Self-Aware Multidimensional Auto-Scaling

Veronika Lesch
Department of Computer Science
Chair for Computer Science II (Software Engineering)

Prof. Dr.-Ing. Samuel Kounev
First Reviewer

Prof. Dr.-Ing. Phuoc Tran-Gia
Second Reviewer

M. Sc. André Bauer
First Advisor

Dipl.-Inform. Nikolas Herbst
Second Advisor

Submission
29. September 2017

www.uni-wuerzburg.de
I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

Würzburg, 29. September 2017

(Veronika Lesch)
Abstract

Modern internet applications are often designed using a layered architecture and are to an increasing degree deployed in public infrastructure cloud environments. In most cases, a fix amount of resources is provisioned, that introduces performance problems when load peaks occur and become inefficient when the load decreases. Therefore, the public cloud provides the possibility to dynamically add or remove resources to the application to match the actual demand. Auto-scalers use this technique to provision and deprovision resources to match the demand in an automatic manner.

Existing auto-scalers often support only single-tier applications and it is possible to instantiate one auto-scaler per tier. However, several problems like bottleneck shifting and oscillations may occur. Therefore, a multi-tier auto-scaler should know all tiers and be able to scale the tiers independently but according to dependencies between the tiers. In addition, when hosting an application in the public cloud, the cloud providers often charge the costs based on instance time. For example Amazon EC2 charges on an hourly basis and rounds all started hours to a full hour. This can increase the costs when using an auto-scaler as resources are provisioned and deprovisioned as required without knowledge about the charging model.

While several multi-tier auto-scaler exist in the literature, they are very limited to specific use-cases. In addition, most of them are either proactive or reactive auto-scalers. Proactive auto-scalers have to rely on their forecast and do not anticipate whether the forecast drifts off the actual demand. Reactive auto-scalers rely on the actual values that can be measured, e.g., request rates. Therefore, if the load increases, the scale up of reactive mechanisms is often to late meaning that the application is already in an overloaded state. Moreover, most of them do not provide a cost-effectiveness logic to control the costs in the public cloud.

In this thesis, a single-tier approach, called Chameleon, is used as basis hybrid auto-scaler and is developed further to support multi-tier auto-scaling. Therefore, the decision logic is changed from measured CPU utilisation to the theoretical queueing theory utilisation per tier. Each tier is modelled as a M/M/n-∞ queue with variable number of service units, i.e., virtual machines. In addition, the decision logic is extended to find scaling decisions for each tier with the knowledge of the scaling at the other tiers. Moreover, a cost-efficiency component is added to make multi-tier Chameleon cost-aware. This component reviews all decisions for cost-efficiency and selects the virtual machine, that introduces less financial loss, to stop.
The experimental evaluation of multi-tier Chameleon is based on two applications, a single-tier and a multi-tier application. Two real world workloads are used: German Wikipedia HTTP requests and requests to the social bookmarking system BibSonomy. The evaluation has shown that it performs better than competing single-tier auto-scalers when scaling the single-tier application. In addition, when evaluating it on the multi-tier application, the effect of bottleneck shifting did not happen with multi-tier Chameleon. In terms of the evaluation metrics, multi-tier Chameleon works for both traces second and third best but the SLO violation rate is in all cases very low. An evaluation using a larger setup with 15, 25 and 10 virtual machines has shown, that the scaling behaviour of multi-tier Chameleon can be transferred to the large setup. A reproducibility evaluation is included comparing both days of the Wikipedia trace against each other. This evaluation has shown that the scaling behaviour at both days is comparable. To evaluate whether to use Telescope or TBATS as forecaster a side-evaluation is conducted showing that Telescope works better in terms of elasticity metrics with multi-tier Chameleon and in a standalone evaluation against TBATS on the BibSonomy trace. Finally, the cost-efficiency component is evaluated using the BibSonomy trace. This evaluation has shown that the cost-efficiency component can reduce the charged costs. Moreover, the real runtime of the virtual machines could be doubled with the cost-efficiency component, resulting in lower SLO violation rates while keeping the costs constant.


zeigen gleiche Characteristika, die ein Auto-Scaler erfassen sollte. Diese Evaluation zeigt, dass Chameleons Verhalten an beiden Tagen sehr ähnlich ist. Eine Randevaluation zeigt, dass die Entscheidung Telescope als Forecaster zu nutzen korrekt war, denn der Alternative Forecaster TBATS zeigt schlechtere Ergebnisse. Schließlich wird die Komponente für Kosteneffizienz evaluiert. Diese Auswertung zeigte, dass die Kosten mit der Komponente leicht gesenkt werden konnten. Zusätzlich wurde die effektiv genutzte Laufzeit der Instanzen verdoppelt, was sich ebenfalls positiv auf die Rate der Antworten auswirkte, welche die Zeitvorgaben überschreiten, ohne dabei höhere Kosten zu verursachen.
Contents

Abstract v

Deutsche Zusammenfassung vii

1. Introduction 1
   1.1. Context 1
   1.2. Motivation 2
   1.3. Idea 2
   1.4. Contribution and Research Questions 3
   1.5. Structure of the Master Thesis 4

2. Background 7
   2.1. Cloud Computing 7
   2.2. Virtualisation 8
   2.3. Load Balancing 8
   2.4. Scalability 9
   2.5. Elasticity 9
   2.6. Self-Aware Computing 10
   2.7. Multi-Tier Architecture 11
   2.8. Performance Analysis of Distributed Systems 11
   2.9. Queueing Theory 12
      2.9.1. Little’s Law 13
      2.9.2. Markov Process 13
      2.9.3. \( M/M/n \) Queueing System 14
   2.10. Deployed Tools 15

3. Related Work 17
   3.1. Single-Tier Auto-Scaling 17
   3.2. Multi-Tier Auto-Scaling 19
      3.2.1. AutoMAP 19
      3.2.2. AGILE 20
      3.2.3. Further Multi-Tier Auto-Scaler 21
      3.2.4. Overview of Related Work 25
   3.3. Discussion 26

4. Approach 27
   4.1. Motivation 27
   4.2. Constraints and Assumptions 28
   4.3. Existing Single-Tier Chameleon 28
   4.4. Multi-Tier Chameleon 29
      4.4.1. Initialisation 32
      4.4.2. Forecasting 32
      4.4.3. Resource Demand Estimation 34
1. Introduction

The thesis starts by explaining the context of this work in Section 1.1. Afterwards, Section 1.2 motivates why a multi-tier auto-scaler is an important achievement for scaling a multi-tier application. Additionally, the reasons for a cost-efficiency component are identified. Afterwards, Section 1.3 describes the idea of this thesis: how the existing single-tier version of Chameleon is changed to support multi-tier applications and cost-efficient scaling. Then, Section 1.4 proposes research questions that should be answered in the course of this thesis and groups them in four categories: Analysis, Development, Realisation, and Evaluation. Finally, Section 1.5 explains the structure of this thesis.

1.1. Context

“Cloud computing has emerged as a popular computing model to support processing large volumetric data using clusters of commodity computers” [RCL09, p. 1]. It increases manageability and efficiency of a web service, as a business does not have to host its own data centre but rather deploy its services inside the public cloud. Amazon EC2, Google Cloud Platform and IBM Bluemix are only a few meaningful cloud platform providers. In most cases, an application runs with a fixed amount of resources. Hereby, two cases have to be taken into account: On the one hand, if the resource supply is higher than the demand, the resource consumption becomes inefficient due to underused resources. On the other hand, if the resource supply is lower than the demand, the availability of the application cannot be guaranteed and the performance may be bad. Therefore, in public clouds, resources can be provisioned on demand to match the actual resource demand. For this purpose, Amazon EC2, for example, offers an auto-scaling mechanism based on thresholds. However, this is a very restricted mechanism only reacting if certain thresholds are exceeded. In addition, a provisioning delay of up to several minutes for virtual machines has to be considered. A prominent cloud user is, e.g., Netflix. It outsources its IT infrastructure to the cloud provider Amazon EC2. Hereby, it uses its own auto-scaling engine called Scryer that predicts future demands and adjusts the resource amount to ensure the best experience for its customers. In the past several auto-scalers have been designed and published in literature. However, there are little use-case scenarios that are publicly available.

1 Amazon EC2: aws.amazon.com/ec2
2 Google Cloud Platform: cloud.google.com
3 IBM Bluemix: www.ibm.com/cloud-computing/bluemix
4 Netflix: www.netflix.com
5 Netflix Scryer: medium.com/netflix-techblog
1.2. Motivation

As there already exist several single-tier auto-scalers, the question arises why a multi-tier auto-scaler is required. The straightforward approach of auto-scaling a multi-tier application is to instantiate a single-tier auto-scaler for every tier separately, and to observe and scale each tier independently. This approach can lead to problems like oscillations and bottleneck shifting. Urgaonkar provides in his paper of 2005 [USCG05] a detailed example for these problems. A three tier architecture is assumed with service rates of 15, 10 and 10.5 req/s per virtual machine at the tiers. Initially, a supply of one virtual machine per tier is provisioned. An arrival rate of 14 requests per second is detected at the first tier. All requests can be served by this tier due to its adequate resource supply. However, the second tier can only process 10 requests per second, drops 4 request and is the bottleneck. Then, the auto-scaler of this tier detects the bottleneck state and provisions a new virtual machine for this tier. Then, the request rate of 14 requests per second can be processed and then passed to the third. This tier is the bottleneck now, with a service rate of 10.5 requests per second and a loss of 3.5 requests per second. Now, the auto-scaler of this tier detects the bottleneck, upscales the tier, and all requests can be served. This example reveals that the bottleneck shifting is an important issue when using multiple independent auto-scalers for a multi-tier application.

A second issue that should be faced in this thesis occurs when hosting an application in the public cloud. The pricing scheme is in most cases based on the time an instance ran. The decision when and which virtual machine to deprovision can have significant impact on the charging costs. For example, in the Amazon EC2 cloud, a virtual machine is charged on an hourly basis. All started hours of runtime are rounded to a full hour no matter whether the virtual machine stopped earlier. For example, if a virtual machine ran for 20 minutes, is then stopped because of a lower arrival rate, and restarted after a few minutes because of an increase in the arrival rate, two hours runtime are charged by Amazon EC2. If an auto-scaler is aware of the pricing scheme of the public cloud, it can find more cost-efficient scaling decisions, i.e., to leave the virtual machine from the above example running even if it is not required for a few minutes. This would save one hour charging costs. However, a proactive mechanism is required for the realisation. In addition, if the auto-scaler is aware of the runtime of each virtual machine, a specific virtual machine can be selected that will be stopped in case of downscaling. For the Amazon EC2 example, the virtual machine that is nearest to the next full hour should be chosen.

1.3. Idea

An approach from the research is the hybrid single-tier auto-scaler Chameleon introduced in the Master thesis of A. Bauer [Bau16]. It implements a reactive and a proactive mechanism and combines forecasts of future arrival rates with the actual observations. Single-tier Chameleon is chosen because experimental results demonstrated superior performance compared to a set of state-of-the-art auto-scalers.

The existing single-tier version of Chameleon is used as the basis for this work. The basic architecture and workflow is adapted, but adjusted to support multi-tier applications. The single-tier version of Chameleon is a hybrid auto-scaler that uses forecasts of future loads, observations of the request rate, and a resource demand estimation. In addition, a performance data repository is used to save observations and holds a DML model of the application. The controller is used to consolidate the observations of the application, the forecasts, and resource demands to find scaling decisions for the actual and future point in time. To support multi-tier applications, the controller is changed to be able
to monitor multiple tiers and to find scaling decisions for each tier independently, but with information about the predecessor tiers. Therefore, the bottleneck shifting effect explained earlier should not take place when using multi-tier Chameleon. In addition to the changed controller, the decision logic is redefined to use queueing theory instead of real measurements of the CPU utilisation, because the CPU utilisation experiments on the public cloud Amazon EC2 have shown, that the CPU utilisation is not reliable. Possible explanations could be migration and hyperthreading effects that can not be controlled. The resource demand estimation and forecast component remain the same as the estimation component can already handle different request classes at different resources and the forecast component has to predict only the arrival rate at the first tier.

A further improvement of multi-tier Chameleon is the integration of a cost-efficiency component. This component is useful when running an application in the public cloud. In the cost-efficiency component, two public clouds and their charging models are considered: Amazon EC2 and Google Cloud Platform. Amazon EC2 charges the runtime of a virtual machine on an hourly basis and therefore rounds all runtimes to the next full hour, no matter if the virtual machine is stopped within this hour. In contrast, Google Cloud Platform charges the first ten minutes of a virtual machine runtime fix on start up and then switches to a charging based on minutes of runtime. The first ten minutes are charged even if the virtual machine is stopped earlier. So, the cost-efficiency component has two tasks: First, it reviews all decisions of the controller and decides whether they are cost-efficient or not. This is done by evaluating the future decisions that are within the charging interval of the public cloud from the actual decision time on. In case of downscaling, it decides which virtual machine should be stopped. For the Amazon EC2 model, the virtual machines nearest to a full hour are chosen. For the Google Cloud Platform, the virtual machines with the longest runtime is chosen to avoid stopping virtual machines that have not run 10 minutes yet.

For the evaluation of multi-tier Chameleon two applications, a single-tier and a multi-tier application, are used. These are driven with two real world traces: German Wikipedia and BibSonomy. In addition, a evaluation on a large setup, a reproducibility analysis and a side-evaluation of the forecasting tools are proposed. The cost-efficiency component is evaluated using the multi-tier application and the BibSonomy trace. The results have shown, that multi-tier Chameleon works best on the single-tier application. At the multi-tier application, multi-tier Chameleon works second or third best, but results in lowest SLO violations and removes the bottleneck shifting effect. The evaluation on the large setup shows that the scaling behaviour of multi-tier Chameleon can be transferred to the large setup. The reproducibility analysis states that the behaviour of multi-tier Chameleon is reproducible for the two days of the Wikipedia trace. The side-evaluation of the two forecasting tools Telescope and TBATS supports the decision to use Telescope as forecaster on the BibSonomy trace. The evaluation of the cost-efficiency component shows that multi-tier Chameleon is able to reduce the costs slightly. In addition the total used virtual machine runtime is doubled. This results in lower SLO violation rates.

1.4. Contribution and Research Questions

In short, this Master thesis contributes multi-tier and cost-efficiency extensions to the single-tier version of Chameleon approach with an extensive evaluation. The below stated research questions are grouped according to the thesis’ structure and are discussed at the end of the according chapter. For the Evaluation the Goal Question Metric (GQM) scheme is used to define the research questions systematically.
1. Analysis of existing auto-scalers (Section 3)
   a) Which single-tier auto-scalers already exist, and can they be classified?
   b) Does multi-tier auto-scalers already exist, and can they be classified?
   c) What is the underlying basis for the scaling decisions?
   d) How can multi-tier Chameleon be separated from these auto-scalers?

2. Development (Section 4)
   a) What are the constraints and assumptions?
   b) How can the request rates at all tiers be determined?
   c) How can these request rates be used to find scaling decisions?
   d) How should the tiers be handled in case of decision process and monitoring?
   e) What are good strategies to become cost-efficient?

3. Realisation (Section 5)
   a) How should the controller be adapted to ensure a reproducible scaling?
   b) Does multi-tier Chameleon need new components?
   c) Which functionalities have to be adapted?

4. Evaluation (Section 6)
   a) Goal: Show that a multi-tier auto-scaler can make better decisions with its overall view than one auto-scaler for every tier
      Question: Does the overall view of a multi-tier auto-scaler enables it to make better decisions than one auto-scaler for every tier?
      Metric: A set of detailed and aggregated elasticity metrics
   b) Goal: Show that multi-tier Chameleon’s scaling behaviour can be adapted to a large environments
      Question: Does the scaling behaviour of multi-tier Chameleon can be transferred to a large scenarios?
      Metric: A set of detailed and aggregated elasticity metrics
   c) Goal: Show that multi-tier Chameleon’s scaling behaviour is reproducible
      Question: Is the scaling behaviour of multi-tier Chameleon reproducible?
      Metric: A set of detailed and aggregated elasticity metrics
   d) Goal: Show that multi-tier Chameleon can be extended to be cost-efficiency.
      Question: Is it possible to enable the auto-scaler to be cost-efficiency?
      Metric: Cost reduction in comparison to an auto-scaler without cost-efficiency
   e) Goal: Show that the costs can be lowered by a significant amount.
      Question: By which amount can the costs be lowered?
      Metric: Cost reduction ratio in comparison to an auto-scaler without cost-efficiency

1.5. Structure of the Master Thesis

The remainder of this thesis is structured as follows: The Chapter 2 summarises important background knowledge and introduces all formulas and fundamentals required in this work. Chapter 3 gives information about existing auto-scalers in the literature.
It is split into two parts, where first, the single-tier auto-scaler is described and then, existing multi-tier auto-scalers are explained and summarised in an overview table. The Chapter 4 provides information about why a multi-tier auto-scaler is required. Then, the constraints and assumptions of multi-tier Chameleon are given. Afterwards, the existing single-tier version of Chameleon is introduced shortly on which this work is based on. Finally, the architecture of multi-tier Chameleon is shown and the workflow of multi-tier Chameleon is defined using a state machine. Fourth, Chapter 5 provides information about the technical details of this work. Fifth, in Chapter 6 the evaluation environment, the competing auto-scalers, the metrics, and the evaluation methodology is described. Afterwards, multi-tier Chameleon is evaluated in six scenarios. Four scenarios are built using a single-tier and a multi-tier application, and two different workload traces: German Wikipedia and of requests measured at the social bookmarking system BibSonomy. In addition to these, a side-evaluation is proposed where the different forecaster are compared and the cost-efficiency component is evaluated. Finally, Chapter 7 summarises the findings of this thesis and gives a view on the work that has to be done in the future.
2. Background

This chapter provides the important background information needed for this work. It starts by defining cloud computing (Section 2.1), virtualisation (Section 2.2) and load balancing (Section 2.3) before giving definitions of scalability (Section 2.4) and elasticity (Section 2.5). Then it introduces the term self-aware computing (Section 2.6) and explains multi-tier applications (Section 2.7). After an overview over performance analysis techniques in Section 2.8, queueing theory (Section 2.9) is explained containing Little’s Law, Markov Processes and the M/M/n queueing system. Finally, the deployed tools LibReDE, DML, WCF, Telescope, ARIMA and TBATS are introduced in Section 2.10.

2.1. Cloud Computing

In 2011, the National Institute of Standards and Technology (NIST) [MG+11] defines cloud computing as:

A model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

Additionally, the committee states five essential characteristics of cloud computing that are summarised in the following. Firstly, on-demand self-service, what means the provisioning of computing capabilities as required without human interaction in an automatic manner [MG+11]. Secondly, broad network access states that the service is “available over the network and accessed through standard mechanisms” [MG+11 p. 2]. Thirdly, resource pooling is the mechanism of aggregating computing resources “to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned” [MG+11 p. 2]. Fourthly, rapid elasticity is a characteristic to provision and release resources automatically to enable scaling [MG+11]. Finally, measured service means that “resource usage can be monitored, controlled and reported providing transparency for both the provider and consumer” [MG+11 p. 2].

Furthermore, the NIST specifies four different deployment models: Private cloud, community cloud, public cloud and hybrid cloud [MG+11]. In a private cloud the resources are “provisioned for exclusive use by a single organization” [MG+11 p. 3]. Likewise, the “data and processes are managed within the organization” [RCL09 p. 45]. The infrastructure of the community cloud is “provisioned for exclusive use by a specific community
of consumers from organisations that have shared concerns” [MG+11 p. 3]. In a public cloud the resources are „provisioned for open use by general public“ [MG+11 p. 3] customers. So, customers provision resources dynamically „from an off-site third-party provider who shares resources“ [RCL09 p. 45]. The hybrid cloud is „a composition of two or more distinct cloud infrastructures [...] that remain unique entities but are bound together“ [MG+11 p. 3].

2.2. Virtualisation

Virtualisation is a „technology that abstracts the coupling between hardware and operating system“ [RCL09 p. 45]. That is, virtualisation abstracts the „logical resources of their underlying physical resources in order to improve agility, flexibility, reduce costs and thus enhance business value“ [RCL09 p. 45]. „A virtualized server is commonly called a virtual machine (VM)” [ZCB10 p. 9]. Besides multiple advantages introduced by virtualisation, it „provides capability of pooling computing resources from clusters of servers and dynamically assigning or reassigning virtual resources to applications on-demand“ [ZCB10 p. 9].

Jain et. al present in their paper from 2013 five „reasons why we need to virtualise resources“ [JP13 p. 28]. First, sharing: „When a resource is too big for a single user, it is best to divide it into multiple virtual pieces“ [JP13 p. 24]. For example today’s multi-core processors can handle multiple virtual machines at once that can then be used by multiple different users [JP13]. Second, isolation: „Multiple users sharing a resource may not trust each other, so it is important to provide isolation among [them]“ [JP13 p. 24]. Virtualisation provides the possibility to isolate the resources of each user so that they are not „able to monitor the activities or interfere with the activities of other users“ [JP13 p. 24]. Third, aggregation: „If the resource is too small, it is possible to construct a large virtual resource that behaves like a large resource“ [JP13 p. 24]. For example, with „a large number of inexpensive unreliable disks [a] large reliable storage“ [JP13 p. 24] can be constructed. Fourth, Dynamics: This is an important feature for the auto-scaling context of this thesis. Due to changing workloads, the resources need to be reallocated in a short time amount to handle the load. When using virtualisation, a faster reconfiguration of the resources in comparison to physical resources is possible [JP13]. Fifth, Ease of management: „Virtual devices are easier to manage because they are software-based and expose a uniform interface through standard abstractions“ [JP13 p. 24]. According to Jain et. al, the last reason is probably the most important one for virtualisation and plays an essential role in the auto-scaling context, as well. If the virtual machines cannot be managed via a standardised API, the starting and stopping of machines would be more complicated.

2.3. Load Balancing

Load balancing is a technique implemented for failure tolerant systems, this means that a system continues working thoroughly even if one or more components have failures [RCL09]. Therefore, „the components are monitored continually and when one becomes non-responsive, the load balancer is informed and no longer sends traffic to it“ [RCL09 p. 45]. Additionally, a load balancer ensures that all components experience the same utilisation. In this work, all instances of one tier have the same resources and therefore, no weights have to be deployed to ensure a fair distribution of the incoming requests and the scheduling strategy Round Robin can be used. Load balancers often provide an interface to trigger health checks. These health checks evaluate whether the load balancer works correctly and produces the same load level on all virtual machines.
2.4. Scalability

Scalability as defined in the work of Herbst et al. [HKR13, p. 3] is the ability of the system to sustain increasing workloads by making use of additional resources.

Scalability can be applied to two terms, application scalability and platform scalability. Application scalability means that the „application maintains its performance goals/SLAs even when [...] its workload increases“ [KHvKR11, p. 3]. In contrast, platform scalability is the „ability of the execution platform to provide [...] as many resources as needed by an application“ [KHvKR11, p. 3]. There are several metrics to evaluate the performance of the scaling mechanism. Amongst others, the „speedup for the same service demand with additional resources and efficiency as a measure of how good the application is using the provided resources“ [KHvKR11, p. 3] is one important example.

The scaling actions can be executed in two different directions, vertical and horizontal. Vertical scaling is the provisioning of „more resources to a given platform node [...] in a way that the platform node can handle larger workload“ [KHvKR11, p. 3]. Whereas horizontal scaling adds „new nodes [...] to a cluster or distributed system in a way that the entire system can handle bigger workloads“ [KHvKR11, p. 4].

There are two strategies for finding scaling decisions: reactive and proactive. Reactive scaling means to monitor the application to optimise and find reactive scaling decisions based on this data. In contrast to this, when using a proactive strategy the scaling mechanism monitors the application as well but forecasts future load based on this monitored data. Then, it uses the scaling decisions based on the forecast values to scale the application before the load peak occurs. This way it can reduce down times caused by unexpected load peaks [LBMAL14].

While scalability is the theoretic basis, auto-scaling is the application for performing the above mentioned functionalities in an automatic manner. An auto-scaling „system should be able to dynamically acquire and release resources [...] required for serving the current workload with a certain level of performance“ [KKMZ17, p. 591]. So, the auto-scaling mechanism monitors the application and finds scaling decisions via a pre-defined logic. Chapter 3 introduces several existing auto-scalers and categorises them into Single-Tier (Section 3.1) and Multi-Tier (Section 3.2) auto-scalers. The tiered architecture that introduces the two categories is explained in Section 2.7.

2.5. Elasticity

Herbst et al. [HKR13, p. 2] define the elasticity of a system as:

The degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an automatic manner, such that at each point in time the available resources match the current demand as closely as possible.

Platform elasticity are the „temporal and quantitative properties of runtime resource provisioning and unprovisioning, performed by the execution platform“ [KHvKR11, p. 7]. The reconfiguration point is the moment „at which platform adaptation [...] becomes effective to that application“ [KHvKR11, p. 7].

Three characteristics of resource elasticity can be defined: effect of reconfiguration, temporal distribution of reconfiguration points, and reaction time. The effect of reconfiguration is „quantified by the amount of added/removed resources and thus expresses the
granularity of possible reconfigurations/adaptations [KHvKR11, p. 8]. The temporal distribution of reconfiguration points describes the "density of reconfiguration points over a possible interval" [KHvKR11, p. 8]. The reaction time is the time required from the trigger of a reconfiguration until the adaptation is completed [KHvKR11]. A metric for evaluating elasticity of a system could be computed by a weighted product of the aforementioned measured characteristics [KHvKR11].

2.6. Self-Aware Computing

In 2017, Kounev et al. define self-aware computing in their book [KKMZ17, p. 5] as follows:

Self-aware computing systems are computing systems that:

1. learn models capturing knowledge about themselves and their environment (such as their structure, design, state, possible actions, and runtime behavior)

2. reason using the models (e.g., predict, analyze, consider, and plan) enabling them to act based on their knowledge and reasoning (e.g., explore, explain, report, suggest, self-adapt, or impact their environment)

in accordance with higher-level goals, which may also be subject to change.

A self-aware computing system may be "built by an entity with [...] higher-level goals in mind. This entity may be a human [...] or a set of humans [...] but it does not necessarily have to be" [KKMZ17, p. 5]. Kounev et al. define two major distinctive characteristics of a self-aware computing system: "The capability to learn models on ongoing basis, capturing knowledge relevant to the purpose" [KKMZ17, p. 5]. And the system "must be able to use the models to reason about this knowledge" [KKMZ17, p. 5]. Both characteristics are driven by higher-level goals. This means that the goals "are at a higher level of abstraction than the system and [...] are not under its direct

Figure 2.1.: Self-aware learning and reasoning loop: LRA-M loop [KKMZ17].
control” [KKMZ17, p. 5]. Kounev et al. describe five types of reasoning [KKMZ17, p. 7]: First, „predict the load of an IT system [...] in a future time horizon”. Second, „predict the system performance (e.g., response time) for a given workload and resource allocation”. Third, „predict the expected impact of a given system adaptation action [...] on the end-to-end system performance”. Fourth, „determine how much resources need to be added to ensure that performance requirements are satisfied under an increasing system workload”. Fifth, „estimate the system’s energy consumption at runtime and compare it to other system configurations in order to select an optimal configuration”.

multi-tier Chameleon can be classified as a self-aware system as it learns models based on the cloud to scale, the arrival process and the service process. With these models it reasons about the future load and the required resource amount to serve all requests. This reasoning results in scaling decisions to adapt the system to varying load.

