R. Bianchini, M. Fontoura, E. Cortez, A. Bonde, A. Muzio, A.-M. Constantin, T. Moscibroda, G. Magalhaes, G. Bablani, and M. Russinovich, "Toward ml-centric cloud platforms," *Commun. ACM*, vol. 63, p. 50–59, Jan. 2020.

[23] J. Grohmann, N. Herbst, A. Chalbani, Y. Arian, N. Peretz, and S. Kounev, "A Taxonomy of Techniques for SLO Failure Prediction in Software Systems," *Computers*, vol. 9, no. 1, 2020.

[24] F. Salfner, M. Lenk, and M. Malek, "A survey of online failure prediction methods," *ACM Comput. Surv.*, vol. 42, pp. 10:1–10:42, Mar. 2010.

[25] T. Pitakrat, D. Okanovic, A. V. Hoorn, and L. Grunske, "An architecture-aware approach to hierarchical online failure prediction," in *2016 12th International ACM SIGSOFT Conference on Quality of Software Architectures (QoSA)*, pp. 60–69, April 2016.

[26] Y. Zhang, Z. Zheng, and M. R. Lyu, "An online performance prediction framework for service-oriented systems," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, pp. 1169–1181, Sep. 2014.

[27] L. Bao, C. Wu, X. Bu, N. Ren, and M. Shen, "Performance modeling and workflow scheduling of microservice-based applications in clouds," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, pp. 2114–2129, Sep. 2019.

[28] A. Jindal, V. Podolskiy, and M. Gerndt, "Performance modeling for cloud microservice applications," in *Proceedings of the 2019 ACM/SPEC International Conference on Performance Engineering*, ICPE '19, (New York, NY, USA), pp. 25–32, ACM, 2019.

[29] P. K. Sen, "Estimates of the regression coefficient based on kendall's tau," *Journal of the American statistical association*, vol. 63, no. 324, pp. 1379–1389, 1968.

[30] H. Theil, "A rank-invariant method of linear and polynomial regression analysis," in *Henri Theil's contributions to economics and econometrics*, pp. 345–381, Springer, 1992.

[31] Q. Du, T. Xie, and Y. He, "Anomaly detection and diagnosis for container-based microservices with performance monitoring," in *Algorithms and Architectures for Parallel Processing* (J. Vaidya and J. Li, eds.), (Cham), pp. 560–572, Springer International Publishing, 2018.

[32] Y. Gan, Y. Zhang, K. Hu, D. Cheng, Y. He, M. Pancholi, and C. Delimitrou, "Leveraging deep learning to improve the performance predictability of cloud microservices," 2019.

[33] J. Lin, P. Chen, and Z. Zheng, "Microscope: Pinpoint performance issues with causal graphs in micro-service environments," in *Service-Oriented Computing* (C. Pahl, M. Vukovic, J. Yin, and Q. Yu, eds.), (Cham), pp. 3–20, Springer International Publishing, 2018.

[34] M. Kalisch and P. Bühlmann, "Estimating high-dimensional directed acyclic graphs with the pc-algorithm," *J. Mach. Learn. Res.*, vol. 8, pp. 613–636, May 2007.

[35] P. Wang, J. Xu, M. Ma, W. Lin, D. Pan, Y. Wang, and P. Chen, "Cloudranger: Root cause identification for cloud native systems," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pp. 492–502, May 2018.

[36] L. Wu, J. Tordsson, E. Elmroth, and O. Kao, "MicroRCA: Root Cause Localization of Performance Issues in Microservices," in *IEEE/IFIP Network Operations and Management Symposium (NOMS)*, (Budapest, Hungary), Apr. 2020.

[37] D. A. Menascé, V. A. F. Almeida, R. Fonseca, and M. A. Mendes, "A methodology for workload characterization of e-commerce sites," in *Proceedings of the 1st ACM Conference on Electronic Commerce*, EC '99, (New York, NY, USA), p. 119–128, Association for Computing Machinery, 1999.