The self-aware learning and reasoning loop (LRA-M) is a concept „capturing the main activities in a self-aware computing system” [KKMZ17, p. 13]. Figure 2.1 illustrates the self and its interfaces, as well as the activities within the self. These activities are driven by goals and its observations represented as empirical observations. The observations are used for the ongoing process of learning models, which are then used as basis for the system’s reasoning process. The reasoning process may trigger actions affecting the behaviour of the system and possibly impacting the environment [KKMZ17].

2.7. Multi-Tier Architecture

„Though a single-tier architecture has relatively simple structure and is easy to setup, most modern Web sites use a multi-tier architecture” [HHM14, p. 1576]. A multi-tier architecture is a layered architecture with completely isolated layers in matters of functionality and computation [DEF+08]. A multi-tier architecture has several advantages like a better structure, a possible distributed deployment, scalability, reliability and greater failure tolerance [DEF+08, HHM14].

The most common multi-tier architecture consists of a three-tier architecture with a presentation, an application, and a database tier. The presentation tier forms the user interface, the application tier is responsible for all functionalities, computing, and logic. The database tier stores and loads data if requested [DEF+08]. If layers are added to a three-tier architecture it results in a N-tier architecture. For example, the presentation tier could be divided into backend and frontend presentation tier [DEF+08]. For each layer one or multiple virtual machine instances are deployed to serve requests [DEF+08]. Figure 2.2 shows an example model of a three-tier architecture. The first tier depicted on the left side is the presentation tier, the application tier is represented by the tier in the middle, and the database tier is shown on the right as the third tier. The virtual machines on each tier are shown as black boxes, whereas there can be different size and amount of virtual machines among all tiers.

However, the optimal resource allocation among these tiers will be more difficult due to interdependency between the tiers [HHM14]. Moreover, the application could span multiple nodes that makes the appropriate scaling of the application more difficult. This is caused by individual service demands of every virtual machine deployed at a tier. While scaling a tier, all virtual machines of the tier and their associated resource demand have to be considered for the decision making.

2.8. Performance Analysis of Distributed Systems

The performance analysis of distributed systems can be executed on different layers of abstraction each with specific methods as explained in the following. Tran-Gia introduces
three levels of abstractions and proposes five methods in his book from 2005 \[TG05\]. A graphical presentation can be seen in Figure 2.3.

The first method is the *measurement*. If a *system under test (SUT)* is already deployed, measurements can evaluate the operational capabilities of the system. Therefore, workloads of users, peripheral systems and other systems in the network can be used for testing under realistic conditions \[TG05\].

*Hardware simulation* forms the second method that is also based on a real system or network. If a prototype already exists, but users and peripheral systems are not present, the system performance can be evaluated using hardware simulations. Therefore, users and peripheral systems are simulated using finite automaton. With these automaton the user behaviour can be simulated in a realistic way and the performance of the system can be evaluated on user-defined load levels \[TG05\].

In contrast, *Detailed model simulation* and *abstract model simulation* are based on a model of the system. The performance of a system can be evaluated during the conception and development phase of a system. In a detailed model simulation, the model components are on a low level of abstraction. All structural components and flow controls are modelled accurate. This results in a very complex model that can only be analysed using simulation. On the contrary, the abstract model simulation builds a more high-level model of the system where essential characteristics have to be included. These models can be analysed by using simulation that are less intense in calculation time or by using analytic methods \[TG05\].

Finally, *analytic methods* are mathematically exact or approximating methods for models on a high-level of abstraction. These methods allow to examine a larger parameter scope with less computing time. Chameleon uses an abstract model of the cloud infrastructure to reason about needed resources. This model is based on *queueing theory*, which is described in the next section \[TG05\].

### 2.9. Queueing Theory

Queueing Theory is a mathematical way for modeling the performance of queues. Queues consist of one or multiple service units and a waiting queue in front of it. The waiting queue can either be limited or have the theoretical property of infinite capacity. S. Kounev introduces the notation of a queueing network in 2005 as: $A/B/C/K/N/D$. In this context, $A$ is the inter-arrival time distribution, $B$ the service time distribution, and $C$ the number of servers or service units. The latter three parameters are optional, where $K$ is the length of the queue, $N$ the population size and $D$ the service discipline. Typical values for both inter-arrival $A$ and service times $B$ are $M$, $D$, $E_k$ and $G$\[LBMAL14\, p. 11\], where $M$ is a Markovian or poisson arrival process with exponential distribution of service times. $D$ is a deterministic, or constant process, $E_k$ is an Erlang distribution with parameter $k$, and $G$ is a general distribution with a known mean and variance \[LBMAL14\, Kou05\].
Figure 2.3.: Methods for performance analysis of distributed systems [TG05, Per].

2.9.1. Little’s Law

Little’s Law examines any system, which can be an arbitrary selected component of a real system or its model [TG05]. It takes arrival rate and request count into account to supply information about the retention time of requests in the system. Figure 2.4 illustrates the system model.

The following parameters are considered:

\[ \lambda : \text{mean arrival rate of the arrival process} \]
\[ E[X] : \text{mean request count in the system} \]
\[ E[T] : \text{mean retention time of the requests in the system} \]

The following formula relates these parameters and forms Little’s Law.

\[ \lambda \cdot E[T] = E[X] \]

2.9.2. Markov Process

A markov process is a particular class of the stochastic processes [TG05]. A stochastic process is called markov process if its future progress depends only upon the current state of the process [TG05]. Let \( x_n \) be the state of the process at observation time, the markov property can be stated as [TG05]:

\[
P(X(t_{n+1}) = x_{n+1} | X(t_n) = x_n, \ldots, X(t_0) = x_0) = P(X(t_{n+1}) = x_{n+1} | X(t_n) = x_n),
\]

with \( t_0 < t_1 < \ldots < t_n < t_{n+1} \).
So, from the specific process time $t_n$ point of view, the evolution of the process depends only upon the state $[X(t_n) = x_n]$ [TG05]. The evolution of the process in the past ($t < t_n$), is not decisive for the future evolution. This is also called memoryless characteristic [TG05]. The markov property can be referred to the arrival-, service- or state-process of a system [TG05]. If the arrival process is memoryless, the process is called Poisson process.

### 2.9.3. M/M/n Queueing System

In practice, the most common queueing systems for performance analysis are modeled with the markov property. Chameleon assumes a $M/M/n - \infty$ waiting queue for each tier to model the clouds system characteristics. The arrival process is a Poisson-process with a negative exponential inter-arrival time ($A$). Similar, the service time ($B$) has a negative exponential distribution function [TG05]:

$$A(t) = P(A \leq t) = 1 - e^{-\lambda t}, \quad E(A) = \frac{1}{\lambda}$$

$$B(t) = P(B \leq t) = 1 - e^{-\mu t}, \quad E(B) = \frac{1}{\mu}$$

The parameter $\lambda$ is called arrival rate and specifies the mean number of incoming requests per time unit. The parameter $\mu$ specifies the service rate [TG05]. The waiting area is assumed to be infinitely large. An incoming request that cannot be served directly because all service units are occupied, has to wait until one service unit is set free [TG05].

Figure 2.5 illustrates the $M/M/n - \infty$ queueing system with poisson arrival process, infinite waiting area, $n$ service units and negative exponential distributed service time.

![Figure 2.5: M/M/n - \infty queueing system](image)

The Chameleon approach utilises several formulas specified for the $M/M/n-\infty$ queueing system. The parameter $a$ represents the mean number of required service units, while $\rho$ is the utilisation of one service unit [TG05]:

$$a = \frac{\lambda}{\mu} = \lambda \cdot E[B]$$

$$\rho = \frac{a}{n}$$

If the arrival rate is higher than the service rate, i.e., the system exhibits more arrivals than it can serve, the system becomes unstable. In this case, the waiting line becomes infinite large [TG05]. So, the stability condition is defined as:

$$a < n \quad \text{or} \quad \rho < 1$$
multi-tier Chameleon uses the parameter $\rho$ to calculate the demanded virtual machines based on arrival rates, service rates, and the actual number of virtual machines. To get the service rates of the application, the tool LibReDE (Library for Resource Demand Estimation) is used to estimate service times online. In addition, multi-tier Chameleon has a DML model with information about the application that should be scaled. In combination with DML LibReDE can estimate resource demands on every tier separately.

multi-tier Chameleon is a hybrid auto-scaler and therefore, acts reactive and proactive. To find proactive scaling decisions, the future arrival rate needs to be forecasted. Therefore, multi-tier Chameleon adapts the tool WCF to be able to specify the desired forecasting method. WCF provides continuous forecasting results of the specified method. multi-tier Chameleon can access the forecasting methods ARIMA, TBATS and the hybrid forecasting tool Telescope. The integrated tools and methods are introduced in the next section. The way of integration and concrete implementation of these tools and methods are described later in Chapter 4 (Approach) and Chapter 5 (Implementation).

### 2.10. Deployed Tools

The Library for Resource Demand Estimation (LibReDE) is a library of ready-to-use implementations of state-of-the-art approaches to resource demand estimation that can be used for online and offline analysis [Lib]. Based on the actual system and the available monitoring data, the estimation library can automatically determine a set of candidate estimation approaches and execute them [SCZK14, p. 1]. Afterwards, a cross-validation is executed to show which approach produced the best results. This approach can then be chosen for future estimations.

The Descartes Modeling Language (DML) is an architecture-level modeling language for modeling Quality-of-Service (QoS) and resource management related aspects of modern dynamic IT systems, infrastructures, and services [KBH14, p. 6]. At the moment, DML is focused on performance and availability including capacity, responsiveness and resource efficiency aspects [KBH14, p. 6]. Chameleon holds a DML model to include structural knowledge of the application. Based on this information, LibReDE can estimate the resource demands on each tier separately.

The Workload Classification & Forecasting (WCF) tool provides continuous forecast results that are supposed to be interpreted by resource management components to enable proactive resource provisioning [wcf]. The tool implements a novel forecasting methodology that dynamically selects at run-time a suitable forecasting method for a given context [HHKA14, p. 3]. Selection is based on a decision tree that captures the users’ forecasting objectives, requirements of individual forecasting methods and integrates direct feedback cycles [HHKA14, p. 3]. To utilise WCF, the users need to specify their objectives including frequency, horizon, accuracy and overhead [HHKA14]. The tool will then dynamically select a forecasting method. It is able to continuously provide time series of point forecasts of the workload intensity with confidence intervals and forecast accuracy metrics in configurable intervals and with controllable computational overhead during run-time [HHKA14, p. 3].
The tool “Telescope is a hybrid forecasting tool written in R and designed to perform multi-step-ahead forecasts for univariate time series while maintaining a short runtime” [tel]. Its’ “approach is based on time series decomposition and makes use of existing forecasting methods, i.e., ARIMA, ANN, and XGBoost” [ZBH+17, p. 1]. After several preprocessing steps, the given time series is split into season, trend and remainder. “Afterwards, the season and trend forecasting is executed” [ZBH+17, p. 4]. Additionally, the time series is cut into single periods, then clustered using k-means and forecasted. Finally, a composition step is executed where all forecasts are put together [ZBH+17].

ARIMA is a method for time series forecasting using stochastic models [AA13]. It is an extension of the ARMA model that is a combination of AR (Autoregressive) and MA (Moving Average) models. “However, in practice many time series [...] show non stationary behavior” [AA13, p. 21]. Thus, “ARMA models are inadequate to properly describe” [AA13, p. 22] those time series. Therefore, „in ARIMA models a non-stationary time series is made stationary by applying finite differencing of the data points” [AA13, p. 22].

TBATS is a method for time series forecasting based on state space models [DLHS11]. TBATS includes the following key features: Box-Cox transform, ARMA errors, Trend and Seasonal components [DLHS11]. The Seasonal components are modelled using a trigonometric formulation based on Fourier series, which introduces the first T in TBATS. The paper of De Livera [DLHS11] explains the mode of operation detailed.
3. Related Work

This chapter gives an introduction into the related work of auto-scaling. There exist single-tier as well as multi-tier auto-scalers. First, five different methods for single-tier auto-scaling are explained and example approaches are given in Section 3.1. Then, two multi-tier auto-scalers are selected, one reactive (Section 3.2.1) and one proactive (Section 3.2.2), and described in detail. Afterwards further multi-tier auto-scalers are mentioned in Section 3.2.3. Finally, Table 3.2.4 gives an overview over the existing multi-tier auto-scaler with their main properties: method, proactive or reactive, horizontal or vertical scaling, optimisation, cost-effectiveness, and evaluation.

3.1. Single-Tier Auto-Scaling

Auto-scaling can be based on two different strategies as stated in Section 2.4: reactive and proactive. While reactive techniques react „to system changes but do not anticipate to them“, proactive techniques „try to anticipate to future needs and consequently acquire or release resources in advance“ [LBMAL14].

Based on the reactive and proactive strategies, different auto-scaling techniques exist, that can be classified into five categories as stated in Lorido-Botran’s work [LBMAL14]: (i) Static, threshold-based policies, (ii) reinforcement learning, (iii) queueing theory, (iv) control theory and (v) time-series analysis. While threshold-based policies can be clearly classified as reactive strategies, time-series analysis is a proactive strategy. The remaining three techniques can be used in reactive and proactive approaches [LBMAL14]. In the following, the five categories are explained and an example of each category is introduced. The example approaches are as well taken from Lorido-Botran’s review.

First, the static, threshold-based policies vary the „number of VMs in the target application […] according to a set of rules“ [LBMAL14]. There exist typically two rules in this technique, „one for scaling up and one for scaling down“ [LBMAL14]. For each rule several parameters are defined by the user: Upper and lower thresholds as well as time values. Time values define „how long the condition must be met to trigger a scaling action“ [LBMAL14]. M. Maurer et al. present in their work of 2011 a rule-based approach for scaling web applications [MBST11]. They define five escalation levels and according actions that could be done to satisfy SLAs. In addition to this, they introduce three regions representing over-provisioned, under-provisioned and well provisioned states of the application. The regions are characterised by thresholds for resource utilization. Scaling
actions are found to achieve a target value of utilization that is exactly in the center of the well-provisioned region.

The second technique is reinforcement learning that tries to understand and automate „goal-directed learning and decision-making“ [LBMAL14]. It implements „learning through direct interaction between an agent and its environment“ [LBMAL14]. An agent learns from practical experience which action is the best for a particular state of the environment. This learning process tries to optimise a returned reward. In the auto-scaling environment, the auto-scaler is the agent and the environment is a scalable application. Possible actions are addition or removal of resources. The state is the current input workload or performance. The returned reward is the application response time [LBMAL14] that should be minimised. An example auto-scaler that uses reinforcement learning is the one introduced by Rao et al. [RBX+09]. The idea is to monitor the existing virtual machines and find optimal resource configurations based on the performance reward. This reward „is measured by a score which is the ratio of current throughput [...] to a reference throughput plus possible penalties when response time [...] based SLAs [...] are violated“ [RBX+09].

The third method is queueing theory that is the „mathematical study of waiting lines, or queues“ [LBMAL14]. A scalable system is modelled as a service unit and a waiting queue in front where „client requests [...] are enqueued until they are processed“ [LBMAL14]. There are several parameters that specify the model: Arrival rate λ, number of service units and mean service rate µ. The Kendall notation is a standardization how to define a queueing model. As mentioned earlier, Section 2.9 gives a more detailed explanation of queueing theory with the Kendall notation, Markov assumption, Little’s law and several formulas that can be utilised in queueing networks. In 2007, D. Villela et al. introduce in their work a scaling approach based on queueing theory [VPR07]. They characterize the arrival process to be Markovian and model their system as M/G/1/PS queueing system. Afterwards, they find scaling decisions by approximate the needed number of servers per customer.

Fourth, we describe the control theory that can be divided into three types of control systems: open loop, feedback and feed-forward [LBMAL14]. The open-loop theory works without any feedback. It uses „only the current state and its model of the system [to] compute the input to the target system“ [LBMAL14]. When using the feedback theory, the system observes the output and is „able to correct any deviations from the desired value“ [LBMAL14]. The feed-forward theory is a proactive approach. It „predict[s] the behaviour of the system, based on a model, and react[s] before the [output] error actually occurs“ [LBMAL14]. The feed-forward approach may fail because of misleading predictions and therefore, is often combined with a feedback controller. The main objective of control theory is to „maintain the output [...] of the target system [...] to the desired level [...] by adjusting the control input“ [LBMAL14]. Ali-Eldin et al. „model[s] a service deployed in the cloud as a closed loop control system“ [AETE12]. With this approach they build an auto-scaler that can find horizontal scaling decisions.

Finally, time-series analysis is the idea of „making an estimation of the future workload or resource usage“ [LBMAL14]. It is based on a periodical sampling of a certain performance metric to obtain a time-series. Time-series analysis includes two main goals: First, the „forecasting [of] future values of the time-series, based on the last observations“ [LBMAL14]. And second, the identification of the underlying pattern the time-series follows to „extrapolate it to predict future values“ [LBMAL14]. There exist several techniques that can be classified as time-series analysis techniques like moving average, auto-regression, ARMA, exponential smoothing and machine learning. Chandra et al. [CGS03] present in their paper an auto-scaler based on time-series analysis.
techniques. They estimate the future arrival rates and service demands by using auto-regression techniques.

The aforementioned single-tier auto-scalers monitor and scale one specific tier of the target application. However, nowadays web applications often have multiple tiers that cannot be scaled by one single-tier scaler at once. Therefore, multiple single-tier auto-scalers have to be deployed, one for each tier. This introduces overhead to the application and lack an overall view that a multi-tier auto-scaler could have. A multi-tier auto-scaler monitors and scales multiple tiers of one application at once and therefore, is able to manage the application more efficiently. In the next section, several multi-tier auto-scalers are presented.

3.2. Multi-Tier Auto-Scaling

This section introduces several multi-tier auto-scaling approaches. The first two mentioned approaches AutoMAP [Bel15] and AGILE [NSG +13] are described more detailed because they are representatives for a reactive and a proactive auto-scaler. Afterwards, several other multi-tier auto-scalers with reactive and proactive mechanisms are introduced. At the end of this section, Table 3.2.4 summarizes all related multi-tier approaches and gives a short overview of the most important properties.

3.2.1. AutoMAP

In 2015, M Beltrán presented AutoMAP that is the representative for a reactive auto-scaler [Bel15]. Its provisioning model is based on response time triggers, so it implements the rule-based approach. It supports vertical and horizontal scaling as well as cost-effectiveness. Cost-effectiveness in this context means to find the optimal resource configuration via choosing virtual machine image sizes to reduce overall costs. AutoMAP can be centralized or distributed deployed in a cluster where each data centre has its own independent AutoMAP instance. Additionally, there has to be deployed an agent on each host for monitoring. The system consists of four different modules: Controller, Balancer, Informer and Scaler.

The Controller forms the admission control mechanism that considers acceptable risk levels. It calculates the probability of having insufficient capacity to satisfy end user’s service level agreement. If new service requests arrive, it checks for historical data to find resource demands for this kind of request. Based on this information, it decides whether the application can handle this request within SLA constraints or not considering the current available resources. If there is no historical data available, it estimates the service demand from other services or from statistical models.

The Balancer includes the load balancers in front of every tier. This is required to optimize the usage of the virtual resources. “The load balancer module can be hosted and managed in a centralized way running on the same hardware infrastructure than the rest of modules” [Bel15].

The Informer module gathers information at runtime. It collects information about the utilization of physical resources, the virtual machine utilization, application configuration and behaviour as well as load balancing overheads.

The Scaler evaluates the behaviour of the system periodically and decides if a scaling action should be triggered. The scaling can be horizontal or vertical, which means to select different sized machine images. It computes the number and type of virtual machines that are required to provision in each tier. It considers the costs for over-provisioning and reconfigurations and tries to minimize them. The procedure works as
follows: First, it checks if the response time is below a threshold for over-provisioning scenarios and calls the `ReleaseVM` function. This function identifies all tiers that can be candidates for scale-down and decides which tier should be reconfigured. The function uses `Balanced Job Bounds` for average response time estimations. Second, it checks whether the response time is above the threshold given for under-provisioning scenarios and calls the function `CreateVM` if this is true. The `CreateVM` function identifies the tier with highest utilization and adds one virtual machine to it. With `Balanced Job Bounds` it estimates the new response time and iterates until the response time is below the threshold. Finally, the scaler calls the `Optimizer` module that searches for a configuration with the lowest costs for the end user that still fulfils the desired average response time. So, the cost-effectiveness is based on the public cloud providers with different virtual machine image sizes and according costs.

This approach was evaluated using an experimental setup. The multi-tier web benchmark of an auction site RUBiSwas used with three load intensities: low, moderate and high level. There are two evaluation scenarios: Only horizontal scaling allowed, and horizontal and vertical scaling allowed. The results of both scenarios show that AutoMAP is able to scale a multi-tier application dynamically according to the provided SLAs. Additionally, it keeps the response time observed by the user near to the threshold for under-provisioning. When vertical scaling is allowed, the total costs can be reduced in moderate and large workload scenarios but not in low workload ones. The results show that there can be up to 62% economic savings in the moderate workload compared to a static configuration of the application with maximum number of virtual machines [Bel15].

3.2.2. AGILE

The second multi-tier auto-scaler is a proactive mechanism introduced by Nguyen et al. in 2013 [NSG+13]. It uses a `wavelet` based approach for providing medium-term resource demand predictions. These predictions provide enough lead time to start up new application server instances to be prepared for the load peak. Additionally, dynamic virtual machine cloning reduces the start up times.

AGILE’s architecture consists of agile slaves and an agile master. The slaves monitor resource usage and are deployed on every virtual machine. They are very lightweight and therefore, introduce only 1% CPU overhead [NSG+13]. The agile master collects the monitored data from the slaves and manages them. Based on this data, it predicts future CPU demands with a wavelet approach (polynomial curve fitting). AGILE forecasts up to two minutes. In addition to this, the agile master maintains a dynamic resource pressure model for each application using online profiling. Furthermore, AGILE implements pre-copy live cloning for virtual machine replication that provides an immediate performance scale-up whenever needed.

The resource demand estimation based on wavelets works as follows: At each sampling interval it predicts the resource demand over a prediction window of predefined length. It first decomposes the original resource demand time series into sets of wavelet based signals. Then, it performs predictions for each signal separately. Finally, it synthesizes future resource demands by adding up all individual signal predictions.

The agile master’s resource pressure model utilises the predicted resource usage. Additionally, it maintains an application-agnostic model to map the SLO violation rate target into a maximum resource pressure to handle. The model is based on this resource pressure that is the ratio of resource usage to allocation. The model may change over time due to variations in the workload mix and online profiling.

AGILE uses dynamic server pool scaling to reduce the start up time of newly provisioned virtual machines. The server pool contains a set of application virtual machines providing
the same replicated service. Pre-copy live cloning means to instantiate new servers from already running virtual machines that are started after almost all memory has been copied. The copy procedure is run by copying at a minimum rate to finish the cloning process right before the overload happens. This approach reduces the start up time significantly and provides more time for the auto-scaler to find adequate scaling decisions.

For experimental evaluation the RUBiS benchmark was used as well as workloads from the World Cup 98, NASA, EPA and ClarkNet. AGILE is compared to the best alternatives that could be found: PRESS and auto-regression. The results show that AGILE „is statistically significantly better than the PRESS scheme and the auto-regression scheme. It can improve the true positive rate up to 3.42 \times [NSG+13]\) and reduce the false positive rate by up to 0.41 \times [142]\) ×\ [142]\). Moreover, it „achieves consistently the lowest SLO violation rate and the shortest SLO violation time“ [NSG+13]. So, „AGILE is practical for online system management“ [NSG+13].

3.2.3. Further Multi-Tier Auto-Scaler

In the following, further multi-tier auto-scaler that are related will be introduced.

Z. Shen et al. [SSGW11] introduced an elastic resource scaling approach called Cloud-Scale. CloudScale is a proactive approach that supports vertical scaling for voltage scaling and migrates virtual machines out of overloaded hosts. It contains a hybrid prediction approach that consists of signature-driven and state-driven algorithms. It supports scaling of CPU and memory resources as well as the migration of virtual machines. If the SLO violations can be solved by local scaling, scaling of CPU and memory take place. Otherwise, migration of virtual machines out of the over-utilized host is performed until the utilization of the host is reduced. They implemented two models to capture the performance of the application. The signature-driven model runs a fast Fourier-Transformation on monitored time series of resource usage data to identify repeating patterns. The state-driven model raises resource cap by a predefined ratio if under-provisioning is detected. For experimental evaluation the RUBiS benchmark with workloads from the World Cup 1998 and EPA were used. The results show that CloudScale can reduce the total SLO violation time from 351s on local conflict solving to 60s. Additionally, it saves 8-10% total energy consumption and 39-71% workload energy consumption by reducing the CPU frequency [SSGW11].

W. Iqbal et al. [IDCJ11] described in „Adaptive resource provisioning for read-intensive multi-tier applications in the cloud“ a horizontal scaling approach for web and database tier. It is a hybrid approach with a reactive and a predictive model. The reactive model is used to identify scale-up events using heuristics and active profiling of the CPU. The provided SLA is based on response time. It reads web server proxy logs, clusters the entries to dynamic and static content requests and calculates the 95th percentile of the average response time. If the static content requests indicate saturation, the web server tier is scaled. Otherwise, if the dynamic content requests indicate saturation, the database tier is scaled. The predictive model is used to identify over-provisioned resources that is triggered if the reactive model finds satisfied response time requirements for \(k\) consecutive intervals. It implements polynomial regression models to predict the required number of web and database server instances. For experimental evaluation, synthetic workloads are used showing two triangles. There are three scenarios: one with minimum resource allocation, one with high resource allocation and one with adaptive allocation. The results show that the approach can „adaptively add[...] and remove[...] virtual machines to each tier over time“ [IDCJ11]. The scenario with minimal resources leads to system failure and the scenario with high resource allocation leads to over-provisioning. So, the „proposed adaptive resource allocation method is able to maintain a maximum response time SLA while utilizing minimal resources“ [IDCJ11].
Q. Zhu and G. Agrawal [ZA10] introduced an approach based on control theory to scale applications in cloud environments vertically. They focus on a series of interacting service components and scale them independently. The vertical scaling is based on CPU cycles and memory allocation. They take a fixed time-limit and a resource budget into account to maximize the Quality of Service (QoS). They developed a system model to capture the relationship input of the system and its performance. The model uses an auto-regressive-moving-average with exogenous inputs (ARMAX) of second order to represent the system behaviour. The parameters for ARMAX are recalculated with SVM and updated at every interval. Additionally, they developed a resource model for each service component that is trained offline and updated online at every execution interval. The data points contain CPU, memory usage, performance indicators and a time stamp. These points are clustered via the time stamp. Then, the SVM regression learns the relationship between the performance indicators and the resource usage. Finally, a further investigation of the relationship along the time-axis is executed. Zhu and Agrawal evaluated their approach using an emulated testbed and two real world applications: Great Lake Forecasting System (GLFS) and Volume Rendering. They trained their resource model on other hardware than the one for execution and show that the model converges after a short amount of time. After convergence, the model fits quite well for the used hardware. They compared their approach to a static allocation approach where no adaptations can be made. The results show that their adaptive „approach can always perform better than“ [ZA10] the static one. In addition to this, they evaluated the overhead introduced to the application. The new approach introduces only 0.8 - 4% overhead to the system [ZA10].

B. Urgaonkar et al. [USC+08] provided an auto-scaler that can scale horizontally in each tier and is based on an analytical model of internet applications. It receives the incoming request rate and service demands of the individual requests as input and computes the number of needed servers at each tier. The multi-tier application is modelled as a network of queues where each queue represents one tier. The servers are modelled as G/G/1 systems. There is a predictive component to estimate the workload for the next few hours. A reactive component corrects errors in the long-term predictions or to react to unanticipated flash crowds. The predictive component forecasts peak demands over the next hours or days and then, uses the model to determine the number of needed servers. It maintains a history of session arrival rates seen during each hour of day and builds and trims histograms to estimate future demands. The reactive component is invoked on-demand, when under-provisioning is detected, or once every few minutes. It compares the actual arrival rate to the predicted one and corrects the resource configuration if the difference of observed and predicted values is larger than a threshold. For experimental evaluation they used the RUBiS benchmark and workloads from the Word Cup 1998. The results show that the „predictive provisioning works well on typical days“ [USC+08]. The reactive provisioning is needed to handle „flash crowds“ [USC+08]. The actions derived by the reactive component „may lag the [actual] workload“ [USC+08]. The „predictive and reactive mechanisms, and policing are [...] integral components of an effective provisioning“ [USC+08] and enables this approach to handle diverse workloads.

U. Sharma et al. introduce in their work „Provisioning Multi-tier Cloud Applications Using Statistical Bounds on Sojourn Time“ [SST12] an approach based on queueing theory. Their approach can scale the targeted application horizontal and vertical. The vertical scaling is executed by choosing different sized virtual machine images. The multi-tier application is modelled as chain of M/G/1-PS queues and find decisions based on the end-to-end response time. The approach estimates the end-to-end response time by estimating the response times of all queues separately and then computing the overall
response time. In addition to this, the service time distribution is approximated from the service-time histogram and an expectation-maximisation algorithm (EM). The approach has a cost-aware component, that reconfigures the scaling decisions to find the most cost-efficient heterogeneous configuration. It uses a greedy search for the lowest cost configuration with highest utilization. The evaluation is based on simulative and experimental setups. The simulation is implemented using MATLAB. The experimental setup is deployed in the private cloud with the TPC-W and measured workloads. There are two evaluation scenarios for the simulation setup: homogeneous scaling and heterogeneous scaling. The results of the homogeneous setup show that the approach has a worst case provisioning error of 2.75%, whereas the baseline approach has at maximum 140% provisioning error. The results of the heterogeneous run show that the provisioning error can be reduced by 41% in comparison to a baseline approach. Additionally, the cost-efficient algorithm offers cost savings of about 80%. The results of the experimental evaluation deployed in the private cloud show that the provisioning error could be hold below 3%. The approach "accurately captures service time distribution" \cite{SST12}.

J. Bi et al. \cite{BZTW10} provide in their work of 2010 an approach that uses a flexible hybrid queueing model to determine the number of virtual machines at each tier. The architecture consist of a monitor, an analyser, a resource scheduler, and a virtual application executor. The monitor component collects workload and performance metrics like arrival rates, average service time and CPU utilization. The analyser receives and analyses the measurements from the monitor, estimates future workloads, and gets response times of different customers. The resource scheduler sets up the performance analytical model for each tier and uses an optimizer to determine the resource provisioning decisions. The virtual application executor assigns the virtual machine configuration to the system. For each tier, the assumption of a perfect load balancer in front has to be fulfilled. The first tier is modelled as a M/M/n queueing system and the other tiers as multiple M/M/1 systems with FCFS in all tiers. The models are solved to determine the capacity of multiple virtual machines for each tier. Afterwards, it computes the number of required virtual machines at each tier to satisfy the requirements. The experimental evaluation uses the RUBiS benchmark system. The considered resource type is CPU capacity. "The result[s] show[...] that the system throughput can be accurately predicted with [the] model" \cite{BZTW10}. Additionally, "the model is reasonable accurate and can effectively use the CPU resource" \cite{BZTW10}. With this model a "minimized number of virtual machines as well as the maximized CPU resource utilization can be achieved" \cite{BZTW10}.

P. Lama and X. Zhou \cite{LZ09} described their auto-scaling approach in the work "Efficient Server Provisioning with End-to-End Delay Guarantee on Multi-tier Cluster". They provide horizontal scaling that is consisting of 3 components. Firstly, an efficient server provisioning approach. This approach is based on a M/G/1 queueing model which leads to an optimization problem. Then, they use the Pollaczek-Khinchin-formula to calculate the delay of a request at a specific tier. With this information, they can decompose the end-to-end response time to a per tier response time. Secondly, they implemented a self-tuning fuzzy controller that is model independent. This controller determines the number of virtual machines required at each tier based on a fuzzy rule-base. With this mechanism they can give guarantees on the 90th percentile of the end-to-end delay. Thirdly, they use Lyapunov’s direct method to analyse the stability of the scaling decisions. This method is a time-domain method to analyse stability of a non-linear system. Lama and Zhou evaluated their system with a simulation. This simulation show that their „optimization-based approach uses the minimum number of servers“ \cite{LZ09} to guarantee the end-to-end delay. The used number of virtual machines could be reduced by about 20-25%. In addition to this, they show that the impact of
the delay decomposition on the results is workload dependent [LZ09]. Moreover, a „less number of server switchings“ [LZ09] is required which indicates that the stability analysis works properly. Finally, they achieve a higher convergence rate of the model and reduced the end-to-end delay variation [LZ09].

Q. Zhang et al. provide in their work „A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications“ [ZCS07] a theoretical framework. The framework models session-based systems through performance modelling of their transaction-based environment. The analytical model is based on queueing networks and regression is used to approximate CPU demands. The analytical model uses the results derived by the regression module to parametrize the analytic model of queues. Because of the upper limit on the number of simultaneous connections on the web server, a closed system with a network of queues can be assumed. Mean value analysis is used to solve the model efficiently. The input of the model is the think time in queue 0 and the service demands at queue 1 and 2. It provides the average system throughput, average transaction response time, and average queue length for every queue. For evaluation, a simulation and an experimental setup were used. The TPC-W benchmark was deployed as a 3-Tier application. There were three different workload mixes with according cost functions for evaluation. The result show that the „analytic and simulation models predict higher system throughput than the measured one“ [ZCS07], however, this error remains within 15%. Additionally, if the workload mix is changed and the cost functions are the same, the error of the average throughput reaches 80% and 20% respectively [ZCS07]. As mentioned earlier, the approach is only a theoretical framework that provides measured and predicted data to find scaling decisions externally.

In summary, all proposed approaches can scale a multi-tier application. Most of them are based on queueing theory, some use control theory and others analytical methods. Multi-tier Chameleon joins the majority using queueing theory and forms another approach based on leveraging suitable forecast and resource demand estimation techniques. Another similarity of multi-tier Chameleon to most of the proposed approaches is the support of horizontal scaling. Though, some approaches support vertical scaling like AutoMAP [Bel15], CloudScale [SSGW11] and the approach from U. Sharma [SST12]. All approaches do not support migration of virtual machines except for CloudScale. Multi-tier Chameleon again joins this majority and does not support migration. In addition, all approaches are evaluated via experiments except the one from P. Lama. Multi-tier Chameleon is evaluated using a synthetic multi-tier application with real-world traces. The traces are the requests to all German Wikipedia\footnote{Wikipedia Source: \url{dumps.wikimedia.org/other/pagecounts-raw/2013}}\footnotemark[1] pages in December 2013 and HTTP requests to the social bookmarking system Bibsonomy [BHJ+10] during April 2017. So, multi-tier Chameleon is evaluated using experiments, as well. However, all approaches can scale the application either proactive or reactive. Multi-tier Chameleon is a hybrid auto-scaler and therefore, is able to scale the application proactively and has a reactive fall-back mechanism in case the proactive method fails. Moreover, most of the proposed approaches do not support cost-efficiency. The both approaches that support it, optimise via the selection of a virtual machine image from a pool with different sized images. In contrast, multi-tier Chameleon directly implements two charging models: a one phase model and a two phase model. By these models, multi-tier Chameleon can influence the charging costs directly by let virtual machines stay up and selecting specific virtual machines to be stopped. By selecting the virtual machine whose runtime is closest to a charging time, the financial loss is minimised.
### 3.2.4. Overview of Related Work

<table>
<thead>
<tr>
<th>Related Work</th>
<th>Method</th>
<th>proactive (p)</th>
<th>horizontal (h)</th>
<th>vertical (v)</th>
<th>Optimisation</th>
<th>Cost-Efficiency</th>
<th>Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Section 3.2.1</strong></td>
<td><strong>AutoMAP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[Bel15]</td>
<td>response time threshold triggered</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>experimental (RUBiS, synthetic workload)</td>
</tr>
<tr>
<td><strong>Section 3.2.2</strong></td>
<td><strong>AGILE</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[NSG13]</td>
<td>wavelets, resource demand prediction</td>
<td></td>
<td>p</td>
<td>h</td>
<td>no</td>
<td>no</td>
<td>experimental (RUBiS, real workloads)</td>
</tr>
<tr>
<td>Z. Shen</td>
<td>load pattern extraction (FFT)</td>
<td></td>
<td>p</td>
<td>v</td>
<td>yes</td>
<td>no</td>
<td>experimental (RUBiS, real workloads)</td>
</tr>
<tr>
<td>W. Iqbal</td>
<td>analytic (regression)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>experimental (synthetic workload)</td>
</tr>
<tr>
<td>[IDCJ11]</td>
<td>control theory ARMAX</td>
<td></td>
<td>r</td>
<td>v</td>
<td>no</td>
<td>no</td>
<td>experimental (GLFS, Volume Rendering)</td>
</tr>
<tr>
<td>B. Urgaonkar</td>
<td>queueing network server: G/G/1</td>
<td></td>
<td>p</td>
<td>h</td>
<td>no</td>
<td>no</td>
<td>experimental (RUBiS, real workload)</td>
</tr>
<tr>
<td>[USC08]</td>
<td>queueing network first: M/M/c, rest: M/M/1</td>
<td></td>
<td>r</td>
<td>h</td>
<td>no</td>
<td>yes</td>
<td>simulative &amp; experimental (TPC-W, real workload)</td>
</tr>
<tr>
<td>J. Bi</td>
<td>queueing network tier: M/G/1</td>
<td></td>
<td>p</td>
<td>h</td>
<td>no</td>
<td>no</td>
<td>experimental (RUBiS, synthetic workload)</td>
</tr>
<tr>
<td>P. Lama</td>
<td>queueing network tier: M/G/1</td>
<td></td>
<td>p</td>
<td>h</td>
<td>no</td>
<td>no</td>
<td>simulative (synthetic workload)</td>
</tr>
</tbody>
</table>

---

1. Vertical scaling by different VM types with less/higher resource provisioning.
2. Vertical scaling only for voltage scaling via adapting CPU frequency (optional).
3. GLFS = Great Lake Forecasting System
4. Theoretical framework.
3.3. Discussion

In this section the research questions related to the goal Analysis of existing auto-scalers are discussed.

1. Analysis of existing auto-scalers

   a) Which single-tier auto-scalers already exist, and can they be classified?
      In the literature, multiple single-tier auto-scalers exist. They can be classified into five categories based on their techniques: Static, threshold-based policies, reinforcement learning, queueing theory, control theory and time-series analysis.

   b) Does multi-tier auto-scalers already exist, and can they be classified?
      Several multi-tier auto-scalers exist in the literature. They can be classified by their used technique as stated for single-tier auto-scalers. Additional features like cost-efficiency and migration are supported from some of the auto-scalers.

   c) What is the underlying basis for the scaling decisions?
      Most of the existing multi-tier auto-scalers are based on queueing theory and analytic methods and are supporting horizontal scaling. One of the observed auto-scalers allows optimisation based on migration strategies and two offer cost-efficient scaling.

   d) How can multi-tier Chameleon be separated from these auto-scalers?
      Multi-tier Chameleon can be separated from the other auto-scalers by its hybrid architecture that contains a proactive and a reactive cycle. In addition, the cost-efficiency component differs from the other auto-scalers supporting cost-efficiency. Hence, multi-tier Chameleon implements two charging models of public cloud providers and can find scaling decisions that are aware of the models.
4. Approach

This chapter first introduces issues that have to be faced when designing a multi-tier auto-scaler and motivates the approach of this work in Section 4.1. Afterwards, in Section 4.2, constraints and assumptions are shown that are necessary to enable a smooth workflow of multi-tier Chameleon. Then, the existing single-tier auto-scaler Chameleon is introduced in Section 4.3. The multi-tier Chameleon is explained in detail with its architecture in Section 4.4. A state diagram showing the workflow and a description and pseudo codes for each state are proposed. Finally, the configuration parameters of multi-tier Chameleon are introduced and explained in Section 4.5.

4.1. Motivation

As introduced by Urgaonkar in 2005, the „straightforward multi-tier extension is to employ a single-tier approach at each tier of the application“ [USCG05, p. 2]. With this approach, each tier is scaled independently by a single-tier auto-scaler. However, there are problems like oscillations and bottleneck shifting. Urgaonkar gives a detailed example of bottleneck shifting: Assume a three-tier architecture with service rates of 15, 10 and 10.5 req/s per virtual machine at each tier. Initially, there is a supply of one virtual machine per tier. Now, an arrival rate of 14 requests per second arrives at the first tier. This tier can process all requests and they arrive at the second tier. This tier can handle only 10 requests per second and drops 4 requests. At the last tier 10 requests arrive which are all serviced. So, the second tier is the bottleneck and its auto-scaler starts up a new virtual machine. Again, 14 requests arrive at the first tier and are processed by it. The second tier can now process 20 requests per second and handles all arriving requests. The last tier can only process 10.5 requests per second, drops 3.5 requests and is the bottleneck now. This shows that single-tier auto-scaler may find locally optimal solutions for every tier but cannot anticipate scaling actions on other tiers which lead to an increasing arrival rate at this tier. The bottleneck is shifted from second to third tier. Another example is the implementation of a black-box auto-scaler for multi-tier applications [USCG05]. This auto-scaler finds scaling decisions based on the end-to-end response time of the application. If one tier is a bottleneck, the response time increases and exceeds the SLA. Therefore, such an auto-scaler can detect when to scale an application. However, the problem of this auto-scaler is that it cannot determine where to scale the application, in which tier the new resources should be added.

That is why multi-tier Chameleon is designed as a white-box auto-scaler with knowledge about the application, the arrival rates at the tiers and the service time of each virtual
machine per tier. With this information multi-tier Chameleon can model each tier of
the application as a M/M/n-∞ queue and apply the theoretical utilisation formulas.
With this knowledge the required amount of virtual machines per tier can be calculated
exactly and the application can be scaled. In addition to this, multi-tier Chameleon is
a hybrid auto-scaler with, on the one hand, a proactive scaling mechanism to identify
future needs of the application before they occur. And on the other hand, a reactive
fallback mechanism is deployed to scale the application in case the proactive mechanism
fails.

4.2. Constraints and Assumptions

As multi-tier auto-scaling is a complex approach, there are several constraints and as-
sumptions in this work that have to be satisfied to enable a smooth workflow of multi-
tier Chameleon. Some of the constraints are adopted from the single-tier multi-tier
Chameleon introduced by André Bauer [Bau16], others are added for multi-tier support.
The first constraint is a load balancer scheduling with Round Robin logic in front of
every tier. This is very important because it ensures equal load on every virtual machine
in one tier which is the prerequisite for applying queueing theory. In addition, the arrival
rates at the tiers can be requested from the load balancer in front of the tier. The second
constraint is that there is no automatic DML model extraction of the application. The
DML model has to be either created manually or by use of an external tool. PMX
is a tool that automates the extraction of a performance model based on measurement
data [WSKK17].

In addition to these constraints, there are four assumptions for this work. Firstly, the
SLA has to be defined using the response time of the application. Secondly, multi-tier
Chameleon models each tier of the application as a M/M/n-∞ queue with variable num-
ber of service units, in this case virtual machines. Thirdly, multi-tier Chameleon assumes
that the resources in one tier are of the same size. So, there are only homogeneous vir-
tual machines in one tier. However, different tiers can have different virtual machine
sizes. Fourthly, multi-tier Chameleon supports two charging models: a one-phase model
as utilised in Amazon EC2 and a two-phase model as can be seen in Google Cloud. The
one-phase model has one charging time. All runtimes of virtual machines are rounded
to this time even if they are stopped earlier. With the two-phase model the first interval
is charged fix and all runtimes are rounded to this. Then, the second interval starts and
the runtimes are rounded to this time. For example, in the Google Cloud context, the
first charging interval is ten minutes, so all virtual machines are charged for 10 minutes
at start up. After a virtual machine run longer than ten minutes, the charging interval
switches to one minute and for every minute runtime one minute is charged.

4.3. Existing Single-Tier Chameleon

This work is based on the existing single-tier Chameleon introduced in the Master thesis
of André Bauer [Bau16]. The basic workflow is adapted to add functionalities for multi-
tier support. The architecture of the single-tier Chameleon, as can be seen in Figure 4.1,
consists of multiple components: multi-tier Chameleon Controller, Performance Data
Repository, Resource Demand Estimation Component and Forecast Component.

The multi-tier Chameleon Controller forms the central control unit. It monitors the
application, saves observed data into the repository, calls the Resource Demand Estima-
tion Component and the Forecast Component. Then it finds scaling decisions based on

1PMX tool: se.informatik.uni-wuerzburg.de/tools/pmx
this gathered data and scales the application. The Performance Data Repository saves the observed data in a time series storage and contains a DML model to achieve more accurate resource demand estimation results. The Resource Demand Estimation Component receives the observed historical data and estimates the resource demands of every request type at every service unit. Finally, the Forecast Component receives, as well, the observed historical data and forecasts future load to enable a proactive provisioning strategy.

Figure 4.1.: Architecture of the single-tier version of Chameleon [Bau16].

The workflow of the single-tier multi-tier Chameleon is divided into two cycles: Reactive (red line) and proactive (blue dashed line). In the reactive cycle the Controller monitors the cloud in predefined intervals and saves the data to the Performance Data Repository. Then, it uses these information to find scaling decisions to reconfigure the cloud.

The proactive cycle first requests and fetches historical data from the Performance Data Repository. These information are sent to the Resource Demand Estimation Component and the Forecast Component in predefined intervals. The Forecast Component predicts future arrival rates and sends them to the Controller. The Resource Demand Estimation Component determines the actual resource demands for every type of request and at each resource and sends this information to the Controller. Then, it finds scaling decisions based on CPU utilisation and response time with information about SLA conformance, estimated resource demands and forecasted future load. So, there are decisions for CPU utilisation and response time which are combined to ensure SLA conformance. In addition to this, the proactive and reactive decisions are combined to reduce oscillations because of contrary reactive and proactive decisions. Finally, the decisions are timed for execution to provision resources for future load in time.

4.4. Multi-Tier Chameleon

This section first introduces the architecture of the multi-tier Chameleon and shows differences to the single-tier Chameleon. Then, it presents multi-tier Chameleon as a state machine and explains each state in general and with pseudo code algorithms. Afterwards, the parameters of multi-tier Chameleon are summarised and their impact on the found decisions is described. The multi-tier Chameleon adopts the basic architecture and workflow of the single-tier Chameleon. This approach is used as basis and extended
to support multi-tier applications. Figure 4.2 shows the architecture of the multi-tier Chameleon. As seen for the single-tier approach, the red lines show the reactive cycle, the blue dashed lines show the proactive cycle. In addition, multi-tier Chameleon adapted the basic structure of the Controller, Performance Data Repository, Resource Demand Estimation Component and Forecast Component. However, the cloud now contains a multi-tier application and the controller contains a Tiermanagement to control multiple tiers and find scaling decisions with an overall view. Contrary to the single-tier Chameleon, the decisions are based on the theoretical queueing theory utilisation, and not on measured CPU utilisation. Therefore, the decision logic has changed. Additionally, the found decisions are made with knowledge about the other tiers so that an scaling can be triggered earlier on later tiers. This should remove oscillations on later tiers as explained in the motivation of this chapter with the example of Urgaonkar. Moreover, a Cost-Awareness Component has been added, which is useful when running an application in the public cloud. This component, if activated, reviews all decisions found by the Controller and decides whether they are cost-efficient or not. There are two implemented charging strategies that are used by Amazon EC2 and Google Cloud.

In the following, multi-tier Chameleon is introduced as a state machine. Therefore, a state diagram is shown in Figure 4.3. After a brief description of the workflow of multi-tier Chameleon, all states are described in detail and a pseudo code algorithm is shown. The general properties used in these algorithms are summarised in Table 4.2 and the thresholds and properties for the tiers are shown in Table 4.1.

The state diagram starts with the Initialisation state. This state is used to initialise the components and start reactive and proactive cycles. The configuration parameters of multi-tier Chameleon are loaded, the Forecast, Resource Demand Estimation and Monitoring are initialised. Then, the workflow splits into reactive and proactive cycle which are both initialised and started.

The reactive cycle first calls the Monitoring to gather information about the actual
number of running machines and the current request rate at all tiers. Then, these information are used to find decisions for every tier. If the cost-efficiency is activated, all decisions are given to the *Cost-Efficiency* state that checks every decision. Afterwards, the modified decisions are given to the *Scaling* state. If the cost-efficiency is deactivated, all decisions are directly given to the *Scaling* state. This state times the execution of the decisions to start or stop virtual machines right before required. After all decisions are added to the *Scaling* state, multi-tier Chameleon checks whether to stop its execution. If it should keep on running, the execution starts again at the initial state of the reactive cycle where it waits for the next execution interval.

The proactive cycle first calls the *Forecasting* and *Resource Demand Estimation* states parallel and the results are saved. Then, the *Monitoring* is executed where the actual number of virtual machines and arrival rates at the tiers are determined and DML mod-
els are updated. Afterwards, the gathered data is used by the Decision state to find a scaling decision for every tier. These decisions are forwarded to the Cost-Efficiency state if activated, which checks all decisions for cost-efficiency. Afterwards, the modified decisions are given to the Scaling state. If the cost-efficiency is not activated, the decisions are forwarded to the Scaling state directly. Again, after all decisions are added to the Scaling state, multi-tier Chameleon checks whether it should stop. If it should keep on running, the execution starts at the initial state of the proactive cycle and waits until the next proactive interval starts.

4.4.1. Initialisation

This state (Algorithm 1) initialises all components and starts reactive and proactive cycles. First, it loads the properties from the property file. Then, the multi-tier Chameleon parameters are initialised. The Monitoring and Resource Demand Estimation states and in case of activated cost-efficiency this state is initialised, as well. Then the logging and reactive and proactive cycles are started. The most important change in comparison to single-tier multi-tier Chameleon is the initialisation of the Cost-Efficiency.

Algorithm 1: Initialisation State.

<table>
<thead>
<tr>
<th>Input: propertyFile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function init()</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
</tbody>
</table>

4.4.2. Forecasting

The Forecasting state uses time series forecasting techniques to forecast future arrival rates at the first tier. multi-tier Chameleon integrates three different forecasting approaches: ARIMA, TBATS and Telescope. Telescope is a newly integrated approach in comparison with single-tier Chameleon. The forecasting approaches receive the observed historical data of the arrival rates at the first tier and forecast the future arrival rates for a given horizon. The Forecasting State (Algorithm 2) first checks whether there is a forecaster configured. Then, it calls this forecaster with the number of planned intervals as parameter. As result the forecaster returns the future arrival rates. Then, it checks the size of the forecast if there are enough values to calculate the MASE. The MASE is a measure how reliable the forecast is and is calculated using the last observation as forecast. If there are enough values, it fetches the monitored values. If there are enough values as well, it calculates the MASE value by Formula 4.1, where $Y_t$ is the observation at time $t$ and $F_t$ is the forecast of $Y_t$. In the case of to less values of the forecast or
monitored data, the MASE is set to zero. Finally, the forecasted values are stored. In contrast to the single-tier multi-tier Chameleon that used the MAPE (see Formula 4.2), this approach uses the MASE value. The MAPE is based on percentage error and has the disadvantage of being infinite or undefined if $Y_t = 0$ for any $t$ and has a heavier penalty on positive errors [HK06]. Therefore, in this approach the MASE is implemented. It tries to remove the scale of the data by comparing it to a benchmark forecast [HK06]. Here, the naïve method is used. Though, the MASE has problems because relative errors have statistical distributions with undefined mean and infinite variance, it is used in this work because of the scale independence [HK06].

$$MASE = \frac{1}{n-1} \sum_{i=1}^{n} \left| \frac{Y_i - F_i}{Y_i - Y_{i-1}} \right|$$

(4.1)

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \left| \frac{100 \cdot (Y_i - F_i)}{Y_i} \right|$$

(4.2)

Algorithm 2: Forecasting State.

Input: properties, monitoring

Function forecast()
  // check if forecaster exists and execute forecasting
  if forecaster != null then
    values = forecaster.doForecast(planned_intervals)
    // check if enough values exist and calculate MASE
    if values.size ≥ planned_intervals then
      fcValues = values(0 to planned_intervals)
      observedValues = monitoring.getObservedValues()
      if observedValues.size ≥ planned_intervals then
        mase = calculateMase(fcValues, observedValues)
      else
        mase = 0
      end
    else
      mase = 0
    end
  end
  // store forecast values
  storeForecasts(values)
end

The forecasting tool integrated in this work is called Telescope [ZBH+17]. It is a hybrid forecaster written in R that is specifically designed for auto-scaling use cases where univariate time series occur. In addition, it’s runtime is reduced to a minimum to ensure forecasting results just in time. Figure 4.4 presents a simplified illustration of the workflow of Telescope.