[38] K. Mark and L. Csaba, "Analyzing customer behavior model graph (cbmg) using markov chains," in *2007 11th International Conference on Intelligent Engineering Systems*, pp. 71–76, June 2007.

[39] V. A. F. Almeida, "Capacity planning for web services techniques and methodology," in *Performance Evaluation of Complex Systems: Techniques and Tools* (M. C. Calzarossa and S. Tucci, eds.), (Berlin, Heidelberg), pp. 142–157, Springer Berlin Heidelberg, 2002.

[40] N. Roy, A. Dubey, A. Gokhale, and L. Dowdy, "A capacity planning process for performance assurance of component-based distributed systems," in *Proceedings of the 2nd ACM/SPEC International Conference on Performance Engineering*, ICPE '11, (New York, NY, USA), p. 259–270, Association for Computing Machinery, 2011.

[41] N. Roy, A. Dubey, and A. Gokhale, "Efficient autoscaling in the cloud using predictive models for workload forecasting," in *2011 IEEE 4th International Conference on Cloud Computing*, pp. 500–507, July 2011.

[42] C. M. Aderaldo, N. C. Mendonça, C. Pahl, and P. Jamshidi, "Benchmark requirements for microservices architecture research," in *2017 IEEE/ACM 1st International Workshop on Establishing the Community-Wide Infrastructure for Architecture-Based Software Engineering (ECASE)*, pp. 8–13, May 2017.

[43] X. Zhou, X. Peng, T. Xie, J. Sun, C. Xu, C. Ji, and W. Zhao, "Benchmarking microservice systems for software engineering research," in *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*, ICSE '18, (New York, NY, USA), pp. 323–324, ACM, 2018.

[44] "Trainticket git repository (fork of sealabqualitygroup)." `https://github.com/SEALABQualityGroup/train-ticket`. Accessed: 2020-04-06.

[45] D. Di Pompeo, M. Tucci, A. Celi, and R. Eramo, "A microservice reference case study for design-runtime interaction in mde," 2019.

[46] X. Zhou, X. Peng, T. Xie, J. Sun, C. Ji, W. Li, and D. Ding, "Fault analysis and debugging of microservice systems: Industrial survey, benchmark system, and empirical study," *IEEE Transactions on Software Engineering*, pp. 1–1, 2018.

[47] X. Zhou, X. Peng, T. Xie, J. Sun, W. Li, C. Ji, and D. Ding, "Delta debugging microservice systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pp. 802–807, ACM, 2018.

[48] J. von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, and S. Kounev, "Teastore: A micro-service reference application for benchmarking, modeling and resource management research," in *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 223–236, Sep. 2018.

[49] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, "An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, (New York, NY, USA), pp. 3–18, ACM, 2019.

[50] P. Mueller, "Developing a framework for dynamic performance predictions in containerized environments." unpublished practical work (german), 2019.

[51] C. Bergmeir, M. Costantini, and J. M. Benítez, "On the usefulness of cross-validation for directional forecast evaluation," *Computational Statistics & Data Analysis*, vol. 76, pp. 132–143, 2014.

[52] "Wikimedia dumps page count repository." `https://dumps.wikimedia.org/other/pagecounts-raw/`. Accessed: 2020-05-02.

Appendix

A. Parameters of Machine Learning Models

All tested configurations[1]:

- `KNeighborsRegressor`
  - `n_neighbors` = {1-10, 20, 30}
  - `weights` = {*uniform*, *distance*}
  - `p` = {1, 2}
- `BayesianRidgeRegressor`
  - `n_ iter` = {200, 300, 500}
  - `tol` = {1E-2, 1E-3, 1E-4}
  - `alpha_1` = {0.1, 0.01, 1E-3, 1E-4, 1E-6, 1E-9}
  - `alpha_2` = {0.1, 0.01, 1E-3, 1E-4, 1E-6, 1E-9}
  - `lambda_1` = {0.1, 0.01, 1E-3, 1E-4, 1E-6, 1E-9}
  - `lambda_2` = {0.1, 0.01, 1E-3, 1E-4, 1E-6, 1E-9}
- `SVR`
  - `kernel` = {*rbf*, *linear*, *poly*, *sigmoid*}
  - `gamma` = {*scale*, *auto*}
  - `tol` = {1E-2, 1E-3, 1E-4}
  - `C` = {0.1, 0.5, 1, 2, 10}
  - `epsilon` = {0.01, 0.1, 0.2}
- `RandomForestRegressor`
  - `n_estimators` = {1, 5, 10, 20}
  - `criterion` = {*mse*, *mae*}
  - `max_depth` = {*None*, 2, 5, 10}
  - `min_samples_split` = {2, 5}
  - `min_samples_leaf` = {1, 2}

[1] All parameters are named as defined by the `scikit-learn` documentation (version 0.22.2)

Table A.1.: Parameters of Machine Learning Models - Part I

Model Type	KNeighborsRegressor			BayesianRidgeRegressor					
Parameter	n_neighbors	weights	p	n_iter	tol	alpha_1	alpha_2	lambda_1	lambda_2
Scenario 1	2	*distance*	1	200	1E-4	1E-3	1E-3	1E-3	1E-3
Scenario 2	4	*uniform*	2	200	1E-4	1E-3	1E-3	1E-3	1E-3
Scenario 3	1	*uniform*	1	200	1E-4	1E-3	1E-3	1E-3	1E-3
Scenario 4	5	*uniform*	2	200	1E-4	1E-3	1E-3	1E-3	1E-3
Scenario 5	2	*uniform*	1	200	1E-4	1E-3	1E-3	1E-3	1E-3

Table A.2.: Parameters of Machine Learning Models - Part II

Model Type	SVR					RandomForestRegressor				
Parameter	kernel	gamma	tol	C	epsilon	n_estimators	criterion	max_depth	min_samples_split	min_samples_leaf
Scenario 1	*linear*	*scale*	0.01	2	0.01	20	*mae*	10	5	2
Scenario 2	*linear*	*scale*	0.01	0.5	0.01	10	*mse*	10	2	2
Scenario 3	*linear*	*scale*	0.01	0.1	0.01	5	*mae*	*None*	5	2
Scenario 4	*linear*	*scale*	0.01	0.5	0.01	10	*mse*	10	5	1
Scenario 5	*linear*	*scale*	0.01	10	0.01	10	*mse*	*None*	2	2

102

B. Rating Schemes and Thresholds (TrainTicket)

Table B.1.: Rating Schemes for all TrainTicket Endpoints used in this Work - Part I

Service	Endpoint	Response Time with low load [ms]	Lower bound yellow rating [ms]	Lower bound red rating [ms]
ts-assurance-service	getAllAssuranceType	3	40	80
ts-basic-service	queryForStationId	5	40	80
	queryForTravel	25	40	80
ts-config-service	query	3	40	80
ts-contacts-service	findContacts	8	40	80
	getContactsById	8	40	80
ts-execute-service	collected	15	40	80
	execute	15	40	80
ts-food-map-service	getFoodStoresOfStation	2	40	80
	getTrainFoodOfTrip	3	40	80
ts-food-service	createFoodOrder	5	40	80
	getFood	25	40	80
ts-inside-payment-service	pay	25	40	80
ts-login-service	login	15	40	80
ts-notification-service	order_cancel_success	15	40	80
ts-order-other-service	getOrderInfoForSecurity	3	40	80
	calculate	5	40	80
	create	10	40	80
	findAll	3	40	80
	getById	3	40	80

Table B.2.: Rating Schemes for all TrainTicket Endpoints used in this Work - Part II