First, a preprocessing step is executed on the raw input values. Here, the frequency is estimated using periodograms by applying fast Fourier transformation. The estimated frequency is then used to detect and remove anomalies. Afterwards, decomposition using Loess (STL) is used to split the time series into three components: season, trend and remainder. This can be an additive or multiplicative decomposition depending on the amplitude of seasonal and the trend. Then, season is forecast by simply continuing the seasonality. The trend forecast is computed by using the auto.arima function from
the R package *forecast*. Parallel to the decomposition, the time series with removed anomalies is used to create categorical information by cutting it into single periods. Per period, a feature vector containing two or three characteristics of this period is calculated. Based on these vectors, a clustering is executed using k-means. A cluster is represented by their centroids and a history is created. This history is forecast using artificial neural networks (ANN). To composite the split time series and combine it with the knowledge from the clustering, XGBoost is used. XGBoost is an implementation of gradient boosted decision trees. Finally, XGBoost determines the forecast in two steps. First, it learns a model from historical data of trend, seasonality, centroids and time series. Second, it forecasts using the model and given forecast values from trend, seasonality and cluster labels. In Section 6.9 a side evaluation is proposed where the forecast quality of Telescope is compared to ARIMA and TBATS.

### 4.4.3. Resource Demand Estimation

Because the measurement of resource demands in running systems introduces much overhead to the system, it slows down the operating system and falsifies the measurements [SCBK15 p. 2], the *Resource Demand Estimation* state estimates demands (Algorithm 3). The state first fetches an estimation approach. In this work, the tool LibReDE is used which estimates the service times for time $t - 1$ to $t$ with the given approach. This approach can be: Kalman-Filter, regression, recursive optimisation, approximation with response times and service demand law approach. Then, the services and resources are fetched from the DML model. The forecasts and observed arrival rates are then gathered from *Forecasting and Monitoring* state. Then, for each forecast the proportion of forecast to observed arrival rate is calculated. For each service the new service time is set and the resource it runs on is saved. Then, the residence time is calculated using the service time, the resource and the calculated proportion. Finally, the residence time is stored at the service for the next time.

### 4.4.4. Monitoring

The *Monitoring* state divides into two specific monitorings: *Application* and *Tier Monitoring*. Whilst *Application Monitoring* holds a DML model of the application and
Algorithm 3: Resource Demand Estimation State.

**Input:** properties, LibReDE, DML, forecasts

1. **Function estimate()**
   
   // fetch estimation approach and run estimation
   estimator = getEstimationApproach()
   serviceTimes = LibReDE.runEstimation(estimator, t - 1, t)
   // fetch services and resources from DML model
   services = DML.getServices()
   resources = DML.getResources()
   // fetch forecast and observed values at time t
   forecasts = getForecasts(t)
   observedValues = monitoring.getObservedValues(t)
   // for every forecast value calculate proportion of forecast and observed values
   for i=1; i ≤ forecasting_horizon; i++ do
     proportion = forecasts(i) / observedValues
     // for each service update service time, resource and residence time
     foreach services as service do
       serviceTime = serviceTimes(service)
       resource = resources(service)
       residenceTime = calculateResidenceTime(serviceTime, resource, proportion)
       // store residence time for service and time
       storeResidenceTimeAt(service, t+1, residenceTime)
     end
   end

updates it with observed data, the Tier Monitoring knows how many tiers exist and fetches the arrival rates at the tiers in regular intervals.

4.4.4.1. Application Monitoring

The Application Monitoring (Algorithm 4) observes the application and manages the DML model. It first fetches all virtual machines of the application and updates the DML model accordingly. Then, for each virtual machine it fetches application data like number of requests and departures, residence and response time from the last interval. Then, the requests, departures, residence and response times are extracted from the data and the DML model is updated with the new information about these measures.

4.4.4.2. Tier Monitoring

The Tier Monitoring (Algorithm 5) is responsible for observing the arrival rates at the tiers and saving the measurements to the according data repository. This method is called every monitoring interval, which is defined in the multi-tier Chameleon parameters. First, the Tier Monitoring checks whether a load balancer is installed in front of every tier. Then, for each tier it polls the request rate from the load balancer and sums it up. For each gathered value the counter is incremented. Then it checks whether the actual time modulo the forecasting interval is zero. In other words, if a new forecasting interval starts. Then it checks whether the tier that is processed is the first tier. If this is true, the aggregated arrival rates are divided by the counter which results in an average arrival rate over the last forecasting interval. This average arrival rate is then appended to the
Algorithm 4: Application Monitoring State.

Input: properties, CloudStackController, DML

Function monitor()

// fetch all virtual machines and update DML
vms = CloudStackController.getAllVms()
DML.updateModel(vms)

// for each virtual machine do
foreach vms as vm do
  // fetch application data for this vm
  data = getApplicationData(vm, t - forecast_horizon, t)
  // extract requests, departures, residence and response times
  // and update DML
  request = extractRequestData(data)
  DML.updateRequests(vm, request)
  departures = extractDeparturesData(data)
  DML.updateDepartures(vm, departures)
  measured_residence_time = extractResidenceTimeData(data)
  DML.updateResidenceTime(vm, measured_residence_time)
  measured_response_time = extractResponseTimeData(data)
  DML.updateResponseTime(vm, measured_response_time)
end

A time series of all measured arrival rates and written into a CSV file. Afterwards, the
variables for aggregate arrival rate and counter are set to zero and the algorithm starts
from the beginning.

Algorithm 5: Tier Monitoring State.

Input: properties, tiers, LoadBalancerController, timeSeries, CsvWriter

Function monitor()

// check if LoadBalancerController exists
if LoadBalancerController != null then
  // for each tier fetch request rate, sum up and add one to
  // counter
  foreach tiers as tier do
    request_rate = LoadBalancerController.getRequestRate(tier)
    aggregate = aggregate + request_rate
    counter = counter + 1
    // check if it is time to write measurement and write for
    // first tier
    if time (mod forecasting_interval) = 0 then
      if tier = firstTier then
        timeSeries.append(aggregate / counter)
        CsvWriter.write(aggregate / counter)
      end
    end
  aggregate = 0
  counter = 0
end
end
4.4.5. Decision

The Decision state is divided into a proactive and a reactive decision logic which are called by the according cycle in predefined intervals. Both decision logics find decisions for each tier based on the theoretical queueing theory utilisation $\rho$ as can be seen in Formula 4.4. Therefore, the assumption of a M/M/n-∞ queue per tier is set. If the utilisation exceeds the predefined thresholds of a tier, the needed number of virtual machines is calculated with the Formula 4.5. For this formula a target utilisation is specified in the multi-tier Chameleon parameters for every tier. After all decisions are found, they are given to the Cost-Efficiency state, if configured, else to the Scaling state.

\[
a = \frac{\lambda}{\mu} = \lambda \cdot E[B] \quad \text{(4.3)}
\]

\[
\rho = \frac{a}{n} \quad \text{(4.4)}
\]

\[
n = \frac{a}{\rho} \quad \text{(4.5)}
\]

The decisions found by the proactive decision logic (Algorithm 6) are created using the forecasted arrival rates, estimated resource demands and the number of actual running virtual machines. The horizon of the forecast, how many values are forecasted, is individually set via the multi-tier Chameleon parameter settings. For every forecast, there will be one decision per tier. The arrival rate on the first tier is always the forecasted value. The arrivals on the following tiers is the number of requests served by predecessor tiers in consideration of the forecasted arrivals at the first tier. In other words, if the tiers can be scaled according to the arrival rate at the first tier, the arrival rate at the other tiers equals the one at the first tier. If one tier is scaled to its maximum and therefore is the bottleneck of the application, the arrival rates at later tier is the maximum of requests that can be served by the bottleneck tier. As mentioned earlier, the theoretical utilisation of each tier is calculated using the Formula 4.4. The number of required virtual machines is determined via Formula 4.5 and the predefined target utilisation specified in the multi-tier Chameleon parameters.

The proactive decision logic first creates two variables proactiveDecisions and tierVmNumber. The proactiveDecisions holds all decisions found by the logic and the tierVmNumber maps each tier to a number of virtual machines of the last decision for this tier. Then, for each forecast it creates a new list for decisions and a map called tierArrival. This map contains all tiers as key and the forecasted arrival rates that can be processed by the predecessor tiers after the scaling decision took place. Afterwards, it iterates over all tiers and checks whether the tier is scalable. It fetches the service time from the resource demand estimation and sets the forecast as arrival rate in case of the first tier. If the actual tier is not the first tier, the arrival rate is extracted from the tierArrival variable. After fetching the number of actual running virtual machines, it adds this number to the tierVmNumber variable and calculates the time at which the forecast is planned for. Then, it checks whether service rate, arrival rate and number of virtual machines are greater than zero and calculates the utilisation using the Formula 4.4.

If the calculated utilisation exceeds the upper threshold, the number of needed virtual machines is calculated using the target utilisation and Formula 4.5. If the calculated number of needed virtual machines exceeds the maximum amount of this tier, it is set to the maximum amount. The new number of virtual machines is added to the tierVmNumber and a decision is created for this amount, tier and time and added to the decision list.
Algorithm 6: Proactive Decision State.

**Input:** properties, forecasts, tiers, resourceDemandEstimator, CloudStackController, scaling

```java
Function createDecision()
    proactiveDecisions = new List
    tierVmNumber = new Map  // holds tier → vmNumber
    foreach forecasts as forecast do
        decisions = new List
        tierArrival = new Map  // holds tier → arrivals processed by predecessor
        foreach tiers as tier do
            if tier.isScalable() then
                my = resourceDemandEstimator.getServiceTime(tier)
                if tier.isFirstTier() then
                    lambda = forecast
                else
                    lambda = tierArrival.get(tier)
                end
                n = CloudStackController.getRunningVms(tier)
                tierArrival.add(tier, lambda)
                time = now + numberOfForecast * forecasting-period
                if my > 0 & lambda > 0 & n > 0 then
                    rho = \frac{\lambda}{\mu}n
                    if rho ≥ up_rho_threshold then
                        neededVms = min(\frac{\lambda}{\mu}maxAmount(tier), tier.maxAmount)
                        tierVmNumber.add(tier, neededVms)
                        decisions.add(new Decision(neededVms, tier, time))
                    else if rho ≤ down_rho_threshold then
                        neededVms = max(\frac{\lambda}{\mu}minAmount(tier), tier.minAmount)
                        tierVmNumber.add(tier, neededVms)
                        decisions.add(new Decision(neededVms, tier, time))
                    else
                        tierVmNumber.add(tier, n)
                        decisions.add(new Decision(n, tier, time))
                    end
                else if lambda = 0 then
                    decisions.add(new Decision(tier.getMinAmount, tier, time))
                else
                    tierVmNumber.add(tier, n)
                    decisions.add(new Decision(n, tier, time))
                end
            else if lambda = 0 then
                decisions.add(new Decision(tier.getMinAmount, tier, time))
            else
                tierVmNumber.add(tier, n)
                decisions.add(new Decision(n, tier, time))
            end
        end
    end
    proactiveDecisions.add(decisions);
    combineDecisions(proactiveDecisions)
    if cost_efficiency then
        costEfficiency.processDecisions(proactiveDecisions)
    else
        scaling.processProactiveDecisions(proactiveDecisions)
    end
```


If the calculated utilisation falls below the lower threshold, the number of needed virtual machines is calculated with respect to the minimum amount of this tier. The new amount is added to the `tierVmNumber`. Again, a decision is created for this amount, tier and time and added to the decision list. If the calculated utilisation lies in between the upper and lower threshold, the number of running virtual machines is added to the `tierVmNumber` and a new decision with actual amount, tier and time is created and added to the decision list. In case of one or more of the values service rate, arrival rate and amount of running virtual machines is zero, there is another check, whether the arrival rate is zero. If this is true, a new decision is made with the minimum amount of this tier and added to the decision list. After creating decisions for all tiers, the decision list is added to the `proactiveDecisions`. Then, the function `combineDecisions` is called and the proactive decisions are given to it. Finally, add the decisions to the `Cost-Efficiency` state if enabled, else they are added to the `Scaling` state.

The `combineDecisions` function (Algorithm 7) receives the proactive decisions as input and combines two decisions for one tier. Therefore, it extracts the first decisions and second decisions and creates a new list to save the combined decisions. Then, for each pair of decisions with the same tier it checks multiple possibilities (see Figure 4.5): both decisions do not scale down, both decisions do not scale up and the decisions scale in opposite directions. First, if both decisions do not scale down, it checks whether the amount of the first decision is smaller than the amount of the second decision (top left). If this holds, both decisions are added as they are. If the amount of the first decision is larger than the amount of the second (top centre), the first decision is added and the amount of the second decision is set to the amount of the first. Else, if both decisions do not scale up, it is checked whether the amount of the first decision is smaller than the amount of the second decision (bottom left). If this holds, both decisions are added as they are. If the amount of the second decision is smaller than the amount of the first (bottom centre), the amount of the first decision is set to the amount of the second decision and both are added. If the scale actions are contrary, it is checked whether the first is up and the second is down (top right). In this case, the smoothing factor is extracted from the multi-tier Chameleon parameters. Then, the difference between both is calculated. The new amounts of the decisions are calculated using the original amount minus/plus the difference multiplied by the smoothing factor. Both decisions are added.

![Figure 4.5.: Combination of two proactive decisions.](image-url)
Algorithm 7: Proactive Decision State (combineDecisions).

**Input:** properties, proactiveDecisions

**Function** combineDecisions(proactiveDecisions)

1. `decisionsFirst = proactiveDecisions.getFirst()`  
2. `decisionsSecond = proactiveDecisions.getSecond()`  
3. `combined = new List`  
4. // for each decision pair found for same tier
5. foreach decision pair with same tier as decision1, decision 2 do
   6. if both decisions do not scale down
      7. if decision1.getAction() != Down & decision2.getAction() != Down then
         8. if decision1.getAmount() < decision2.getAmount() then
            9. combined.add(decision1)
            10. combined.add(decision2)
         11. else if decision1.getAmount() ≥ decision2.getAmount() then
            12. decision2.setAmount(decision1.getAmount())
            13. combined.add(decision1)
            14. combined.add(decision2)
      15. else if decision1.getAction() != Up & decision2.getAction() != Up then
         16. if decision1.getAmount() < decision2.getAmount() then
            17. decision1.setAmount(decision2.getAmount())
            18. combined.add(decision1)
            19. combined.add(decision2)
         20. else if decision1.getAmount() ≥ decision2.getAmount() then
            21. combined.add(decision1)
            22. combined.add(decision2)
      23. else if decision1.getAction() = Up & decision2.getAction() = Down then
         24. smoothingFactor = parameter.s
         25. diff = decision1.getAmount() + |decision2.getAmount()|
         26. amount1 = decision1.getAmount() - diff * smoothingFactor
         27. decision1.setAmount(amount1)
         28. amount2 = decision2.getAmount() + diff * smoothingFactor
         29. decision2.setAmount(amount2)
         30. combined.add(decision1)
         31. combined.add(decision2)
      32. else if decision1.getAction() = Down & decision2.getAction() = Up then
         33. smoothingFactor = parameter.s
         34. diff = |decision1.getAmount()| - decision2.getAmount()
         35. amount1 = decision1.getAmount() + diff * smoothingFactor
         36. decision1.setAmount(amount1)
         37. amount2 = decision2.getAmount() - diff * smoothingFactor
         38. decision2.setAmount(amount2)
         39. combined.add(decision1)
         40. combined.add(decision2)
   41. end
42. end
Finally, if the first action is down and the second is up (bottom right), the same procedure is started: fetching the smoothing factor, calculating the difference and calculating the new amount using the difference multiplied by the smoothing factor. Afterwards, both decisions are added.

The reactive decision logic (Algorithm 8) differs in comparison to the proactive one as it uses the measured arrival rate at tier one and not forecasted values. Therefore, there is only one loop iterating over all tiers for this arrival rate and no loop to iterate over multiple forecasted arrival rates at the first tier. So, there is only one decision per tier for the actual time. The arrival rates at later tiers are calculated using the decisions of the predecessor tiers and the arrival rate at the first tier. As the proactive decision logic did, the reactive decision logic creates a new list for the decisions and a map for the arrival rate at each tier. Then, a loop iterates over all tiers and checks whether this tier is scalable. Then, the service time of this tier is fetched from the resource demand estimation and the arrival rate at the first tier is requested from its load balancer. If the actual tier is not the first one, the arrival rate is extracted from the tierArrival map.

After fetching the amount of running virtual machines from the cloud stack controller, the parameters service rate, arrival rate and amount are checked if they are greater than zero and if this holds, the utilisation is calculated using Formula 4.4. If the calculated utilisation exceeds the upper threshold, an upscaling decision is made, if it falls below the lower threshold, a downscaling decision is made. In case of at least one parameter is zero or below a check whether the arrival rate is zero and a decision to scale down the tier to the minimum amount is made. Finally, the decisions are forwarded to either the Cost-Efficiency state if configured, else to the Scaling state.

4.4.6. Cost-Efficiency

The Cost-Efficiency state is reliable for cost-efficient scaling decisions regarding to charging strategies of the public cloud. In this work, two public clouds are considered: Amazon EC2 and Google Cloud. Amazon EC2\(^2\) charges on an hourly basis, this means for every started hour of virtual machine runtime Amazon charges a full hour payment. In contrast, the Google Cloud Platform\(^3\) charges the first ten minutes on start up even if the virtual machine is stopped earlier and then charges per minute. So, whenever a virtual machine run longer than ten minutes, Google charges per minute runtime and does not round to a full hour. All decisions that are made by the Decision state are added immediately to this state. It gathers all decisions and starts processing them timely to start up new virtual machines just in time. By this waiting for execution, it can gather more decisions for the future to which the actual decision can be compared. When comparing to these future decision, multi-tier Chameleon decides whether it is more cost-efficient to let a virtual machine stay up or to stop it now. The cost-efficiency in this work is divided into two blocks: The basic cost-efficiency approach receiving all decisions and the decision handling task that is timed to execute events and combine them to future decisions right before the planned execution time happens.

The basic cost-efficiency approach (Algorithm 9) fetches all made decisions and times a Decision Handling to take future decisions into account. This algorithm is divided into three functions: processing proactive decisions, processing reactive decisions and determining the virtual machines to stop. The processProactiveDecisions function is used to add proactive decisions to the Cost-Efficiency state. It first checks trustworthiness for all decisions. Trustworthiness means that the MASE value is higher than a predefined threshold configured from the multi-tier Chameleon parameters. Then, it gets the

\(^2\)Amazon EC2 charging model: [aws.amazon.com/de/ec2/pricing/on-demand]

\(^3\)Google Cloud Platform charging model: [cloud.google.com/compute/pricing#machinetype]
Algorithm 8: Reactive Decision State.

**Input:** properties, tiers, resourceDemandEstimator, LoadBalancerController, CloutStackController

1. **Function createDecision()**
   2. reactiveDecisions = new List
      // holds tier → arrivals processed by predecessors
   3. tierArrival = new Map
      // for each tier check if it is scalable
   4. foreach tiers as tier do
      5. if tier.isScalable then
         // fetch service and arrival rate and amount of virtual machines
         my = resourceDemandEstimator.getServiceTime(tier)
         if tier.isFirstTier() then
            lambda = LoadBalancerController.getCurrentRequests(tier)
         else
            lambda = tierArrival.get(tier)
         end
         tierArrival.add(tier, lambda)
         n = CloudStackController.getRunningVms(tier)
         // check if values for calculation are greater zero
         if my > 0 & lambda > 0 & n > 0 then
            rho = \( \frac{\text{lambda}}{\text{my} \cdot \text{n}} \)
            if rho ≥ up_rho_threshold then
               // utilisation exceeds upper threshold, upscaling
               neededVms = max(\( \frac{\text{lambda}}{\text{my} \cdot \text{target_rho}} \cdot 1 \))
               reactiveDecisions.add(new Decision(neededVms, tier, now))
            else if rho ≤ down_rho_threshold then
               // utilisation falls below lower threshold, downscaling
               neededVms = max(\( \frac{\text{lambda}}{\text{my} \cdot \text{target_rho}} \cdot 1 \))
               reactiveDecisions.add(new Decision(neededVms, tier, now))
            end
         else if lambda = 0 then
            // if arrival rate is zero downscaling to minimum
            reactiveDecisions.add(new Decision(tier.getMinAmount, Action.DOWN, now))
         end
      6. end
   7. end
   8. if cost_efficiency then
      9. costEfficiency.processDecisions(reactiveDecisions)
   10. else
      11. scaling.processReactiveDecisions(reactiveDecisions)
   12. end

according tier and checks whether there is a decision to compare to for this time and tier.
If this holds, the existing decision that is planned for this time and tier is extracted. If the decision to compare to is proactive and the trustworthiness of the new decision is higher, the decision to compare is replaced by the new one. In the other case, that the decision to compare is reactive, the decision is replaced by the new one. In the case a decision to compare for this time and tier does not exist, the new decision is added to the decision map and a new decisionHandlingTask is scheduled. The delay for this schedule is calculated by the actual time subtracted from a buffer time for calculation and the planned execution time.

The processReactiveDecision iterates over all decisions to be added and checks whether there exists a decision in scale of this decision execution time. This function checks whether there is a proactive decision which is responsible, considering the proactive interval, for the execution time of the actual decision. Then, if there is a proactive decision which is trustworthy, the actual reactive decision is cancelled. If there is no decision in scale, the tier of this decision is extracted and the decision is added to the decision map. Then, the delay is calculated as mentioned in the proactive decision process and a new decisionHandlingTask is scheduled for this execution time with the calculated delay.

Finally, the getVmsToStop function determines which virtual machines should be stopped to have least financial loss. Therefore, it fetches all running virtual machines that could be stopped. Then, if the properties specify the standard (Amazon EC2) charging model, the list of virtual machines is sorted by the runtime of each virtual machine modulo the charging interval. So, the virtual machine with shortest time to fulfil the charging interval is now at first position. If the extended charging model (Google Cloud) is configured, the list of virtual machines is sorted by the runtime of each virtual machine. So, the virtual machine with the largest runtime now is at first position. Now, the virtual machines to stop are the first $x$ virtual machines of the list with $x$ equals the amount to stop. Finally, the selected virtual machines are returned.

![Figure 4.6.: Future look combination.](image)

The Decision Handling (Algorithm 10) is responsible to process the decisions right before the execution time is reached to start virtual machines in time. Additionally, this enables waiting for future decisions to compare to them and find cost-efficient decisions. Therefore, the handle function creates a new map that holds all decisions for all tiers at the target time. It then iterates over all tiers and calls the futureLook function. This function compares the desired decision to all decisions for this tier that are planned for the future. After this process, the possibly modified decisions are added to the Scaling State.

The futureLook function has a look at all decisions planned for the future and decides
Algorithm 9: Cost-Efficiency State.

**Input:** properties, decisions, CloudStackController

1. **Function processProactiveDecisions(decisions)**
   
   ```
   foreach decisions as decision do
   if decision.isTrustworthy then
       tier = decision.getTier()
       decisionsAtTime = decisionMap.get(decision.getTime())
       // decisionMap holds best decision for a time map
       if existsDecisionAtTimeAndTier then
           compareDecision = decisionsAtTime.get(tier)
           if compareDecision.getType ! reactive then
               if compareDecision.getTrustworthiness() < event.getTrustworthiness() then
                   decisionMap.replace(compareDecision, decision)
                   // replaces old compareDecision by new decision
               else
                   decisionMap.replace(compareDecision, decision)
               end
           else
               decisionMap.add(decision)
               delay = max(decision.getTime() - bufferTime - now, 0)
               scheduleDecisionHandlingTask(delay, decision.getTime())
               // schedules handling task for decision time with delay
           end
       end
   end
   ```

2. **Function processReactiveDecisions(decisions)**
   
   ```
   foreach decisions as decision do
   if !existsDecisionInScale(decision) then
       tier = decision.getTier()
       decisionMap.add(decision)
       delay = max(decision.getTime() - bufferTime - now, 0)
       scheduleDecisionHandlingTask(delay, decision.getTime())
       // schedules handling task for decision time delay
   end
   end
   ```

3. **Function getVMsToStop(tier, amount)**
   
   ```
   n = CloudStackController.getRunningVms(tier)
   if basic_cost_efficiency then
       sortByChargingIntervalModulo(n) // sorts collection by runtime
       modulo charging interval (descending)
   else
       sortByRuntimeDesc(n) // sorts collection by runtime
       (descending)
   end
   vmsToStop = subList(0, amount) // gets first amount vms
   return vmsToStop
   ```
whether a downscaling should be processed or not. A downscaling is only processed if the future decisions do not show a stable or higher load level. Figure 4.6 depicts the logic of scale down operations with the future look combination. Therefore, it first gets all target times for which decisions are planned and sorts this list ascending. Then it iterates over this list starting at the execution time of the actual decision to the end of the list. For each future decision it checks whether the standard charging model is enabled and fetches its future look delta. This delta determines how far the future execution times can be. If the extended charging model is enabled, it fetches the first charging interval as future look delta. If the time of the actual decision subtracted from the time of the future decision is smaller than the delta, the number of running virtual machines of this tier is requested. If the amount of the actual decision is smaller than the number of running virtual machines and the future amount is greater or equal the amount of the actual decision, the amount of the actual decision is set to the minimum of amount of actual running virtual machines and future amount. This way, a downscaling which would result in financial loss is avoided.

Algorithm 10: Cost-Efficiency State (Decision Handling).

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Function handle(time)</td>
</tr>
<tr>
<td>2</td>
<td>tierDecisionMap = decisionMap.get(time) // holds all decisions for all tiers at this time</td>
</tr>
<tr>
<td>3</td>
<td>foreach tiers as tier do</td>
</tr>
<tr>
<td>4</td>
<td>futureLook(tier, tierDecisionMap.get(tier))</td>
</tr>
<tr>
<td>5</td>
<td>end</td>
</tr>
<tr>
<td>6</td>
<td>foreach tierDecisionMap as tier do</td>
</tr>
<tr>
<td>7</td>
<td>scaling.processEvent(tierDecisionMap.get(tier), tier) // waits until execution time of decision and scales accordingly</td>
</tr>
<tr>
<td>8</td>
<td>end</td>
</tr>
<tr>
<td>9</td>
<td>Function futureLook(tier, decision)</td>
</tr>
<tr>
<td>10</td>
<td>targetTimes = decisionMap.keys()</td>
</tr>
<tr>
<td>11</td>
<td>sort(targetTimes)</td>
</tr>
<tr>
<td>12</td>
<td>foreach decision from decision.getTime() to end do</td>
</tr>
<tr>
<td>13</td>
<td>futureDecision = decisionMap.getNextDecision()</td>
</tr>
<tr>
<td>14</td>
<td>if basic_cost_efficiency then</td>
</tr>
<tr>
<td>15</td>
<td>futureLookDelta = properties.futureLookDelta // have a look to the future with charging interval length</td>
</tr>
<tr>
<td>16</td>
<td>else</td>
</tr>
<tr>
<td>17</td>
<td>futureLookDelta = properties.firstChargingInt // have a look to the future with maximum first charging interval length</td>
</tr>
<tr>
<td>18</td>
<td>end</td>
</tr>
<tr>
<td>19</td>
<td>if (futureDecision.getTime() - decision.getTime()) ≤ futureLookDelta then</td>
</tr>
<tr>
<td>20</td>
<td>n = CloudStackController.getRunningVms(tier)</td>
</tr>
<tr>
<td>21</td>
<td>if decision.getAmount() &lt; n &amp; futureDecision.getAmount() geq decision.getAmount() then</td>
</tr>
<tr>
<td>22</td>
<td>decision.setAmount(min(n, futureDecision.getAmount()))</td>
</tr>
<tr>
<td>23</td>
<td>end</td>
</tr>
<tr>
<td>24</td>
<td>end</td>
</tr>
<tr>
<td>25</td>
<td>end</td>
</tr>
</tbody>
</table>
4.4.7. Scaling

The Scaling state (Algorithm 11) receives all decisions to plan and execute the scaling. Whenever a decision arrives, it checks whether there is a decision in scope of the new decision. A decision is in scope of another decision if the existing decision is proactive and trustworthy. If the decision to add is reactive the new decision is discarded. Else the new decision is added to be processed. In addition to this, if there are multiple proactive decisions to be execute at one time, the decision with the newest creation is used, because we assume that this decision has better information because of the newer scaling interval.

The function processDecision receives a list containing a list for each forecast. In the list of a forecast the decisions for this execution time are stored. First, a loop iterators over all lists of decisions in the given list. Then for each decision of this list there is a check whether the execution time of the decision is larger than the execution time of the last decision. If this holds, the decision nearest to this decision is fetched and all running virtual machines of the according tier are gathered. If the old decision is not trustworthy or if the old decision amount equals the amount of actual running virtual machines, the old decision is removed and the function addDecision with the new decision is called. Else if the new decision is proactive remove the old one and call the addDecision function with the new decision.

The addDecision method receives one decision and calculates the time to wait until the decision has to be executed. Then, it waits this time and fetches the amount of running virtual machines. If the amount is less than the decision amount, the amount of virtual machines to start is calculated and the tier is upscaled using the cloud stack controller. If the amount of running virtual machines is greater than the decision amount, it is checked whether cost-efficiency is enabled. If this holds, the virtual machines to stop are requested from the Cost-Efficiency State. These virtual machines are then stopped by the cloud stack controller. If no cost-efficiency is configured, a standard downscaling is executed with the cloud stack controller.

4.5. Configuration of Multi-Tier Chameleon

In the following the configuration parameters of multi-tier Chameleon are summarised. Table 4.1 shows how to configure multi-tier Chameleon for multiple tiers and set tier specific thresholds. The table introduces all parameters that have to be set exemplary for the first tier. The other tiers have to be configured in the same way replacing the one at the end of each parameter with the tier number. Afterwards, Table 4.2 presents all tier independent configuration parameters of multi-tier Chameleon, such as the intervals, property files, forecasting, decision and cost-efficiency configuration.

For the parameters in Table 4.2 two assumptions exist that have to be fulfilled to ensure an orderly workflow of multi-tier Chameleon. The monitoring interval specifies the smallest unit of interval. Both, reactive interval and proactive interval must not be smaller than the monitoring interval. In addition, the proactive interval should be a multiple of the reactive interval.

Finally, to sum up the workflow of multi-tier Chameleon, Figure 4.7 depicts an example workflow of Chameleon for one tier with proactive and reactive cycles, the forecasting component and the decisions that are executed after all logic has been executed. At the top, the proactive cycle is depicted. First, the forecaster ($f_1$) is called and calculates the future arrival rates for a given forecast horizon. For each proactive interval two decisions (blue arrows) are found and combined as explained in Section 4.4.5 and Algorithm 7. If
Algorithm 11: Scaling State.

**Input:** properties, decisions, CloudStackController

1. **Function** `processDecisions(decisions)`
   - // for each list of decisions and for each decision
   2. `foreach decisions as list do`
   3. `foreach list as decision do`
   4. `if decision.getTime() > lastDecision.getTime() then`
   5. `oldDecision = getDecisionNearToTime(decision.getTime())`
   6. `n = CloudStackController.getRunningVms(tier)`
   7. `if !oldDecision.isTrustworthy | oldDecision.getAmount() = n then`
   8. `remove(oldDecision)`
   9. `addDecision(decision)`
   10. `else if decision.getType = proactive then`
   11. `remove(oldDecision)`
   12. `addDecision(decision)`
   13. `end`
   14. `end`
   15. `end`
   16. `end`

17. **Function** `addDecision(decision)`

18. `time = decision.getTime() - now`
19. `wait(time) // wait until its time for this decision`
20. `n = CloudStackController.getRunningVms(tier)`
21. `if n < decision.getAmount() then`
22. `amount = min(decision.getAmount() - n, tier.getMaxamount() - n)`
23. `CloudStackController.upscaling(amount, tier)`
24. `else if n > decision.getAmount() then`
25. `amount = max(decision.getAmount() - n, tier.getMinamount() - n)`
26. `if parameters.costEfficiency then`
27. `vmsToStop = costEfficiency.getVmsToStop(tier)`
28. `CloudStackController.stopVms(vmsToStop)`
29. `else`
30. `CloudStackController.downscaling(amount, tier)`
31. `end`
32. `end`

The proactive decisions are not trustworthy or have the same amount as actual running virtual machines (blue arrows with diamond), they are skipped as explained in the Scaling state (see Section 4.4.7). The scope of a proactive decision is the time between this and
Table 4.1.: Configuration of the Tiers of Multi-Tier Chameleon.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>name_tier1</td>
<td>The name of the tier.</td>
</tr>
<tr>
<td>minamount_tier1</td>
<td>The minimum amount of virtual machines for this tier.</td>
</tr>
<tr>
<td>maxamount_tier1</td>
<td>The maximum amount of virtual machines for this tier.</td>
</tr>
<tr>
<td>scalable_tier1</td>
<td>The boolean to set scalability of this tier.</td>
</tr>
<tr>
<td>up_rho_threshold1</td>
<td>The upper threshold for theoretical utilisation of this tier.</td>
</tr>
<tr>
<td>down_rho_threshold1</td>
<td>The lower threshold for theoretical utilisation of this tier.</td>
</tr>
<tr>
<td>target_rho1</td>
<td>The target utilisation for calculation of needed virtual machines for this tier.</td>
</tr>
<tr>
<td>loadbalancer_prop_tier1</td>
<td>The path to the load balancer information file for this tier.</td>
</tr>
</tbody>
</table>

the next proactive decision. Then, the next forecast \((f_2)\) is executed and decisions are created. At \(f_3\) another forecast is started. The forecaster exceeds the waiting time for the results. Therefore, the first found decision is timed in scope of decision \(p_{3,2}\) and skipped as defined in the *Scaling* state (see Section 4.4.7). With this decision logic, reactive decision \(r_6\) to \(r_{13}\) and \(r_{18}\) to \(r_{21}\) are skipped. Afterwards, two proactive decisions, \(p_{4,2}\) and \(p_{2,3}\) are timed at the same execution time and combined as defined in the *Scaling* state (see Section 4.4.7). The reactive cycle is displayed in the centre of the figure. As can be seen, the reactive interval is smaller than the proactive interval. At every reactive interval, one decision is found. If the decision is in scope of a trustworthy proactive decision, the reactive one is skipped as defined in the *Scaling* state (see Section 4.4.7). At the bottom of the figure, the finally executed decisions are depicted. As can be seen, there are different scaling intervals dependent of which decisions are executed. If the decisions are proactive there are two decisions per interval, else the reactive interval is the scaling interval.

Figure 4.7.: Example of multi-tier Chameleon decision logic.
Table 4.2.: Configuration Parameters of Multi-Tier Chameleon.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intervals</td>
<td></td>
</tr>
<tr>
<td>monitoring_interval</td>
<td>The cycle time for the monitoring task.</td>
</tr>
<tr>
<td>reactive_interval</td>
<td>The cycle time of the reactive cycle.</td>
</tr>
<tr>
<td>proactive_interval</td>
<td>The cycle time of the proactive cycle.</td>
</tr>
<tr>
<td>Properties</td>
<td></td>
</tr>
<tr>
<td>cloud_properties</td>
<td>A path to a file with information about the cloud.</td>
</tr>
<tr>
<td>tier_properties</td>
<td>A path to a file with information about the tiers.</td>
</tr>
<tr>
<td>logging_path</td>
<td>A path to a directory where the decisions of each tier are saved.</td>
</tr>
<tr>
<td>Forecast</td>
<td></td>
</tr>
<tr>
<td>forecast_trustworthiness</td>
<td>The minimum MASE value for being a trustworthy forecast.</td>
</tr>
<tr>
<td>accuracy_tolerance</td>
<td>The value of the forecast may deviate from the measured arrival rate.</td>
</tr>
<tr>
<td>planned_intervals</td>
<td>The number of planned proactive intervals each proactive cycle.</td>
</tr>
<tr>
<td>horizon</td>
<td>The number of forecasts in one proactive interval.</td>
</tr>
<tr>
<td>forecast_horizon</td>
<td>The number of forecasted values for each prediction.</td>
</tr>
<tr>
<td>values_per_day</td>
<td>The number of data points per day.</td>
</tr>
<tr>
<td>historical_days</td>
<td>The number of historical days given to the forecaster.</td>
</tr>
<tr>
<td>max_days_in_buffer</td>
<td>The maximum of days in buffer for the prediction.</td>
</tr>
<tr>
<td>confidence_level</td>
<td>The confidence level for the prediction.</td>
</tr>
<tr>
<td>Decision</td>
<td></td>
</tr>
<tr>
<td>s</td>
<td>The shock absorption factor, if two proactive events plan different scaling directions.</td>
</tr>
<tr>
<td>Cost-Efficiency</td>
<td></td>
</tr>
<tr>
<td>cost_efficiency</td>
<td>The boolean to switch cost-efficiency on and off.</td>
</tr>
<tr>
<td>extended_cost_efficiency</td>
<td>The boolean to switch to the extended charging model.</td>
</tr>
<tr>
<td>standard_charging_interval</td>
<td>The charging interval of the basic cost-efficient model.</td>
</tr>
<tr>
<td>ext_charg_int1</td>
<td>The first charging interval of the extended charging model with two phases.</td>
</tr>
<tr>
<td>ext_charg_int2</td>
<td>The second charging interval of the extended charging model with two phases.</td>
</tr>
<tr>
<td>future_look_delta</td>
<td>The amount of time to look into the future when basic cost-efficiency is enabled.</td>
</tr>
<tr>
<td>min_provisioning_time</td>
<td>The minimum provisioning time of one virtual machine.</td>
</tr>
</tbody>
</table>

4.6. Discussion

In this section the research questions related to the goal Development are discussed.

2. Development

a) What are the constraints and assumptions?
There are several constraints defined in this work: (i) A load balancer in front of every tier, (ii) a provided DML model, (iii) response time based SLA, (iv) each tier is modelled as a M/M/n-∞ queue, (v) there is a homogeneous resource setup, (vi) all resources of one tier are of the same size, and (vii) the charging models of Amazon EC2 and Google Cloud Platform are supported.

b) How can the request rates at all tiers be determined?
The request rates at the tiers can be polled from the load balancer in front of every tier.

c) How can these request rates be used to find scaling decisions?
The request rates, estimated resource demands, and amount of virtual machines are used to solve the queueing model to determine the theoretic utilisation. This is used to determine the amount of virtual machines per tier required to serve the load.

d) How should the tiers be handled in case of decision process and monitoring?
All tiers are handled independently in terms of monitoring. For the decision making process, the scaling decisions of predecessor tiers are taken into account to reduce bottleneck shifting effects.

e) What are good strategies to become cost-efficient?
There are two strategies to find cost-efficient scaling decisions. First, if a scale down decision is found, all events planned for the future are evaluated to decide whether the virtual machines should remain running. Second, if a down-scale should happen, the instances that are closest to the next charging interval are stopped to minimise the financial loss.
5. Implementation

This chapter provides technical details of the implementation of multi-tier Chameleon. Therefore, first an overview of the main components of multi-tier Chameleon is shown in Figure 5.1. Afterwards, the components that have been changed for the multi-tier extension are explained. Finally, the cost-efficiency component is shown and described.

Since multi-tier Chameleon is based on the previous work of A. Bauer [Bau16], the main components and basic structure has been adapted. Figure 5.1 gives a schematic...
overview of how these components are linked. In the centre of the figure, the Chameleon Controller is depicted. This component is the entry and endpoint of the workflow of multi-tier Chameleon. It initialises and starts all other components. As described in Section 4.4.2 and Section 4.4.3 the components ResourceDemandEstimation and Forecasting are taken from single-tier Chameleon without modification. In contrast, the decision logic has been modified due to the changed underlying metrics as described in Section 4.4.5. In single-tier Chameleon the real CPU utilisation of the virtual machines is measured and taken to find scaling decisions. This is switched to the theoretical utilisation per tier derived from queueing theory. Even if the changes in this component are profound, a difference in the class structure can not be seen. However, the monitoring and event management have been modified. In addition, the two components cost-efficiency and logging were added. The changes made to these classes and the newly added classes are described in the following.

The monitoring consist of three classes: IMonitoring, Monitoring and TierMonitoring. IMonitoring is the interface defining the functions of the monitoring. These functions can be used for starting and stopping the monitoring as well as to receive information about the number of running virtual machines per tier and the current request rate at a tier. The class Monitoring holds all TierMonitorings in a map and forwards the requests for a specific tier to the according TierMonitoring. The TierMonitoring class holds the tier for which it is responsible as well as the CloudManager that implements the IMonitoring interface. The requests for these information received from the Monitoring class are responded using the CloudManager.

Figure 5.2.: UML diagram of the monitoring component.

The structure of the logging component is comparable to the one of the monitoring component. Its task is to persist the events for each tier at a specified interval to CSV files for later evaluation and fault diagnostics. An interface exists that defines the functions the Logging class has to implement. These are starting and stopping the logging as well as adding events to the logging. The Logging class holds information about the interval on which the events should be logged as well as a TierLogger per tier that is responsible for the events of the according tier. The TierLogging class holds as well information about the logging interval at which the events should be persisted. In addition, it holds the tier for which it is responsible and a list of events for this tier. At every logging interval, the TierLogging starts a LoggingTask that receives an output path as well as a list of events it should persist. This task iterates over all events in the list and persist them to the output file in the CSV file format.

The event management is based on three classes. A central EventManagement class holds one TierEventManager per tier to which all Events are added. In addition, the EventManagement holds a monitoring, a logging and a CloudManager. It provides functions for starting and stopping the management as well as adding and receiving events. Additionally, it provides information about the number of events. These functions forward the requests to the according TierEventManager. The TierEventManager holds a Logging and a list of all events for the according tier. It provides the same functions as the EventManagement and returns the results if requested. The TierEventManager
can hold zero, one or more Events that contain the targeted amount at a specific time. In addition, the scaling action, i.e., up, down or nop, is held as well as the type of the event: reactive or proactive. The trustworthiness of the event is held as well as a tag for which tier the event is created. The Event class provides the common get-functions for all of its variables.

Finally, the cost-efficiency is based on four classes. First, an interface defining the two basic functions getVmsToShutdown() and addEvent(). The first function determines the virtual machines that should be stopped taking the charging model into account. The second function is used to add a new found event to the cost-efficiency component. There exist two classes that implement the ICostEfficiency interface: BasicCostEfficiency and ExtCostEfficiency. The BasicCostEfficiency class is used for the charging model with one interval, e.g., Amazon EC2, that charges per full hour. The ExtCostEfficiency class is used for the charging model with two phases, i.e., the Google Cloud Platform. Here, every virtual machine is charged for ten minutes at start up fix. Afterwards, the charging switches to a minute-by-minute model. Both classes contain the Monitoring, a list of all added events and a list of all virtual machines and their runtimes. For each execution time of the events that are added to the cost-efficiency,
an EventHandling task is timed. This task holds the target time for which all events should be processed, a monitoring and a list of all added events. At a specific interval before the target time, the EventHandling starts processing all events that are listed for the specified target time.

![UML diagram of the cost efficiency component.](image)

Figure 5.5.: UML diagram of the cost efficiency component.

### 5.1. Discussion

In this section the research questions related to the goal Realisation are discussed.

3. Realisation

a) How should the controller be adapted to ensure a reproducible scaling?

In multi-tier Chameleon, only deterministic logic is used to find scaling decisions. The only component that could introduce probabilistic behaviour is the forecasting component but the implemented forecaster have deterministic behaviour, as well. In addition, a logging component is added to ensure reproducible scaling decisions. All decisions are added to the logging component that is responsible for persisting the events to a log file.

b) Does multi-tier Chameleon need new components?

Multi-tier Chameleon has two new components: the Logging and the cost-efficiency component.

c) Which functionalities have to be adapted?

For multi-tier support, several functionalities have changed. The decision logic is now based on the queueing theory utilisation instead of measured CPU utilisations. In addition, the handling of events and monitoring has changed as there are multiple tiers.
6. Evaluation

This chapter provides information about the performance of multi-tier Chameleon. First, the evaluation environment is introduced with the load generator, and the single- and multi-tier applications in Section 6.1. Afterwards, Section 6.2 the competing single-tier auto-scalers are introduced. Then, the user- and system-oriented metrics are explained in Section 6.3. The evaluation on a single-tier application (see Section 6.5 and Section 6.6) is followed by the multi-tier evaluation (see Section 6.7 and Section 6.8). These evaluations contain the evaluation of two data traces, an evaluation with more virtual machines as in the standard scenario (Section 6.7.1), a reproducibility analysis (Section 6.7.2) and a side-evaluation (Section 6.9) in which the two forecaster Telescope and TBATS are compared. Finally, the cost-efficiency is evaluated on the multi-tier application with one data trace in Section 6.10.

6.1. Evaluation Environment

The experiment setup consists of multiple components: the application, a load balancer, the auto-scalers, the load generator, and the experiment controller. In this work, multi-tier Chameleon is evaluated using a single-tier and a multi-tier application. Both applications are deployed in a private cloud environment called CloudStack1. CloudStack manages virtualised Xen-Server hosts and is running in a cluster of 11 identical HP servers. Eight of them are reserved for CloudStack while overbooking and hyperthreading are deactivated. The remaining three servers are used to host the load balancer (Citrix Netscaler2) and the management system of CloudStack, multi-tier Chameleon and the other auto-scalers, as well as the load driver and the experiment controller. The specification of the physical machines is shown in Table 6.1 and the specification of the virtual machines is summarised in Table 6.2.

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Model</td>
<td>HP DL160 Gen9</td>
</tr>
<tr>
<td>CPU</td>
<td>8 cores @ 2.6GHz (Intel E5-2630v3)</td>
</tr>
<tr>
<td>Memory</td>
<td>32GB</td>
</tr>
</tbody>
</table>

1CloudStack: cloudstack.apache.org
2Citrix Netscaler: www.citrix.de/products/netscaler-adc
Table 6.2.: Specification of the virtual machines.

<table>
<thead>
<tr>
<th>Component</th>
<th>Single-Tier</th>
<th>Multi-Tier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operating System</td>
<td>Ubuntu 16.06</td>
<td>Ubuntu 16.06</td>
</tr>
<tr>
<td>vCPU</td>
<td>2 cores</td>
<td>1 core</td>
</tr>
<tr>
<td>Memory</td>
<td>2GB</td>
<td>2GB</td>
</tr>
<tr>
<td>Webserver</td>
<td>Tomcat 7</td>
<td>Tomcat 7</td>
</tr>
</tbody>
</table>

The single-tier application is a CPU-intensive re-implementation using Java Enterprise of the LU worklet from SPEC’s Server Efficiency Rating Tool SERT™2 as benchmark application \[vKBB^+15\]. The application calculates the LU Decomposition of a random generated \(n \times n\) matrix. The parameter \(n\) is a GET parameter of each HTTP request. After calculation, the result is returned. The application is deployed on a Tomcat 7 server as described above.

The multi-tier application is a synthetic application whose workflow is presented in Figure [6.1] and is explained in the following. It contains of a presentation (PT), a business (BT), and a database tier (DT). Each tier has a predefined number of requests that can be served that is controlled using a semaphore. The number of requests that can be handled are 17 (PT), 10 (BT) and 25 (DT). These restrictions are chosen to enable bottleneck shifting effects to take place. If all tiers could handle the same amount of requests, there would be no need to scale them independently.

If a request arrives, the presentation tier checks with a semaphore if the request can be served. If the semaphore blocks the request, the application returns an error 408. If the application can serve the request, i.e., the semaphore lets it pass, the request is forwarded to the business tier. A second semaphore controls the number of incoming requests at this tier. If the semaphore blocks the request, an error is returned. When this error arrives at the presentation tier, it waits the remaining time until one second is over from the first entry of this request to return the error because the semaphore locked for this request. If the presentation tier would return the error response immediately, the semaphore would let one more request pass, which we want to avoid. Then, if the semaphore lets the request at the business tier pass, the request is forwarded to the database tier. And again, if the next semaphore blocks this request, an error response is generated. When this error arrives at the business tier, it has to wait for one second. If the semaphore in the database tier lets the request pass, it has to wait for 200ms to simulate work in progress and is returned correctly after this time. After this correct processed request arrives at the business tier, it has to wait. The time passed from first entry at the business tier until sending the response should become nearly one second. Then, the request is forwarded correctly to the presentation tier, where it is returned to the load generator. The multi-tier application is configured to have ten virtual machines at every tier. However, for one measurement, this is changed to 15 virtual machines at the presentation tier, 25 virtual machines at the business tier and 10 virtual machines at the database tier.

6.2. Single-Tier Auto-Scaler

For evaluating multi-tier Chameleon against other existing auto-scalers, a subset of existing single-tier auto-scalers are selected that are used in the work of \[LAEH^+17\] as well. Both auto-scaler types, reactive and proactive are represented. The reactive auto-scaler is React, proactive auto-scalers are Adapt, Reg, Hist and ConPaaS. In the following, these five single-tier auto-scalers are described.
React, introduced in 2009 by T. Chieu et al. [CMKS09], is a reactive auto-scaler for horizontal scaling. Its architecture consists of a front-end load balancer, a number of web app virtual machines, a provisioning sub-system and a service monitor sub-system with a dynamic scaling algorithm. The algorithm is based on a threshold or a scaling indicator of the web application. This could be the number of active sessions or login sessions in each web application. The monitor gathers these indicators for each virtual machine and calculates the moving average. The scaling algorithm works as follows: First, the current web application virtual machines with active sessions above or below the given threshold are determined. Then, if all virtual machines have active sessions above the threshold a new web application instance is provisioned. If there are virtual machines with active sessions below the threshold and with at least one virtual machine that has no active session, the idle instance is removed. Afterwards, the load factors for all active virtual machines are recalculated and given to the load balancer to re-distribute the request workload to each virtual machine evenly.

Adapt of A. Ali-Eldin et al. is a proactive auto-scaler that supports horizontal scaling [AETE12]. It contains a model of each service of the cloud based on a closed loop control system. ADAPT models the infrastructure using queueing theory as G/G/n stable queue with variable number of servers n. Using this model the authors build two adaptive controllers that are parameter independent. Any performance metric can be used as controlled parameter. It estimates the future service capacity using a gain parameter that determines the estimated change in the workload in the future. The two controllers are built by using two different gain parameters: the „periodical rate of
change of the system load\footnote{\cite[p. 206]{AETE12}} and the „ratio between the change in the load and the average system service rate over time” \footnote{\cite[p. 206]{AETE12}}.

**Reg** from W. Iqbal et al. \cite{IDCJ11}, is a proactive auto-scaler using response times to find scaling decisions to remove bottlenecks. It has a reactive model for the scale up: if the capacity is less than the load, a scale-up decision is made. This is very similar to the behaviour of React. For downsizing, a proactive mechanism decides when and how much to deprovision. Therefore, a regression model is used to predict the number of virtual machines required at each time. This model is updated every time a new observation is added from the reactive mechanism that feeds these observations to the proactive mechanism. The model is recalculated using the complete history of the workload. If the current load is lower than the capacity, the model determines the required amount of virtual machines that can fulfil this load.

**Hist**, introduced by B. Urgaonkar in 2008 \cite{USC08}, is a proactive auto-scaler based on analytical models. It receives the request rate and the service demand of the requests and computes the number of required virtual machines using queueing theory. Each virtual machine is modelled as a G/G/1 system. The predictive provisioning estimates the workload for the next few hours. A reactive provisioning corrects errors in the long-term predictions or to react to unanticipated flash crowds. The predictor uses the past observations of the workload to predict the next period. In addition, it maintains a history of session arrival rates during each hour of the day from the last few days. Then, a histogram is generated for each hour using these observations. The peak load of an hour is estimated through a high percentile of the arrival rate distribution of that hour.

**ConPaaS** from H. Fernandez et al. \cite{FPK14} is a proactive auto-scaler that finds scaling decisions that are based on the throughput of the application. Therefore, the last ten minutes are monitored and the change in throughput is determined. It uses a resource manager and the monitoring engine of the Platform as a Service (PaaS). The system architecture contains a profiler, a predictor, a dynamic load balancer and a scaler. The profiler measures the computing capacity of different configurations and creates profiles for each resource type. The predictor takes monitoring data and uses different time-series analysis techniques to predict the future service demand for the next monitoring window. The techniques can be Linear Regression, ARMA, Exponential Smoothing Holt Winters, Autoregression and Vector Autoregression. The dynamic load balancer adapts the system to the heterogeneity of requests and cloud resources and adjusts weights to distribute incoming traffic. The scaler module works as follows: For scale-out it is checked whether the loaded system state exceeds an upper threshold and the predictor confirms that such a traffic changes remain at least during the next monitoring window. If this holds, new resources are added. The scale-back mechanism checks whether the loaded system state exceeds a lower threshold and the predictor confirms that the load remains at this level at least during the next monitoring window. If this holds, resources are released. In addition, the scaler measures the resource performance. It monitors the workload, generates medium-term predictions and selects the optimal scaling plan based on the trade-off cost versus SLO guarantee.