Service	Endpoint	Response Time with low load [ms]	Lower bound yellow rating [ms]	Lower bound red rating [ms]
ts-order-other-service	getTicketListByDateAndTripId	5	40	80
	modifyOrderStatus	5	40	80
	queryForRefresh	15	40	80
ts-order-service	getOrderInfoForSecurity	3	40	80
	calculate	5	40	80
	create	10	40	80
	findAll	3	40	80
	getById	3	40	80
	getTicketListByDateAndTripId	5	40	80
	modifyOrderStatus	5	40	80
	queryForRefresh	15	40	80
ts-payment-service	pay	8	40	80
ts-preserve-other-service	preserveOther	400	700	1000
ts-preserve-service	preserve	400	700	1000
ts-price-service	query	3	40	80
ts-route-service	queryById	2	40	80
ts-seat-service	getLeftTicketOfInterval	75	125	200
	getSeat	50	100	150
ts-security-service	check	15	40	80
ts-sso-service	findById	3	40	80
	login	5	40	80
	verifyLoginToken	3	40	80

104

Table B.3.: Rating Schemes for all TrainTicket Endpoints used in this Work - Part III

Service	Endpoint	Response Time with low load [ms]	Lower bound yellow rating [ms]	Lower bound red rating [ms]
ts-station-service	exist	2	40	80
	queryByIdBatch	2	40	80
	queryForId	2	40	80
ts-ticketinfo-service	queryForStationId	8	40	80
	queryForTravel	30	70	120
ts-train-service	retrieve	2	40	80
ts-travel-service	getRouteByTripId	5	40	80
	getTrainTypeByTripId	5	40	80
	getTripAllDetailInfo	200	350	550
	query (Scenario 1+4)	250	400	600
	query (Scenario 2+3)	250	600	1000
ts-travel2-service	getRouteByTripId	5	40	80
	getTrainTypeByTripId	5	40	80
	getTripAllDetailInfo	200	350	550
	query (Scenario 1+4)	250	400	600
	query (Scenario 2+3)	250	600	1000
ts-verification-code-service	generate	8	40	80
	verify	2	40	80

C. Rating Schemes and Thresholds (Teastore)

Table C.1.: Rating Schemes for all Teastore Endpoints Used in this Work

Endpoint (Short Name)	Endpoint (Long Name)[a]	Response Time with low load [ms]	Lower bound yellow rating [ms]	Lower bound red rating [ms]
/auth/cart/add	/t.d.t.auth/rest/cart/add/<PARAMS>	3	80	130
/auth/useractions/isloggedin	/t.d.t.auth/rest/useractions/isloggedin	<1	80	130
/auth/useractions/login	/t.d.t.auth/rest/useractions/login	15	80	130
/auth/useractions/logout	/t.d.t.auth/rest/useractions/logout	<1	80	130
/image/getProductImages	/t.d.t.image/rest/image/getProductImages	2	80	130
/image/getWebImages	/t.d.t.image/rest/image/getWebImages	<1	80	130
/persistence/categories	/t.d.t.persistence/rest/categories	2	80	130
/persistence/category	/t.d.t.persistence/rest/categories/<PARAMS>	<1	80	130
/persistence/orders/user	/t.d.t.persistence/rest/orders/user/<PARAMS>	2	80	130
/persistence/products	/t.d.t.persistence/rest/products/<PARAMS>	<1	80	130
/persistence/users	/t.d.t.persistence/rest/users/<PARAMS>	<1	80	130
/persistence/users/name	/t.d.t.persistence/rest/users/name/<PARAMS>	2	80	130
/recommender/recommend	/t.d.t.recommender/rest/recommend	1	80	130
/webui/cart	/t.d.t.webui/cart	28	80	130
/webui/cartAction	/t.d.t.webui/cartAction	5	80	130
/webui/category	/t.d.t.webui/category	40	80	130
/webui/login	/t.d.t.webui/login	6	80	130
/webui/loginAction	/t.d.t.webui/loginAction	9	80	130
/webui/product	/t.d.t.webui/product	40	80	130
/webui/profile	/t.d.t.webui/profile	15	80	130

[a] t.d.t = tools.descartes.teastore

106