### 6.3. Metrics

To evaluate the performance of multi-tier Chameleon and compare it to the single-tier auto-scalers introduced above, a set of both system- and user-oriented metrics is used. The system-oriented metrics consist of elasticity metrics, that are used in literature, e.g. in the work of A. Ilyushkin \cite{IAEH17}. Additionally, this thesis evaluates a subset of the single-tier auto-scalers as used in this paper. In addition to the system-oriented metrics explained below, multi-tier Chameleon is evaluated using user-oriented metrics as well.
Therefore, five user-oriented metrics are selected: average amount of virtual machines, amount of adaptations, mean response time, median response time and SLO violations. To evaluate the cost-efficiency of multi-tier Chameleon, the total runtime of all virtual machines is compared to the charged runtime of all virtual machines. In the following, the system-oriented metrics are introduced.

6.3.1. System-Oriented Metrics

The following metrics are based on resource demand and resource supply. The resource demand \((d)\) is the minimum amount of virtual machines required to meet the given service level objective (SLO). The resource supply \((s)\) is the monitored amount of virtual machines, the auto-scaler provisioned to serve incoming requests. The parameter \(T\) specifies the total experiment duration. Both, demand and supply are discrete curves where \(d_t\) and \(s_t\) denote the demand, respectively supply, at time \(t \in [0,T]\).

First, the provisioning accuracy \(\theta\) specifies the relative amount of resources that are under-provisioned, respectively over-provisioned, during the experiment. The under-provisioning accuracy \(\theta_U\) (see Formula 6.1) is the average fraction by which the demand exceeds the supply. The missing virtual machines to meet the SLO are divided by the current demand and normalised by the experiment time. The over-provisioning accuracy \(\theta_O\) (see Formula 6.2) is defined analogous, as the average fraction by which the supply exceeds the demand. These metrics lie in the interval \([0,1]\) and represent percentage values in decimal form. The closer these values are to zero the better the auto-scaler performs.

\[
\theta_U[\%] := \frac{1}{T} \cdot \sum_{t=1}^{T} \frac{\max(d_t - s_t, 0)}{d_t} \triangle t
\]

\[
\theta_O[\%] := \frac{1}{T} \cdot \sum_{t=1}^{T} \frac{\max(s_t - d_t, 0)}{d_t} \triangle t
\]

Second, the wrong provisioning time share \(\tau\) captures the percental time in which the system is under-provisioned or over-provisioned, during the experiment. The accuracy metric allows no reasoning whether the average amount of wrong provisioned virtual machines results from a few big deviations or if it is caused by a constant small deviation. Therefore, the wrong provisioning time share provides insights about the fraction of time in which wrong provisioning occurs. The under-provisioning time share \(\tau_U\) (see Formula 6.3) is the time relative to the measurement duration in which the system has insufficient resources. The over-provisioning time share \(\tau_O\) (see Formula 6.4) is the time in percentage where over-provisioning occurs. The values of this metrics lie in the interval \([0,1]\) and represent percentage values in decimal form. The closer these values are to zero, the better the auto-scaler performs.

\[
\tau_U[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \max(sgn(d_t - s_t), 0) \triangle t
\]

\[
\tau_O[\%] := \frac{100}{T} \cdot \sum_{t=1}^{T} \max(sgn(s_t - d_t), 0) \triangle t
\]

Thirdly, the instability \(v\) describes the time in percentage in which the demand and supply curves move in opposite directions. It is calculated by the fraction of time in
which the demand and supply change have different signs (see Formula 6.5). The result of this metric lies in the interval [0,1], where zero is the best value. This means that the demand and supply curves are always moving in the same direction during experiment.

\[ v[\%] := \frac{1}{T - t_1} \cdot \sum_{t=2}^{T} \min(\left| \text{sgn}(\Delta s_t) - \text{sgn}(\Delta d_t) \right|, 1) \Delta t \quad (6.5) \]

All of these metrics can be calculated per tier. Therefore, another metric is introduced to find one representative value for each tier, the auto-scaler deviation \( \delta \). This metric compares the values of the above metrics with the theoretically optimal auto-scaler. For this comparison, the Minkowski distance metric (see Formula 6.6) is adapted. The vectors are built using the aforementioned metrics. These are specified as percentage, that means, the closer the values are to zero, the better the auto-scaler performs. This is also valid for the distance to the theoretically optimal auto-scaler.

Let \( x, y \in \mathbb{IR} \) and \( 1 \leq p \leq \infty \):

\[ d_p(x, y) := \|x - y\|_p = \left( \sum_{i=1}^{n} |x_i - y_i|^p \right)^{1/p} \quad (6.6) \]

To calculate this metric, the provisioning accuracy and wrong provisioning time share for under- and over-provisioning are aggregated using the mean value of both (see Formula 6.7 and Formula 6.8). The factor \( p \) enables giving more weight either to under- or to over-provisioning. A factor greater than 0.5 would increase weights on under-provisioning that means that under-provisioning is worse than over-provisioning.

\[ \theta[\%] := p \cdot \theta_U + (1 - p) \cdot \theta_O \quad (6.7) \]
\[ \tau[\%] := p \cdot \tau_U + (1 - p) \cdot \tau_O \quad (6.8) \]

When all metrics are calculated, they are aggregated as defined in Formula 6.9. This formula is inferred from the Minkowski distance metric but with other exponents. As instability and wrong provisioning time share both have timely dimensions in their formula and accuracy is defined without timely values, the accuracy and the other metrics have different exponents. This means, to give timely and non-timely aspects the same weight, the provisioning accuracy gets the exponent two.

\[ \delta := (\theta^2 + \tau^1 + v^1)^{1/4} \quad (6.9) \]

This metric is calculated for each tier and states how good the auto-scaler performs on this tier based on system-oriented metrics. The smaller the value the better the auto-scaler performs in comparison with the theoretically optimal auto-scaler. To evaluate the performance over all tiers, the auto-scaler deviations of all tiers are summed up (see Formula 6.10).

\[ \delta_{\text{overall}} := \sum_{\text{tiers}} \delta_{\text{tier}} \quad (6.10) \]
6.3.2. Calculating Instability

In this work, the demand and supply are stair functions, where positive or negative impulses are used. These impulses have no duration and the curve skips to another y-value. So, at every point in the duration of the experiment, demand and supply curves are parallel. Therefore, the instability metric as defined above cannot be calculated straightforward because it would always result in the value zero. To calculate the instability anyway, we introduce a provisioning and a deprovisioning time right before an impulse happens, to artificially generate increasing and decreasing intervals where demand and supply change their amount. Then, the instability can be calculated as defined. Figure 6.2 shows the process of inserting provisioning times before an impulse. First, the demand and supply curves are given. Afterwards, at each impulse is replaced by a diagonal line. This diagonal is created using the provisioning time for upscaling and the deprovisioning time for downscaling. For upscaling, the upper end of the impulse is used as the endpoint of the diagonal. The provisioning time is used as distance to move to the left for finding the x-value of the start point of the diagonal, i.e. the start of the diagonal represents the time when a virtual machine is requested and the end of the diagonal the time when the virtual machine is available. The y-value is defined as the y-value of the lower end of the impulse. This way, a triangle is created and the diagonal is inserted. For downscaling the calculation works analogously. The lower end point of the impulse is used as end point of the diagonal. Then, the deprovisioning time is used to find the x-value of the start point and the upper end of the impulse is used to define the y-value. This is done for all impulses in demand and supply curves. In case the inserted diagonals intersect each other as can be seen in the center of Figure 6.2 on the right, there has to be additional handling that is described in the following. After both are processed, they can be merged and the instability can be calculated. At the bottom of Figure 6.2, the red areas show the unstable phases, where demand and supply do not change in the same direction, that are considered in the metric. Here, it does not matter how large the gradient is. Instead, only the sign of the gradient is important, whether

![Diagram 6.2: Preparations for Instability.](image-url)
the function increases or decreases.

The additional handling of intersection diagonals is described in the following. In general, the intersection point has to be calculated and the y-value is set to the y-value of the lower end points of the diagonal as shown at the bottom in Figure 6.3. Therefore, the mathematical equation of a line is used (see Formula [6.11]). This equation of a line can be defined using the first derivation of it as can be seen in Formula [6.12]. Therefore, the y-intersect is needed, that can be calculated by using the negative gradient that times the x-value (Formula [6.13]). Finally, the derivation of the equation of a line is the gradient (see Formula [6.14]).

\[
\begin{align*}
  f(x) &= m \cdot x + c \quad (6.11) \\
  f(x) &= f'(x) \cdot x + f(0) \quad (6.12) \\
  f(0) &= (-m) \cdot (x) \quad (6.13) \\
  f'(x) &= m \quad (6.14)
\end{align*}
\]

Four points are given to define two diagonals. These are used to find the intersection point. To calculate the intersection point, two points per diagonal are given. Figure 6.3 visualises the given points, the diagonals, the intersection and the resulting point after calculation. The points \(a\) and \(b\) are the original points. The points \(a'\) and \(b'\) are the newly inserted points for creating the diagonal.

\[
\begin{align*}
  f(a_x) &= a_y \\
  f(a'_x) &= a'_y \\
  f(b_x) &= b_y \\
  f(b'_x) &= b'_y
\end{align*}
\]

These given points can now be used to define the two equations of a line for the diagonals. Therefore, the gradient of both diagonals is calculated using the y-values of the points (Formula [6.15], Formula [6.16]). Afterwards, the values can be inserted into the

Figure 6.3.: Preparations for Instability - intersection resolution.
equation of a line (Formula 6.12), that results in two equations of a line for the diagonals (Formula 6.17, Formula 6.18).

\[ m_a = a_y - a'_y \]  
\[ m_b = b_y - b'_y \]  
\[ f_a(x) = m_a x + a_x \cdot (-m_a) \]  
\[ f_b(x) = m_b x + b_x \cdot (-m_b) \]

Afterwards, the equations for the diagonals are set equal (Formula 6.19) and the resulting equation is solved to get the x-value of the intersection point \( i_x \) (see Formula 6.20). The y-value of the point is set to be the y-value of one of the surrounding points (\( a \) or \( b' \)). After finding this intersection point and inserting it into the data, the surrounding points, \( a \) and \( b' \) are removed. Now, the instability can be calculated as defined in Formula 6.5.

\[ f_a(x) = f_b(x) \]  
\[ m_a x + a_x \cdot (-m_a) = m_b x + b_x \cdot (-m_b) \]  
\[ m_a x - m_b x = b_x \cdot (-m_b) - a_x \cdot (-m_a) \]  
\[ i_x = \frac{b_x \cdot (-m_b) - a_x \cdot (-m_a)}{m_a - m_b} \]  

6.4. Evaluation Methodology

For the evaluation of multi-tier Chameleon, two different real world workload traces are used: German Wikipedia and BibSonomy. The German Wikipedia\textsuperscript{3} trace contains the page requests to all German Wikipedia sites during December 2013. Here, a subtrace of two days is used. The BibSonomy trace consists of HTTP request to the social bookmarking system BibSonomy \cite{BHJ10} during April 2017. Two days were chosen from this trace to benchmark the auto-scalers. So, both traces contain the progress of requests at two days. The two days traces are then compressed to get a measurement duration of six and a half hours for each trace.

In the following, for each evaluation, two figures are shown: One shows the result of multi-tier Chameleon and the other one the best competing auto-scaler. The figures of the remaining competing auto-scalers are depicted in the appendix. All shown figures have the same structure: a demand versus supply graph for each tier at the top and a request evaluation at the bottom. All graphs have the experiment duration of about 380 minutes at the x-axis. The y-axis shows for the demand supply plot the number of virtual machines, for the request evaluation the requests per second. The demand and supply graph shows the demand as black line and the supply as a red line. If the red line (supply) falls below the black line (demand) there are to less virtual machines provisioned. In case the red line exceeds the black line, to many virtual machines are instantiated. So, the optimal auto-scaler would result a red line matching the black line during the experiment. The request evaluation graph shows the sent requests as a black line, the requests processed conform to the SLO as green line and the requests that violate the SLOs as red line. If the green line matches the black line and the red line is zero during the experiment all requests have been served within the SLO. If the red line is not equal to zero and the green line drops below the black line, more SLO violations happen. An user-oriented auto-scaler tries to configure the application so that

\textsuperscript{3}German Wikipedia: \url{dumps.wikimedia.org/other/pagecounts-raw/2013}
all requests can be served within the SLO and therefore, the green line should match the black line.

In addition to the figures of multi-tier Chameleon and the best competing auto-scaler, a shortened table of the most important metrics is shown. It contains the aggregated provisioning accuracy, aggregated wrong provisioning time share, instability and auto-scaler deviation for each tier, as well as the mean and median response time, the SLO violation rate and the auto-scaler deviations summed up for all tiers. The table with all metrics can be found in the appendix.

### 6.5. Single-Tier Wikipedia

First, the results of the single-tier application driven with the Wikipedia trace are discussed. Figure 6.4 shows the results of multi-tier Chameleon. For this run, ARIMA is used as forecaster. The red line, that is shown in the graphic at the top, nearly matches the black line from the start to minute 40 and from 190 to 230 in the both increasing phases. In the areas where the demand stays almost stable, from minute 40 to 130 and 230 to 330, the red line is slightly above the black line. At the remaining areas, from minute 130 to 190 and 330 to the end, the red line converges to the black line at the end of these areas. The graphic of the requests at the bottom shows that the green line matches almost all the time the black line. The red line shows nearly no peak. This shows, that multi-tier Chameleon can ensure the serving of all requests in conformance to the SLOs. So, multi-tier Chameleon fits the demand in up-scaling areas quite well, over-provisions slightly in stable phases and converges to the demanded amount in downscaling phases. This can be explained by the aim of multi-tier Chameleon that is to ensure the SLO conform serving of all requests. Therefore, multi-tier Chameleon over-provisions the demand and accepts to have more virtual machines running than required. However, multi-tier Chameleon can anticipate upcoming increase of requests and scale up just in time.

![Figure 6.4: Demand, supply and requests evaluation of multi-tier Chameleon with Wikipedia trace on single-tier application.](image-url)

Figure 6.5 shows the best competing auto-scaler for this scenario: Adapt. As can be seen in the first 40 minutes of the measurement, the red line is below the black line. So, Adapt provisions to less instances in the first 40 minutes in the up-scaling area. This has the effect, that the red line in the requests plot is nearly at the black line and all requests
violate the SLO. After minute 40, the red line fits the slightly changing black line until minute 130. However, the up-scaling occurs too late and the requests that violate the SLO increase strongly. In the downscaling area from minute 130 to 190, the red line follows the black line with a small distance, and all requests can be served with SLO conformance due to the over-provisioned resources. Then, in the up-scaling area from minute 190 to 230 the resources are provisioned too late and the SLO violations increase. Afterwards, from minute 230, the red line follows the black line at the slightly changing area until 330 and finally, falls down as the black line with a small distance as observed earlier. In this time, most of the requests can be served within the SLOs. So, Adapt provisions as many resources as required in the stable fluctuating and down-scaling phases. However, it provisions resources too late in the increasing areas that results in SLO violated requests.

Finally, the results of multi-tier Chameleon and the competing auto-scalers are presented using the system- and user-oriented metrics in Table 6.3. As can be seen in the first row, multi-tier Chameleon and Adapt receive the same provisioning accuracy score. The other auto-scalers have a significantly higher accuracy value. The smaller the values of accuracy, time share, instability, and auto-scaler deviation is, the better the auto-scaler performs. However, multi-tier Chameleon has a slightly greater wrong provisioning time share than Adapt. The auto-scaler Reg has a lower wrong provisioning time share than multi-tier Chameleon, as well. The auto-scalers React and Hist have a greater value for this metric. When having a look at the instability metric, multi-tier Chameleon shows the smallest value and therefore, has less instable phases than all competing auto-scalers. The auto-scaler deviation of multi-tier Chameleon is the lowest of all evaluated auto-scalers, as well as the mean and median response time. The SLO violation rate is rounded to two decimal places. Multi-tier Chameleon has a violation rate of zero while the other auto-scalers have greater violation rates. Here, React has five percent SLO violations, and Hist has nine percent. Reg has by far the highest violation rate with eleven percent. All in all, multi-tier Chameleon shows the best auto-scaler deviation with a value of 0.86 in comparison to Adapt with a value of 0.90. In addition, multi-tier Chameleon has the lowest SLO violation rate of zero. The competing auto-scalers have violation rates from 5 to 33 percent.
Table 6.3.: Results of the single-tier evaluation with Wikipedia trace.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.0573</td>
<td>0.0573</td>
<td>0.0951</td>
<td>0.0965</td>
<td>0.1844</td>
<td>0.1835</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.3925</td>
<td>0.3266</td>
<td>0.4192</td>
<td>0.3919</td>
<td>0.4206</td>
<td>0.4325</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.1638</td>
<td>0.3247</td>
<td>0.2771</td>
<td>0.4422</td>
<td>0.6166</td>
<td>0.6816</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.86</td>
<td>0.90</td>
<td>0.92</td>
<td>0.96</td>
<td>1.02</td>
<td>1.04</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>236.29</td>
<td>415.09</td>
<td>318.63</td>
<td>653.92</td>
<td>287.77</td>
<td>511.83</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>209.00</td>
<td>226.00</td>
<td>218.00</td>
<td>244.00</td>
<td>221.00</td>
<td>211.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.00</td>
<td>0.12</td>
<td>0.05</td>
<td>0.33</td>
<td>0.09</td>
<td>0.44</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>0.86</td>
<td>0.90</td>
<td>0.92</td>
<td>0.96</td>
<td>1.02</td>
<td>1.04</td>
</tr>
</tbody>
</table>

6.6. Single-Tier BibSonomy

For the second evaluation, the BibSonomy trace on the single-tier application is used. Figure 6.6 shows the scaling behaviour of multi-tier Chameleon. Due to the high fluctuations over time, the BibSonomy trace is a more difficult trace than Wikipedia. These fluctuations are hard to forecast and therefore, the reactive mechanism of multi-tier Chameleon often has to correct the decisions of the proactive mechanism. Telescope is used as forecaster. However, at the top of the figure, the red line lies often within the range of the black line. Even though, there exist several areas where the supply curve falls below the demand curve and the SLO violations in the plot at the bottom increase. However, at the beginning, between minute 60 and 70, at minute 100, 170, 200, and 280, the supply curve is above the demand curve and there exist nearly no SLO violations at this time.

![Figure 6.6: Demand, supply and requests evaluation of multi-tier Chameleon with BibSonomy trace on single-tier application.](image)

For comparison, Adapt as the best of the competing auto-scalers for the single-tier BibSonomy evaluation, is depicted in Figure 6.7. Also, the red line lies within the range of the black line. However, the peaks and drops of the black line are not anticipated as they are in the multi-tier Chameleon experiment, and therefore, the SLO violated requests (red line) increase at the bottom of the figure. In comparison to the figure of
multi-tier Chameleon, there are less adaptations to fit the demand. So, Adapt smoothes
the demand curve and supplies medium number of virtual machines and ignores the
peaks. Though, there are some areas, where the red line (supply) lies above the black
line and the SLO conform requests (green line at the bottom plot) lies very near to the
black line. If the red line exceeds the black line at the bottom plot, the application sent
more responses than requests were sent. This can be explained due to waiting effects, if
the application cannot respond to all incoming requests. Theses incoming requests are
added to a waiting queue and processed when the application can serve them.

![Plot of multi-tier Chameleon and Adapt](image)

Figure 6.7.: Demand, supply and requests evaluation of Adapt with BibSonomy trace on
single-tier application.

After the discussion of the plots of multi-tier Chameleon and Adapt, the results using
the metrics for all auto-scalers are presented in Table 6.4. The provisioning accuracy
shows, that multi-tier Chameleon fits the demanded virtual machines best. The wrong
provisioning time share states, that multi-tier Chameleon has the smallest amount of
time in which the required amount of virtual machines are not equal to the supplied
amount. However, multi-tier Chameleon has a higher instability than Adapt and Reg.
This can be explained by the amount of adaptations multi-tier Chameleon processes to
fit the demand as good as possible. The auto-scaler deviation shows, that all auto-scalers
have approximately the same deviation to the theoretical optimal auto-scaler. The mean
and median response time of the application scaled by multi-tier Chameleon is higher
than the ones scaled with the competing auto-scalers. In addition, the SLO violation rate
of multi-tier Chameleon is higher than the one of React and Hist. The other competing
auto-scalers have comparable violation rates except for React whose violation rate is
about 11%.

In summary, multi-tier Chameleon tries to fit the demand curve as good as possible
and has therefore low provisioning accuracy and wrong provisioning time share values.
However, the instability is high due to the amount of adaptations multi-tier Chameleon
processes. The auto-scaler deviation of all auto-scalers is comparable. The violation rate
of multi-tier Chameleon is the third lowest but the both auto-scaler with lower violation
rates have a slightly higher auto-scaler deviation. The violation rate of Adapt and Reg
are comparable to the one of multi-tier Chameleon.
Table 6.4.: Results of the single-tier evaluation with BibSonomy trace.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.1337</td>
<td>0.1434</td>
<td>0.1732</td>
<td>0.1537</td>
<td>0.2147</td>
<td>0.2251</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.4121</td>
<td>0.4497</td>
<td>0.4375</td>
<td>0.4333</td>
<td>0.4425</td>
<td>0.4458</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.6025</td>
<td>0.5685</td>
<td>0.6077</td>
<td>0.5847</td>
<td>0.6145</td>
<td>0.6686</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>1.01</td>
<td>1.01</td>
<td>1.02</td>
<td>1.01</td>
<td>1.02</td>
<td>1.04</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>1,022.69</td>
<td>845.89</td>
<td>350.26</td>
<td>766.10</td>
<td>614.49</td>
<td>599.40</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>292.00</td>
<td>270.00</td>
<td>229.00</td>
<td>267.00</td>
<td>247.00</td>
<td>213.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.57</td>
<td>0.58</td>
<td>0.11</td>
<td>0.61</td>
<td>0.45</td>
<td>0.33</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>1.01</td>
<td>1.01</td>
<td>1.02</td>
<td>1.01</td>
<td>1.02</td>
<td>1.04</td>
</tr>
</tbody>
</table>


The next evaluation is run on the multi-tier application with the Wikipedia data trace. At each tier, the maximum number of virtual machines that can be provisioned is ten. Figure 6.8 shows the results of this scenario for multi-tier Chameleon with ARIMA as forecaster. As the top three plots of this figure show, all tiers are scaled up in time in the first upscaling area from start of the measurement until minute 40. Then, the demand remains nearly stable and multi-tier Chameleon over-provisions slightly at every tier. Later, in the downscaling phase the supply curve falls more slowly than the demand curve. At the second half of the experiment, the upscaling phase shows that multi-tier Chameleon scales up just in time. At the stable phase, a bit over-provisioning is measured and at the downscaling phase, the supply drops a bit later than the demand. This is caused by the conservative downscaling characteristics of multi-tier Chameleon. The requests plot shows that the upscaling behaviour results in little SLO violations, so the upscaling could be triggered a bit earlier, to remove those violations.

Figure 6.9 shows the results of the best competing auto-scaler Reg. Here, the problem stated in the introduction, bottleneck shifting, can be seen. At the first tier, the supply curve in the upscaling phase is a bit below the demand curve. At the business tier, the supply curve increases later than the one in the presentation tier. At the database tier, the supply increases latest. This is the expected behaviour of three single-tier auto-scalers when bottleneck shifting takes place. At minute 50 a drop of the supply curve in the presentation tier can be detected. This is passed through the business tier to the database tier, that are all downscaled at this point. A few minutes later, all virtual machines are provisioned to match the demand. Afterwards, in the stable phase, the red line matches the black line very closely with a few under provisioning phases. In the downscaling phase, the supply curve falls a bit later than the demand curve. Again, in the upscaling phase the upscaling delay between black and red curve becomes larger from the first to the last tier. In the stable phase, where the demand curve changes slightly, a drop of the red curve at the presentation tier and two drops at the business tier can be seen. And again, a few minutes later, an upscaling is triggered and the red line matches the black line. At the downscaling phase the red line falls a bit later than the black line. At the bottom of the figure the response time plot shows drops in the green line and peaks in the red line when the drops at the supply curve happen. For the rest of the experiment, the green line, that represents the SLO conformance, is very close to the black line.

Table 6.5 summarises the results of all competing auto-scalers based on the metrics. Here, a new column is inserted called Mixed. This is a mixture of the best single-tier
auto-scalers per tier based on the auto-scaler deviation metric. So, for the first tier Reg is chosen, for the second tier Adapt, and for the last tier Reg. The provisioning accuracy of multi-tier Chameleon at the first tier is the fourth best with 7.34%. Here, Adapt performs best with 3.5%. The wrong provisioning time share at the first tier states, that multi-tier Chameleon has the second worst value and Adapt the best. This can be explained due to the characteristic of multi-tier Chameleon to over-provision slightly, while the other auto-scaler try to match the demand accurately. However, the instability at the first tier shows, that multi-tier Chameleon is the third best. Here, React acts best. In summary, the auto-scaler deviation of the first tier states, that multi-tier Chameleon is the fourth best behind Reg, Hist and the mixed Approach. The metrics of the second tier show, that multi-tier Chameleon is third best in terms of provisioning accuracy behind Adapt and the mixed approach. The value of the wrong provisioning time share at tier two states that multi-tier Chameleon is on rank four and Adapt is best. In terms of instability at the second tier, multi-tier Chameleon is the third best behind React and Hist. In summary, the auto-scaler deviation of the second tier shows that multi-tier Chameleon is on the second place behind Adapt. At the third tier, multi-tier Chameleon acts best in terms of provisioning accuracy. When comparing the wrong provisioning time share of the third tier, multi-tier Chameleon performs best followed by Adapt. The instability metric at the third tier shows that multi-tier Chameleon acts third best behind Reg and the mixed approach. In terms of auto-scaler deviation at the third tier, multi-tier Chameleon acts best followed by the mixed approach. The mean and median response times of all auto-scalers are comparable. In terms of SLO violation
rate, multi-tier Chameleon acts second best with 3% violations behind React with 1% violations. The summed up auto-scaler deviation states, that multi-tier Chameleon acts second best and has the same value as the mixed approach. However, when comparing the SLO violation rates, the first and second auto-scaler in terms of auto-scaler deviation Reg and the mixed approach have a higher violation rate than multi-tier Chameleon.

In summary, multi-tier Chameleon wins the comparison to the competing auto-scalers when upscaling areas occur, because it provisions the resources just in time. In case of stable phases, the other auto-scalers can compete because they try to match the demand exactly while multi-tier Chameleon over-provisions slightly. However, the other auto-scalers show specific characteristics as seen with the drops in supply with Reg. In addition, the summed up auto-scaler deviation states that multi-tier Chameleon acts second best of all auto-scalers. When comparing the SLO violation rate of the best auto-scaler in terms of the deviation Reg with the rate of multi-tier Chameleon, the rate of multi-tier Chameleon is four times lower than the one of Reg. The reason why the comparison is that close is because the multi-tier evaluation is based on the synthetic application of which the exact resource demands are known. If the resource demands have to be estimated, errors of estimations are introduced. In addition, there is only one request class that is processed by the application. Real world applications have to handle multiple request classes with different resource demands.
### Table 6.5.: Results of the multi-tier evaluation with Wikipedia trace.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.0734</td>
<td>0.0351</td>
<td>0.2555</td>
<td>0.0574</td>
<td>0.1089</td>
<td>0.1417</td>
<td>0.0563</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.3009</td>
<td>0.1040</td>
<td>0.4792</td>
<td>0.1516</td>
<td>0.2350</td>
<td>0.2733</td>
<td>0.1525</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.0537</td>
<td>0.3516</td>
<td>0.0485</td>
<td>0.0634</td>
<td>0.0506</td>
<td>0.3120</td>
<td>0.0628</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracy</td>
<td>0.0518</td>
<td>0.0434</td>
<td>0.1333</td>
<td>0.0729</td>
<td>0.1249</td>
<td>0.1500</td>
<td>0.0504</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShare</td>
<td>0.3223</td>
<td>0.1975</td>
<td>0.4461</td>
<td>0.2767</td>
<td>0.3243</td>
<td>0.4095</td>
<td>0.2043</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.0928</td>
<td>0.1819</td>
<td>0.0700</td>
<td>0.1352</td>
<td>0.0713</td>
<td>0.2893</td>
<td>0.3939</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracy</td>
<td>0.0735</td>
<td>0.1005</td>
<td>0.3097</td>
<td>0.1128</td>
<td>0.1357</td>
<td>0.2522</td>
<td>0.1015</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShare</td>
<td>0.2038</td>
<td>0.2148</td>
<td>0.4788</td>
<td>0.2559</td>
<td>0.2549</td>
<td>0.3498</td>
<td>0.2248</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.0501</td>
<td>0.1348</td>
<td>0.3438</td>
<td>0.0443</td>
<td>0.1900</td>
<td>0.2604</td>
<td>0.0497</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>984.75</td>
<td>974.96</td>
<td>982.70</td>
<td>962.74</td>
<td>975.00</td>
<td>930.56</td>
<td>985.03</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.03</td>
<td>0.09</td>
<td>0.01</td>
<td>0.12</td>
<td>0.05</td>
<td>0.23</td>
<td>0.10</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.29</td>
<td>2.38</td>
<td>2.71</td>
<td>2.24</td>
<td>2.37</td>
<td>2.71</td>
<td>2.29</td>
</tr>
</tbody>
</table>

#### 6.7.1. Large Setup

In the following experiment, multi-tier Chameleon is evaluated against the best competing auto-scaler in terms of overall auto-scaler deviation, Reg. This experiment is used to evaluate whether the scaling behaviour of multi-tier Chameleon and Reg is the same for larger configurations. Therefore, the virtual machines provided per tier is changed: At the first tier there can be 15 virtual machines at maximum, at the second tier 25 virtual machines and at the third tier 10 virtual machines.

Figure 6.10 shows the scaling behaviour of multi-tier Chameleon. As can be seen at the top three plots of this figure, the scaling behaviour is similar to the one at the smaller setup. The supply curve increases a bit to late in case of the upscaling areas, but at the stable phases, a slight over-provisioning can be detected. At the downscaling areas the red line converges to the black line as seen in the smaller setup. As the red line is in most cases above or matches the black line, the SLO conform requests match the number of requests sent. However, at minute 60 a small drop of the supply curve can be detected and the violation rate increases. All in all, the scaling behaviour of multi-tier Chameleon on the large setup is comparable to the behaviour on the small setup.

Figure 6.11 shows the behaviour of Reg with the large setup of the multi-tier application. The bottleneck-shifting effect detected in the small setup can be seen more clearly now. The red line at the first tier is slightly below the black line. At the second tier, the distance between these lines is larger and at the last tier is largest. Then, at minute 40 a drop of the red line in the first tier is detected that is passed through the second and to the third tier. Now, the supply starts increasing at the first tier and again the bottleneck shifting effect takes place. The database tier needs most time until the red curve converges to the black one. At minute 140 another drop of the red line can be seen, now at the database tier. After the downscaling phase, the upscaling shows the bottleneck shifting again, and again a drop of the red line in the first tier can be seen. This is passed through all tiers and the same effect as seen at the first drop occurs. Again, the database tier needs most time to recover. While the red line at the database tier
is increasing, another drop at the first tier can be seen. At minute 340, one more drop of the red line is shown in the database tier. All of those drops and shifting effects that delay the upscaling at the later tiers can be seen in the last plot at the bottom: The red line representing the SLO violations spikes up at every drop and falls down slowly until the supply at the database tier converges to the demand.

In summary, when comparing the overall auto-scaler deviation and SLO violation rate of multi-tier Chameleon and Reg of the large to the small setup, multi-tier Chameleon shows comparable scaling behaviour. In contrast, the SLO violation rate of Reg is many times higher in the large setup. So, it can be said that multi-tier Chameleon scales its behaviour even in a larger setup, while the behaviour of Reg cannot be transferred to a larger setup.

The summary of the metrics of multi-tier Chameleon and Reg is shown in Table 6.6. At the first tier, multi-tier Chameleons provisioning accuracy and instability is lower than the one of Reg. Only the wrong provisioning time share of Reg is smaller than the one of multi-tier Chameleon. The auto-scaler deviation of multi-tier Chameleon is smaller than the one of Reg at the first tier. At the second tier, the values of all metrics for multi-tier Chameleon are less than the ones for Reg. The provisioning accuracy and wrong provisioning time share of the third tier for multi-tier Chameleon is less than the one of Reg. The instability and auto-scaler deviation of multi-tier Chameleon is larger for the third tier. The mean and response times of both are comparable. The overall auto-scaler deviation of multi-tier Chameleon is smaller than the one of Reg. In addition, the SLO violation rate of Reg is many times higher than the one of multi-tier Chameleon.
Figure 6.11.: Demand, supply and requests evaluation of Reg with Wikipedia trace on multi-tier application with large setup.

6.7.2. Reproducibility

In this section, the reproducibility of multi-tier Chameleon is evaluated. Therefore, the run of multi-tier Chameleon on the small setup of the multi-tier application with the Wikipedia trace is used. This trace contains requests measured over two days. Both days have almost the same request curve, so the scaling behaviour of multi-tier Chameleon should be comparable for both days. To be able to interpret the results of multi-tier Chameleon on both days, the discrepancy of both days is calculated using a baseline. This baseline is created with a fix resource supply of four, five and two virtual machines at presentation, business and database tier. These are the medium amounts that are provisioned during the measurement of multi-tier Chameleon. Table 6.7 contains the elasticity metrics of this baseline. As can be seen, the values at the first tier are higher than on the second except for instability, that remains the same. The overall auto-scaler deviation of day one is 1.16 times higher than the one of the second day. With this knowledge, the measurement of multi-tier Chameleon can now be evaluated.

The results of the comparison of first and second day for multi-tier Chameleon are summarised in Table 6.8. When comparing the provisioning accuracy for day one and two of all the tiers, it can be seen, that the value of the provisioning accuracy is a bit higher for day one, as was seen for the baseline as well. The wrong provisioning time share of day one compared to day two is higher for all tiers. The instability value for all tiers is the same for both days. The same has been observed for the baseline evaluation. The auto-scaler deviation is higher at the first day. The mean and median response times are comparable and the overall auto-scaler deviation is higher for the first tier.
Table 6.6.: Results of the multi-tier evaluation with Wikipedia trace with large setup.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.0831</td>
<td>0.1069</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.4279</td>
<td>0.3587</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.1170</td>
<td>0.2076</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.86</td>
<td>0.87</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracy</td>
<td>0.0637</td>
<td>0.1611</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShare</td>
<td>0.4385</td>
<td>0.4468</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.2641</td>
<td>0.3264</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.92</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracy</td>
<td>0.0831</td>
<td>0.2169</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShare</td>
<td>0.3569</td>
<td>0.4523</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.3926</td>
<td>0.1926</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.93</td>
<td>0.91</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>982.87</td>
<td>905.41</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>984.00</td>
<td>984.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.06</td>
<td>0.29</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.71</td>
<td>2.73</td>
</tr>
</tbody>
</table>

Table 6.7.: Results of the multi-tier evaluation for the baseline with Wikipedia trace day comparison.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Day 1</th>
<th>Day 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.1714</td>
<td>0.1081</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.4653</td>
<td>0.2460</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.0276</td>
<td>0.0276</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.85</td>
<td>0.73</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracy</td>
<td>0.1978</td>
<td>0.1047</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShare</td>
<td>0.4819</td>
<td>0.2562</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.0497</td>
<td>0.0497</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.87</td>
<td>0.75</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracy</td>
<td>0.1980</td>
<td>0.1153</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShare</td>
<td>0.4089</td>
<td>0.2215</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.0166</td>
<td>0.0166</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.83</td>
<td>0.71</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.54</td>
<td>2.19</td>
</tr>
</tbody>
</table>
This can be explained by the higher over-provisioning at the stable phase at the first day. In addition, the SLO violation rate is higher for the first day. The overall auto-scaler deviation of day one is 1.19 times higher than the one of the second day. This is in the range of the baseline approach, that was 1.16, and therefore it can be stated, that the scaling behaviour of multi-tier Chameleon is comparable for both days in terms of the elasticity metrics.

In addition, when looking at Figure 6.8, a similar scaling behaviour at first and second day can be observed. In the increasing phase of the demand curve, the supply curve is a bit late. Moreover, at the stable phase, over-provisioning occurs due to the characteristic of multi-tier Chameleon to not fully load all provisioned resources at both days. At the decreasing phase of the demand curve, the supply curve starts decreasing later than the demand line. However, the supply curve converges to the demand curve while increasing at both days.

Table 6.8.: Results of the multi-tier evaluation with Wikipedia trace day comparison.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Day 1</th>
<th>Day 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.0879</td>
<td>0.0311</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.3637</td>
<td>0.1261</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.0620</td>
<td>0.0620</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.81</td>
<td>0.66</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracy</td>
<td>0.0533</td>
<td>0.0262</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShare</td>
<td>0.3321</td>
<td>0.1627</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.0884</td>
<td>0.0884</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.81</td>
<td>0.71</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracy</td>
<td>0.0841</td>
<td>0.0330</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShare</td>
<td>0.2457</td>
<td>0.0857</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.0488</td>
<td>0.0488</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.74</td>
<td>0.61</td>
</tr>
<tr>
<td></td>
<td>MeanRespTime</td>
<td>984.76</td>
<td>984.73</td>
</tr>
<tr>
<td></td>
<td>MedianRespTime</td>
<td>984.00</td>
<td>984.00</td>
</tr>
<tr>
<td></td>
<td>SLOViolations</td>
<td>0.04</td>
<td>0.02</td>
</tr>
<tr>
<td></td>
<td>Deviation</td>
<td>2.36</td>
<td>1.97</td>
</tr>
</tbody>
</table>

In summary, the metrics have shown, that the scaling behaviour of multi-tier Chameleon is comparable for the first and second day. The difference of the overall auto-scaler deviation lies in the range of the ratio of the baseline. Additionally, the instability and SLO violation rate of both days are comparable. The scaling behaviour that can be seen in the plot, shows the same behaviour at both days, like the in time upscaling in increasing demand curve phases, over-provisioning in stable and decreasing over-provisioning in downscaling phases. So, the basic behaviour of multi-tier Chameleon is comparable at both days.
6.8. Multi-Tier BibSonomy

In the following, multi-tier Chameleon is evaluated using the small setup of the multi-tier application and the BibSonomy trace. Figure 6.12 shows the scaling behaviour of multi-tier Chameleon with the BibSonomy trace. As can be seen in all tiers, multi-tier Chameleon tries to match the demand as good as possible. It anticipates the peaks and drops and scales accordingly. However, during the first 50 minutes spikes in the supply curve can be seen. This happened because of improper forecast values. The over-provisioning is detected by the reactive scaling mechanism of multi-tier Chameleon and corrected instantly. In addition, at some points the supply curve falls below the black curve and under-provisioning happens. This can be seen in the plot at the bottom as well, where the red line has some spikes over time. However, most of the time, the green line matches the black line and all requests can be served in conformance to the SLOs.

![Figure 6.12.: Demand, supply and requests evaluation of multi-tier Chameleon with BibSonomy trace on multi-tier application.](image)

In this experiment, Reg is the auto-scaler performing best in terms of overall auto-scaler deviation. Its scaling behaviour is shown in Figure 6.13. As can be seen in the top three plots of the figure, Reg smoothes the demand curve and supplies medium virtual machines. The spikes and drops in the demand curve are not anticipated as they were from multi-tier Chameleon. This can be seen in the requests plot at the bottom as well, as the red line fluctuates up and down in accordance to the unanticipated peaks. In addition to this, several drops of the supply curve in the presentation and business tier can be seen between minute 250 and 300. This results in many requests that violate the SLOs and the red line at the bottom plot increases.
The evaluation based on metrics is summarised in Table 6.9. Here, multi-tier Chameleon and all competing auto-scalers are shown. When comparing the metrics of the first tier, multi-tier Chameleon has the lowest values in terms of provisioning accuracy and wrong provisioning time share. The instability of multi-tier Chameleon is greater than the one of the React, Hist and Reg that is caused by the aim to match the demand curve as good as possible. The auto-scaler deviation of the first tier shows that multi-tier Chameleon has the third best value behind Reg and Hist. At the second tier, multi-tier Chameleon has again the best values for provisioning accuracy and wrong provisioning time share. However, the instability of multi-tier Chameleon is larger than the others. The auto-scaler deviation at the second tier shows that multi-tier Chameleon performs third best behind Reg and Hist. At the third tier, Reg performs best for all metrics. However, multi-tier Chameleon is at the second place at provisioning accuracy and wrong provisioning time share. In terms of instability multi-tier Chameleon is on the fourth place. The auto-scaler deviation of the third tier states, that multi-tier Chameleon performs third best at this tier. The mean and median response times of all auto-scalers are comparable. The overall auto-scaler deviation shows that multi-tier Chameleon is the third best auto-scaler. When comparing it to the two better auto-scalers using the SLO violations, multi-tier Chameleon is has less violations than Reg but has a slightly worse violation rate than Hist.

In summary, multi-tier Chameleon tries to match the demand as close as possible and anticipates peaks and drops in the demand curve while Reg smoothes the demand curve and does not react to peaks. Therefore, the SLO violations of multi-tier Chameleon are smaller than the one of Reg. However, in terms of overall auto-scaler deviation Reg and
Hist perform better, but Reg has a worse SLO violation rate than multi-tier Chameleon.

Table 6.9.: Results of the multi-tier evaluation with BibSonomy trace.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.1207</td>
<td>0.1533</td>
<td>0.2934</td>
<td>0.1286</td>
<td>0.1605</td>
<td>0.2398</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.2866</td>
<td>0.3209</td>
<td>0.4523</td>
<td>0.2901</td>
<td>0.3340</td>
<td>0.3534</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.2772</td>
<td>0.4509</td>
<td>0.1904</td>
<td>0.2008</td>
<td>0.1790</td>
<td>0.2806</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.87</td>
<td>0.94</td>
<td>0.92</td>
<td>0.84</td>
<td>0.86</td>
<td>0.91</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracy</td>
<td>0.1129</td>
<td>0.1420</td>
<td>0.1899</td>
<td>0.1130</td>
<td>0.1578</td>
<td>0.1780</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShare</td>
<td>0.3246</td>
<td>0.3805</td>
<td>0.4232</td>
<td>0.3288</td>
<td>0.4022</td>
<td>0.4063</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.4570</td>
<td>0.4450</td>
<td>0.3364</td>
<td>0.3242</td>
<td>0.2967</td>
<td>0.4309</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.94</td>
<td>0.96</td>
<td>0.94</td>
<td>0.90</td>
<td>0.92</td>
<td>0.97</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracy</td>
<td>0.0895</td>
<td>0.1558</td>
<td>0.3452</td>
<td>0.0893</td>
<td>0.1073</td>
<td>0.1889</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShare</td>
<td>0.1890</td>
<td>0.2790</td>
<td>0.4673</td>
<td>0.1814</td>
<td>0.2288</td>
<td>0.3116</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.1978</td>
<td>0.2615</td>
<td>0.1195</td>
<td>0.1136</td>
<td>0.1145</td>
<td>0.2530</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.79</td>
<td>0.87</td>
<td>0.92</td>
<td>0.74</td>
<td>0.77</td>
<td>0.88</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>985.11</td>
<td>970.80</td>
<td>981.42</td>
<td>969.36</td>
<td>972.88</td>
<td>965.27</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>985.00</td>
<td>985.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>985.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.07</td>
<td>0.18</td>
<td>0.02</td>
<td>0.11</td>
<td>0.05</td>
<td>0.24</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.61</td>
<td>2.77</td>
<td>2.79</td>
<td>2.49</td>
<td>2.55</td>
<td>2.76</td>
</tr>
</tbody>
</table>

6.9. Side-Evaluation Forecasting

In this section, multi-tier Chameleon is evaluated in terms of different forecaster. Therefore, the multi-tier application and the BibSonomy trace is used. Multi-tier Chameleon is run twice on this setup: First, with Telescope as forecaster and then using TBATS.

Figure 6.14 shows the scaling behaviour of multi-tier Chameleon when using Telescope as forecaster. The top three plots of this figure show, that the red line is very close to the black line and most of the peaks and drops are recognised. However, at the beginning until minute 50, there are some spikes in the red line that can be explained by wrong forecasting results. So, Telescope forecasts to much load at the beginning and the proactive mechanism scales up. Then, the reactive mechanism detects the lower load and scales down. As can be seen at the bottom of the figure, the green line matches the black line most of the time and only a few small peaks of the red line occur. This means, that most of the incoming requests can be served within the SLO.

The second figure (Figure 6.15) shows the scaling behaviour of multi-tier Chameleon when TBATS is used as forecaster. The first three plots in the figure show, that there are less adaptations in this run. Most of the peaks and drops of the demand curve are not anticipated and the supply curve lies above the demand curve. There are no peaks in the supply curve as they were seen in the Telescope plot above. Though, there are a few drops in the supply curve where the supply falls below the demand. This results in a peak of the violations curve at the bottom. Even though, the SLO conform curve fits the sent curve at the bottom plot most of the time. So, multi-tier Chameleon scales the application so that most of the requests can be served within the SLO.

Table 6.10 summarises the results of both runs based on the metrics. At the first tier, multi-tier Chameleon with Telescope has better values for provisioning accuracy and
Figure 6.14.: Demand, supply and requests evaluation of multi-tier Chameleon with BibSonomy trace on multi-tier application with Telescope.

wrong provisioning time share than TBATS. The instability at the first tier is slightly worse. However, the auto-scaler deviation at the first tier is better for the Telescope run. At the second and third tier, the same results can be observed. The Telescope run is better in terms of provisioning accuracy, wrong provisioning time share and auto-scaler deviation but is worse in case of instability. This is caused by the accurate forecast of Telescope that fits most of the peaks and drops of the supply curve and therefore, more scaling actions have to be performed. The mean and median response time of both runs are comparable and the SLO violations are equal. However, the run with Telescope has a lower overall auto-scaler deviation than the run with TBATS.

In addition to the standard evaluation figures and table, Telescope and TBATS are evaluated using the BibSonomy trace directly. Therefore, the BibSonomy trace is given to Telescope and TBATS, while 80% of the trace is used for training and the remaining 20% are forecast at once.

Figure 6.16 shows the forecast results of this experiment. The x-axis shows the forecast horizon and the y-axis the number of HTTP requests that are forecast. The black line presents the 20% real values of the trace. The green line shows the forecast values from Telescope, the blue line shows the values from TBATS and the red line shows the values from ARIMA. Telescope anticipates the peaks and drops much better than TBATS that is smoothing them. In contrast, ARIMA does not deliver meaningful results in this experiment because the forecast horizon is too large.

In addition, Table 6.11 shows the MASE and MAPE values of the three forecaster as well
as the duration the method needs to forecast 20% of the trace. As can be seen, MASE
and MAPE values of Telescope are significantly smaller than the ones from TBATS and
ARIMA. Moreover, the forecast duration of Telescope is many times smaller than the
duration of TBATS and ARIMA. This shows, that Telescope is a better forecaster for
this time series than TBATS and ARIMA and therefore, is used for the evaluation of
multi-tier Chameleon on the BibSonomy trace.

In summary, the decision to use Telescope for the runs with the BibSonomy is meaning-
ful. The figures of the different scaling behaviour of multi-tier Chameleon with Telescope
Table 6.10.: Results of the multi-tier evaluation of Telescope and TBATS based on the Metrics.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Telescope</th>
<th>TBATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracy</td>
<td>0.1207</td>
<td>0.2253</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShare</td>
<td>0.2866</td>
<td>0.3916</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.2772</td>
<td>0.2166</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.87</td>
<td>0.90</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracy</td>
<td>0.1129</td>
<td>0.1987</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShare</td>
<td>0.3246</td>
<td>0.4204</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.4570</td>
<td>0.3508</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracy</td>
<td>0.0895</td>
<td>0.1799</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShare</td>
<td>0.1890</td>
<td>0.3382</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.1978</td>
<td>0.1662</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.79</td>
<td>0.86</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>985.11</td>
<td>985.16</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>985.00</td>
<td>985.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.61</td>
<td>2.71</td>
</tr>
</tbody>
</table>

Table 6.11.: Results of the standalone evaluation of Telescope, TBATS and ARIMA with BibSonomy.

<table>
<thead>
<tr>
<th>Method</th>
<th>MASE</th>
<th>MAPE</th>
<th>Duration (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Telescope</td>
<td>0.8923</td>
<td>24.0749</td>
<td>1.69</td>
</tr>
<tr>
<td>TBATS</td>
<td>1.1193</td>
<td>36.2431</td>
<td>44.92</td>
</tr>
<tr>
<td>ARIMA</td>
<td>1.8469</td>
<td>57.9300</td>
<td>81.29</td>
</tr>
</tbody>
</table>

and TBATS have shown that a very accurate scaling can be observed, which anticipates most of the peaks and drops of the demand curve and scales the application accordingly. Telescope matches the fluctuating pattern well, while TBATS smooths the curve and ignores the ups and downs. The SLO violation rate of both runs with multi-tier Chameleon were equal but the auto-scaler deviation of the Telescope run was better. In addition to that, the MASE and MAPE values of Telescope are significantly lower than the ones from TBATS and ARIMA. Moreover, the runtime of Telescope is multiple times lower than the runtime of TBATS and ARIMA. This is a very important factor for auto-scaling because if the forecast arrives late, the application cannot be scaled in time.

6.10. Multi-Tier Cost-Efficiency

This section evaluates the cost-efficiency component of multi-tier Chameleon. Therefore, the small setup of the multi-tier application with the BibSonomy trace is used. In
addition to the above mentioned metrics, in this scenario the virtual machines started and stopped by multi-tier Chameleon are measured and the total runtimes and charged runtimes are compared.

Figure 6.17 shows the results of the run of multi-tier Chameleon with the Amazon EC2 charging model. In this model, each virtual machine is charged on an hourly basis and all started hours are charged fully, no matter if the virtual machine stops earlier. The scaling behaviour is now a very different one compared to the standard run of multi-tier Chameleon on BibSonomy without cost-efficiency. The peaks and drops of the demand curve are now ignored and the supply curve does not follow the demand curve as close. The supply curve lies in most of the time above the demand curve. This is the desired behaviour of the cost-efficiency component: running virtual machines should not be stopped if the forecast values state that they will be required in the future. However, there are a few peaks at the beginning of the measurement of the supply curve that can be explained by wrong forecasting results. The same peaks have been seen earlier in the standard multi-tier Chameleon measurement with BibSonomy without cost-efficiency, where the reactive mechanism decides to scale the application down in these cases. In this measurement, the reactive mechanism decides to scale down the application as well, but the decision is ignored because a future proactive decision states a demand increase and the virtual machines remain running. Though, after these peaks in the supply are corrected at minute 100, the supply curve lies above the demand curve very closely. However, at short intervals, the supply curve lies below the demand curve and the SLO violation curve at the bottom of the figure increases. Though, at most of the time, the SLO conform curve matches the sent requests curve.

Figure 6.18 shows the scaling behaviour of multi-tier Chameleon with the BibSonomy trace and the Google Cloud Engine charging model. In this charging scheme, a virtual machine is charged for the first ten minutes on start fix and then charged per minute. The figure shows, that the scaling behaviour is comparable to the behaviour seen in the run with the Amazon charging scheme. The peaks of the supply curve at the beginning occur again. In addition, the supply curve lies above the demand curve and does not follow all fluctuations the demand curve shows. Instead, the supply curve remains stable at the maximum of the demand curve. The virtual machines are not stopped if the proactive decisions state that they will be required in the next ten minutes. The requests plot at the bottom of the curve shows, that at most of the time, the green line matches the black line and all requests can be served within the SLO. However, at short intervals the red line has small drops where the supply falls below the demand.

Table 6.12 summarises the results of three runs of multi-tier Chameleon: without cost-efficiency, with Amazon EC2 pricing model and with Google Cloud Platform (GCP) pricing model. When comparing the charged runtime with the Amazon EC2 charging model the charged hours have been reduced slightly. This small decrease of the charged hours can be explained by the increased real runtime in hours when using the cost-efficiency component. So, the cost savings are not as high as desired, though, the application can serve more requests within the SLOs and the violation rate decreases from 7% to 2.6%. When comparing the charged runtime calculated using the Google Cloud Platform model the charged fix ten minutes are reduced to a half when using the cost-efficiency component. The charged minutes and charged hours increased when using the cost-efficiency component due to the over-provisioning. When summing the amount of charged minutes and 10 minutes up, multi-tier Chameleon without cost-efficiency is charged for 264 ten minutes and with the cost-efficiency it is charged for 111 ten minutes.

---

4Amazon EC2 charging model: [aws.amazon.com/ec2/pricing/on-demand](aws.amazon.com/ec2/pricing/on-demand)
5Google Cloud Platform charging model: [cloud.google.com/compute/pricing#machinetype](cloud.google.com/compute/pricing#machinetype)
intervals. This is a decrease to the half. When taking the real runtime in hours into account, where multi-tier Chameleon with cost-efficiency has twice as much hours runtime, the decrease becomes important. In addition to this, the SLO violation rate has been decreased from 7% to 1.8% using the cost-efficiency component due to the higher runtime of virtual machines with the component and the over-provisioning effect when leaving virtual machines running if they are required in the future.

### Table 6.12.: Results of the multi-tier evaluation of the cost-efficiency component of multi-tier Chameleon.

<table>
<thead>
<tr>
<th>Charging Model</th>
<th>Charging Interval</th>
<th>Without Cost</th>
<th>EC2 Cost</th>
<th>GCP Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>EC2 Cost</td>
<td>hour</td>
<td>263</td>
<td>232</td>
<td>-</td>
</tr>
<tr>
<td>GCP Cost</td>
<td>10 minutes + minutes</td>
<td>264 + 7,530</td>
<td>169.5</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>hour</td>
<td>111 + 11,649</td>
<td>212.65</td>
<td></td>
</tr>
<tr>
<td>Real Runtime</td>
<td>hour</td>
<td>93.11</td>
<td>173.51</td>
<td>186.44</td>
</tr>
</tbody>
</table>

In conclusion, the cost-efficiency component of multi-tier Chameleon is able to reduce the charged runtime of virtual machines slightly. When comparing this with the real runtime of virtual machines, that is as double as high with the cost-efficiency component, the component is able to reduce the costs while providing doubled amount of resources. This also results in significant better SLO violation rate, that can be reduced from 7% without cost-efficiency to circa 2% with cost-efficiency.
6.11. Limitations

The evaluation on the multi-tier application has shown that multi-tier Chameleon performs second and third best in terms of overall auto-scaler deviation. When comparing the SLO violation rates, multi-tier Chameleon performs better than the competing auto-scalers. Possible reasons for this close results could be the synthetic multi-tier application. This application works exactly as desired that is done using semaphores. A real application would have more fluctuations in processing time. The design of the application allows to give the auto-scalers exact values of the resource demands and therefore, estimation errors introduced by LibReDE do not have an effect on the performance. In addition to this, there is only one request class that drives the application. Therefore, the auto-scalers have homogeneous workloads and do not have to anticipate fluctuations.

6.12. Discussion

In this section the research questions related to the goal Evaluation are discussed.

4. Evaluation

a) Goal: Show that a multi-tier auto-scaler can make better decisions with its overall view than one auto-scaler for every tier

Question: Does the overall view of a multi-tier auto-scaler enables it to make better decisions than one auto-scaler for every tier?

Metric: A set of detailed and aggregated elasticity metrics
The evaluations show that multi-tier Chameleon works in the two multi-tier evaluations second and third best in terms of auto-scaler deviation. However, the SLO violation rate is lower for multi-tier Chameleon. In addition, the bottleneck shifting effect seen for the single-tier auto-scalers could not be observed when using multi-tier Chameleon.

b) Goal: Show that multi-tier Chameleon’s scaling behaviour can be adapted to a large environments
Question: Does the scaling behaviour of multi-tier Chameleon can be transferred to a large scenarios?
Metric: A set of detailed and aggregated elasticity metrics

The evaluation on the large setup shows that the scaling behaviour of multi-tier Chameleon is comparable on large and small setup. In contrast, the single-tier auto-scaler Reg that is used for comparison, works worse on the large setup and the bottleneck shifting effect becomes more visible.

c) Goal: Show that multi-tier Chameleon’s scaling behaviour is reproducible
Question: Is the scaling behaviour of multi-tier Chameleon reproducible?
Metric: A set of detailed and aggregated elasticity metrics

The evaluation of multi-tier Chameleon on first and second day of the Wikipedia trace shows that both days have comparable characteristics and therefore, the reproducibility of multi-tier Chameleon can be assumed.

d) Goal: Show that multi-tier Chameleon can be extended to be cost-efficient.
Question: Is it possible to enable the auto-scaler to be cost-efficient?
Metric: Cost reduction in comparison to an auto-scaler without cost-efficient

The evaluation of the cost-efficiency component shows that the costs can be lowered by twelve percent for the Amazon EC2 charging strategy. For the charging model of the Google Cloud Platform, the costs of the fix charged amount are reduced to the half but the charged minutes increased by 50 percent.

e) Goal: Show that the costs can be lowered by a significant amount.
Question: By which amount can the costs be lowered?
Metric: Cost reduction ratio in comparison to an auto-scaler without cost-efficient

The evaluation of the cost-efficiency component shows that the costs can be lowered by a small amount. However, the total used hours of virtual machines can be increased and the costs can be used more efficiently. Moreover, the SLO violation rate is reduced to a minimum. This can be explained by the increased real runtime of the virtual machines.
7. Conclusion

Finally, this thesis is summarised in this chapter. Section 7.1 recaps the approach and evaluation of this thesis. Section 7.2 describes the work that could be done in the future.

7.1. Summary

This thesis focuses on scaling of multi-tier applications hosted in the cloud in an automatic manner. Therefore, the existing single-tier Chameleon is used as a basis. It is a hybrid auto-scaler that has a proactive and reactive mechanism to scale the application in time. It is based on measured values of the arrival rate and the CPU utilisation as well as estimated resource demands of virtual machines. This auto-scaler is developed further to support multi-tier applications. Therefore, the central controller is changed in multi-tier Chameleon to find scaling decisions for each tier independently, but with the knowledge of the other tiers. Hereby, the effects of bottleneck shifting and oscillations are minimised. In addition, the decision logic is changed and is now based on the theoretic utilisation using queueing theory and the measured, respectively forecast, arrival rates, and estimated resource demands.

A second goal of this thesis is to create a cost-aware auto-scaling mechanism. This is useful when hosting an application in the public cloud where the used resources are charged on instance minutes or hours. Therefore, a cost-efficiency component is added to multi-tier Chameleon. This component reviews all found scaling decisions and checks whether they are cost-efficient or not with the information of future decisions based on forecast values. Two charging models of two well-known cloud providers are used: Amazon EC2 and Google Cloud Platform. Amazon EC2 charges on an hourly basis and rounds every started hour to the next full hour and Google Cloud Platform charges the first ten minutes fix on startup and then switches to a charging minute by minute. Based on these charging models and by using decisions found for the future multi-tier Chameleon can evaluate whether it is more cost-efficient to remain a virtual machine running or to stop it. When the decision is to stop a virtual machine, multi-tier Chameleon can select the virtual machine that is, e.g., closest to a full hour for Amazon EC2 charging to stop it.

The multi-tier Chameleon is evaluated in this thesis using two applications: a single-tier and a multi-tier application. In addition two real world workload traces are used: German Wikipedia and the social bookmarking system BibSonomy. The evaluation on the single-tier application has shown that multi-tier Chameleon works best for the Wikipedia trace and has comparable results for the BibSonomy trace.
The evaluation on the multi-tier application with Wikipedia traces have shown that multi-tier Chameleon has no bottleneck shifting effect. In addition, multi-tier Chameleon works second best for this trace in terms of auto-scaler deviation. When comparing the SLO violations of multi-tier Chameleon to the violations of Reg, that is the best auto-scaler in terms of auto-scaler deviation, it can be seen, that multi-tier Chameleon has four times lower violation rate than Reg. The evaluation on the large setup with 15, 25 and 10 virtual machines on the tiers has shown, that the scaling behaviour of multi-tier Chameleon is transferable to the larger setup. Contrary, the scaling behaviour of Reg is not comparable to the one of the smaller setup. There have been more drops in the supply curve and therefore many times higher SLO violations in the large setup. Additionally, the bottleneck shifting effect becomes visible clearly in the large setup. For reproducibility evaluation of multi-tier Chameleon, the Wikipedia trace is used and the run is split into first and second day. Then, both days are compared using the elasticity metrics and by evaluating the plots. The results of the metrics have shown that the scaling behaviour of multi-tier Chameleon is comparable for the first and second day. In addition, the SLO violation rate is comparable. Moreover, the scaling behaviour shows similar characteristics for the first and second day.

The multi-tier evaluation using the BibSonomy traces have shown that multi-tier Chameleon tries to match the demand as close as possible and anticipates peaks and drops in the demand curve. The best competing auto-scaler in this setup, Reg, smooths the demand and does not anticipate peaks and drops. The SLO violation rate of multi-tier Chameleon is smaller than the one of reg. In terms of overall auto-scaler deviation, Hist and Reg perform better, but Reg has a worse violation rate than multi-tier Chameleon. For the BibSonomy evaluation the tool Telescope is used as forecaster. Therefore, a side-evaluation where Telescope is compared to TBATS is made. This evaluation has shown that the decision to use Telescope on BibSonomy trace was correct. Multi-tier Chameleon with Telescope has better results in terms of elasticity metrics than with TBATS. In addition, the standalone evaluation of Telescope and TBATS using BibSonomy has shown that the MASE and MAPE values of Telescope are much lower than the ones for TBATS. Moreover, the forecast duration of Telescope is many times lower than the one of TBATS.

To evaluate the cost-efficiency component of multi-tier Chameleon, the BibSonomy trace on the small setup of the multi-tier application is used. The results have shown that the cost-efficiency component is able to reduce the charged runtime of virtual machines slightly. In addition to that, multi-tier Chameleon is able to provide more effective virtual machine runtime that is as double as high without the cost-efficiency component. This also results in a reduced SLO violation rate that is reduced from 7% to circa 2%.

7.2. Future Work

In the future, several additional features and further evaluations could be made. First, an extension to support vertical scaling could be focussed. This could be combined with horizontal scaling where a decision logic can evaluate which scaling direction is more efficient. Therefore, a cost function needs to be added. Second, besides the existing cost-efficiency component, another component responsible for optimising energy consumption is possible. This component should monitor the energy consumption and execute voltage scaling, e.g., by tuning the CPU frequency, or stop virtual machines in exchange for a higher CPU frequency. The overall target of this component should be to minimise the energy consumption of the overall application. Third, the functionality of measuring the mean startup time of a virtual machine for timing the cost-efficiency handler could be used in the standard multi-tier Chameleon, as well. It could be used to time proactive
decisions so that the new provisioned virtual machines of this decision are already up when the decisions target time happens.

In addition to further components or functionalities, the evaluation of multi-tier Chameleon should be extended. First, the measurements on the large setup with 15, 25 and 10 virtual machines at the tiers could be used to evaluate the remaining competing auto-scalers. Additionally, the scaling behaviour of multi-tier Chameleon could be evaluated on the large setup using the BibSonomy trace. Furthermore, this setup could be used to evaluate the cost-efficiency at larger environments. Second, the multi-tier Chameleon should be evaluated using a more realistic multi-tier application with real functionalities and computing durations. Third, multi-tier Chameleon could be evaluated on an environment with hundreds of virtual machines by running a simulation of the cloud and scaling the simulated application.
# List of Figures

2.2. Example model of a multi-tier application ........................................ 12  
2.3. Methods for performance analysis of distributed systems ....................... 13  
2.4. Little’s Law .............................................................. 13  
2.5. $M/M/n - \infty$ queueing system ............................................ 14  

4.1. Architecture of the single-tier version of Chameleon ............................ 29  
4.2. Architecture of multi-tier Chameleon .............................................. 30  
4.3. State diagram of multi-tier Chameleon ............................................ 31  
4.4. Workflow of Telescope .......................................................... 34  
4.5. Combination of two proactive decisions ......................................... 39  
4.6. Future look combination ........................................................... 43  
4.7. Example of multi-tier Chameleon decision logic .................................. 48  

5.1. Overview of the class relations of multi-tier Chameleon ......................... 51  
5.2. UML diagram of the monitoring component ....................................... 52  
5.3. UML diagram of the logging component ......................................... 53  
5.4. UML diagram of the event management component ............................. 53  
5.5. UML diagram of the cost efficiency component .................................. 54  

6.1. Sequence diagram of the multi-tier application ................................... 57  
6.2. Preparations for Instability ....................................................... 61  
6.3. Preparations for Instability - intersection resolution .......................... 62  
6.4. Evaluation of Chameleon with Wikipedia on single-tier application ........ 64  
6.5. Evaluation of Adapt with Wikipedia on single-tier application ............. 65  
6.6. Evaluation of Chameleon with BibSonomy on single-tier application ...... 66  
6.7. Evaluation of Adapt with BibSonomy on single-tier application ............ 67  
6.9. Evaluation of Reg with Wikipedia on multi-tier application ................. 70  
6.10. Evaluation of Chameleon with Wikipedia on multi-tier application with large setup ......................................................... 72  
6.11. Evaluation of Reg with Wikipedia on multi-tier application with large setup .............. ......................................................... 73  
6.12. Evaluation of Chameleon with BibSonomy on multi-tier application ...... 76  
6.13. Evaluation of Reg with BibSonomy on multi-tier application ............... 77  
6.15. Evaluation of Chameleon with BibSonomy on multi-tier application with TBATS ................................................................. 80  
6.16. Forecasting results of Telescope, TBATS and ARIMA with BibSonomy ... 80  
6.17. Evaluation of Chameleon with BibSonomy on multi-tier application with the Amazon charging strategy ........................................... 83
6.18. Evaluation of Chameleon with BibSonomy on multi-tier application with
the Google charging strategy. .................................................. 84

A.1. Evaluation of React with Wikipedia on single-tier application. ........ 102
A.2. Evaluation of Reg with Wikipedia on single-tier application. ........ 102
A.3. Evaluation of Hist with Wikipedia on single-tier application. ........ 103
A.4. Evaluation of ConPaaS with Wikipedia on single-tier application. .. 103
B.5. Evaluation of React with BibSonomy on single-tier application. ....... 104
B.6. Evaluation of Reg with BibSonomy on single-tier application. ......... 105
B.7. Evaluation of Hist with BibSonomy on single-tier application. ......... 105
B.8. Evaluation of ConPaaS with BibSonomy on single-tier application. ... 106
C.13 Evaluation of Mixed Auto-Scalers with Wikipedia on multi-tier applica
tion. ................. 112
D.15.Evaluation of React with BibSonomy on multi-tier application. ...... 117
D.17.Evaluation of ConPaaS with BibSonomy on multi-tier application. .. 119
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1.</td>
<td>Overview of Related Work</td>
<td>25</td>
</tr>
<tr>
<td>4.1.</td>
<td>Configuration of the Tiers of Multi-Tier Chameleon</td>
<td>48</td>
</tr>
<tr>
<td>4.2.</td>
<td>Configuration Parameters of Multi-Tier Chameleon</td>
<td>49</td>
</tr>
<tr>
<td>6.1.</td>
<td>Specification of the physical servers</td>
<td>55</td>
</tr>
<tr>
<td>6.2.</td>
<td>Specification of the virtual machines</td>
<td>56</td>
</tr>
<tr>
<td>6.3.</td>
<td>Results of the single-tier evaluation with Wikipedia trace</td>
<td>66</td>
</tr>
<tr>
<td>6.4.</td>
<td>Results of the single-tier evaluation with BibSonomy trace</td>
<td>68</td>
</tr>
<tr>
<td>6.5.</td>
<td>Results of the multi-tier evaluation with Wikipedia trace</td>
<td>71</td>
</tr>
<tr>
<td>6.6.</td>
<td>Results of the multi-tier evaluation with Wikipedia trace with large setup</td>
<td>74</td>
</tr>
<tr>
<td>6.7.</td>
<td>Results of the multi-tier evaluation for the baseline with Wikipedia trace day comparison</td>
<td>74</td>
</tr>
<tr>
<td>6.8.</td>
<td>Results of the multi-tier evaluation with Wikipedia trace day comparison</td>
<td>75</td>
</tr>
<tr>
<td>6.9.</td>
<td>Results of the multi-tier evaluation with BibSonomy trace</td>
<td>78</td>
</tr>
<tr>
<td>6.10.</td>
<td>Results of the multi-tier evaluation of Telescope and TBATS based on the Metrics</td>
<td>81</td>
</tr>
<tr>
<td>6.11.</td>
<td>Results of the standalone evaluation of Telescope, TBATS and ARIMA with BibSonomy</td>
<td>81</td>
</tr>
<tr>
<td>6.12.</td>
<td>Results of the multi-tier evaluation of the cost-efficiency component of multi-tier Chameleon</td>
<td>83</td>
</tr>
<tr>
<td>A.1.</td>
<td>All results of the single-tier evaluation with Wikipedia</td>
<td>101</td>
</tr>
<tr>
<td>B.2.</td>
<td>All results of the single-tier evaluation with BibSonomy</td>
<td>104</td>
</tr>
<tr>
<td>C.3.</td>
<td>All results of the multi-tier evaluation with Wikipedia</td>
<td>107</td>
</tr>
<tr>
<td>C.4.</td>
<td>All results of the multi-tier evaluation with Large Setup on the Wikipedia</td>
<td>113</td>
</tr>
<tr>
<td>C.5.</td>
<td>All of the multi-tier evaluation with Wikipedia Day Comparison</td>
<td>114</td>
</tr>
<tr>
<td>D.6.</td>
<td>All results of the multi-tier evaluation with BibSonomy</td>
<td>115</td>
</tr>
<tr>
<td>D.7.</td>
<td>All results of the multi-tier evaluation of Telescope and TBATS</td>
<td>120</td>
</tr>
</tbody>
</table>
List of Algorithms

1. Initialisation State ................................................. 32
2. Forecasting State .................................................. 33
3. Resource Demand Estimation State ................................ 35
4. Application Monitoring State ...................................... 36
5. Tier Monitoring State .............................................. 36
6. Proactive Decision State ........................................... 38
7. Proactive Decision State (combineDecisions) ...................... 40
8. Reactive Decision State ............................................. 42
9. Cost-Efficiency State ................................................ 44
10. Cost-Efficiency State (Decision Handling) ....................... 45
11. Scaling State ....................................................... 47
Bibliography


98


Appendix


Table A.1.: All results of the single-tier evaluation with Wikipedia.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0042</td>
<td>0.0256</td>
<td>0.0145</td>
<td>0.1587</td>
<td>0.0361</td>
<td>0.1792</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td>0.1104</td>
<td>0.0890</td>
<td>0.1757</td>
<td>0.0343</td>
<td>0.3328</td>
<td>0.1877</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.0370</td>
<td>0.1646</td>
<td>0.0780</td>
<td>0.4912</td>
<td>0.1696</td>
<td>0.5710</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.7479</td>
<td>0.4886</td>
<td>0.7603</td>
<td>0.2925</td>
<td>0.6715</td>
<td>0.2941</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.1638</td>
<td>0.3247</td>
<td>0.2771</td>
<td>0.4422</td>
<td>0.6166</td>
<td>0.6816</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>14.75</td>
<td>13.76</td>
<td>14.87</td>
<td>11.90</td>
<td>15.73</td>
<td>12.16</td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>81</td>
<td>83</td>
<td>71</td>
<td>202</td>
<td>91</td>
<td>323</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.86</td>
<td>0.90</td>
<td>0.92</td>
<td>0.96</td>
<td>1.02</td>
<td>1.04</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>236.29</td>
<td>415.09</td>
<td>318.63</td>
<td>653.92</td>
<td>287.77</td>
<td>511.83</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>209.00</td>
<td>226.00</td>
<td>218.00</td>
<td>244.00</td>
<td>221.00</td>
<td>211.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.00</td>
<td>0.12</td>
<td>0.05</td>
<td>0.33</td>
<td>0.09</td>
<td>0.44</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>0.86</td>
<td>0.90</td>
<td>0.92</td>
<td>0.96</td>
<td>1.02</td>
<td>1.04</td>
</tr>
</tbody>
</table>
Figure A.1.: Demand, supply and requests evaluation of React with Wikipedia trace on single-tier application.

Figure A.2.: Demand, supply and requests evaluation of Reg with Wikipedia trace on single-tier application.
Figure A.3.: Demand, supply and requests evaluation of Hist with Wikipedia trace on single-tier application.

Figure A.4.: Demand, supply and requests evaluation of ConPaaS with Wikipedia trace on single-tier application.
B. Evaluation Single-Tier BibSonomy

Table B.2.: All results of the single-tier evaluation with BibSonomy.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0718</td>
<td>0.1369</td>
<td>0.0414</td>
<td>0.1969</td>
<td>0.0392</td>
<td>0.1417</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td>0.1955</td>
<td>0.1498</td>
<td>0.3049</td>
<td>0.1105</td>
<td>0.3903</td>
<td>0.3085</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.3168</td>
<td>0.4507</td>
<td>0.2041</td>
<td>0.5399</td>
<td>0.1946</td>
<td>0.3922</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.5074</td>
<td>0.4488</td>
<td>0.6710</td>
<td>0.3267</td>
<td>0.6905</td>
<td>0.4993</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.6025</td>
<td>0.5685</td>
<td>0.6077</td>
<td>0.5847</td>
<td>0.6145</td>
<td>0.6686</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>10.46</td>
<td>9.98</td>
<td>11.52</td>
<td>8.89</td>
<td>11.94</td>
<td>11.06</td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>354</td>
<td>143</td>
<td>125</td>
<td>231</td>
<td>64</td>
<td>349</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>1.01</td>
<td>1.01</td>
<td>1.02</td>
<td>1.01</td>
<td>1.02</td>
<td>1.04</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>1,022.69</td>
<td>845.89</td>
<td>350.26</td>
<td>766.10</td>
<td>614.49</td>
<td>599.40</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>292.00</td>
<td>270.00</td>
<td>229.00</td>
<td>267.00</td>
<td>247.00</td>
<td>213.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.57</td>
<td>0.58</td>
<td>0.11</td>
<td>0.61</td>
<td>0.45</td>
<td>0.33</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>1.01</td>
<td>1.01</td>
<td>1.02</td>
<td>1.01</td>
<td>1.02</td>
<td>1.04</td>
</tr>
</tbody>
</table>

Figure B.5.: Demand, supply and requests evaluation of React with BibSonomy trace on single-tier application.
Figure B.6.: Demand, supply and requests evaluation of Reg with BibSonomy trace on single-tier application.

Figure B.7.: Demand, supply and requests evaluation of Hist with BibSonomy trace on single-tier application.
Figure B.8.: Demand, supply and requests evaluation of ConPaaS with BibSonomy trace on single-tier application.
## C. Evaluation Multi-Tier Wikipedia

Table C.3.: All results of the multi-tier evaluation with Wikipedia.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
<th>Mixed</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0035</td>
<td>0.0316</td>
<td><strong>0.0007</strong></td>
<td>0.0300</td>
<td>0.0229</td>
<td>0.1385</td>
<td>0.0340</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td>0.1432</td>
<td>0.0385</td>
<td>0.5104</td>
<td>0.0849</td>
<td>0.1949</td>
<td>0.1450</td>
<td><strong>0.0787</strong></td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.0174</td>
<td>0.1012</td>
<td><strong>0.0034</strong></td>
<td>0.0816</td>
<td>0.0964</td>
<td>0.3406</td>
<td>0.0991</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.5844</td>
<td>0.1068</td>
<td><strong>0.3485</strong></td>
<td>0.0634</td>
<td>0.0506</td>
<td>0.3120</td>
<td>0.0628</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.0537</td>
<td>0.3516</td>
<td><strong>0.0485</strong></td>
<td>0.471</td>
<td>0.18</td>
<td>0.24</td>
<td>0.18</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>4.79</td>
<td>4.17</td>
<td>6.03</td>
<td>4.30</td>
<td>4.18</td>
<td>4.27</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>32</td>
<td>57</td>
<td>24</td>
<td>40</td>
<td>37</td>
<td>17.6</td>
<td>36</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.77</td>
<td>0.82</td>
<td>0.88</td>
<td><strong>0.68</strong></td>
<td>0.74</td>
<td>0.88</td>
<td><strong>0.68</strong></td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyU</td>
<td>0.0046</td>
<td>0.0506</td>
<td><strong>0.0014</strong></td>
<td>0.0757</td>
<td>0.0337</td>
<td>0.2166</td>
<td>0.0612</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyO</td>
<td>0.0990</td>
<td><strong>0.0362</strong></td>
<td>0.2652</td>
<td>0.0701</td>
<td>0.2161</td>
<td>0.1013</td>
<td>0.0395</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareU</td>
<td>0.0317</td>
<td>0.2045</td>
<td><strong>0.0089</strong></td>
<td>0.3074</td>
<td>0.1687</td>
<td>0.5969</td>
<td>0.2252</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareO</td>
<td>0.6129</td>
<td>0.1905</td>
<td>0.8834</td>
<td>0.2459</td>
<td>0.4798</td>
<td>0.2222</td>
<td><strong>0.1833</strong></td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.0928</td>
<td>0.1819</td>
<td><strong>0.0700</strong></td>
<td>0.1352</td>
<td>0.0713</td>
<td>0.2893</td>
<td>0.3939</td>
</tr>
<tr>
<td>2</td>
<td>AverageAmount</td>
<td>7.47</td>
<td>6.66</td>
<td>8.24</td>
<td>6.57</td>
<td>7.66</td>
<td>5.85</td>
<td>6.66</td>
</tr>
<tr>
<td>2</td>
<td>AmountAdapt</td>
<td>43</td>
<td>97</td>
<td>23</td>
<td>73</td>
<td>35</td>
<td>190</td>
<td>127</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.80</td>
<td><strong>0.79</strong></td>
<td>0.85</td>
<td>0.80</td>
<td>0.80</td>
<td>0.92</td>
<td>0.88</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyU</td>
<td>0.0089</td>
<td>0.1201</td>
<td><strong>0.0007</strong></td>
<td>0.0913</td>
<td>0.0633</td>
<td>0.2265</td>
<td>0.0941</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyO</td>
<td>0.1380</td>
<td><strong>0.0808</strong></td>
<td>0.6188</td>
<td>0.1343</td>
<td>0.2081</td>
<td>0.2779</td>
<td>0.1089</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareU</td>
<td>0.0238</td>
<td>0.3255</td>
<td><strong>0.0013</strong></td>
<td>0.3145</td>
<td>0.1889</td>
<td>0.4761</td>
<td>0.2926</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareO</td>
<td>0.3837</td>
<td><strong>0.1042</strong></td>
<td>0.9562</td>
<td>0.1973</td>
<td>0.3209</td>
<td>0.2236</td>
<td>0.1570</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.0501</td>
<td>0.1348</td>
<td>0.3438</td>
<td><strong>0.0443</strong></td>
<td>0.1990</td>
<td>0.2604</td>
<td>0.0497</td>
</tr>
<tr>
<td>3</td>
<td>AverageAmount</td>
<td>3.40</td>
<td>2.73</td>
<td>4.49</td>
<td>2.92</td>
<td>3.24</td>
<td>3.27</td>
<td>2.89</td>
</tr>
<tr>
<td>3</td>
<td>AmountAdapt</td>
<td>24</td>
<td>120</td>
<td>19</td>
<td>19</td>
<td>30</td>
<td>134</td>
<td>23</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td><strong>0.71</strong></td>
<td>0.77</td>
<td>0.98</td>
<td>0.75</td>
<td>0.83</td>
<td>0.91</td>
<td>0.73</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>984.75</td>
<td>974.96</td>
<td>982.70</td>
<td>962.74</td>
<td>975.00</td>
<td>930.56</td>
<td>985.03</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.03</td>
<td>0.09</td>
<td><strong>0.01</strong></td>
<td>0.12</td>
<td>0.05</td>
<td>0.23</td>
<td>0.10</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.29</td>
<td>2.38</td>
<td>2.71</td>
<td><strong>2.24</strong></td>
<td>2.37</td>
<td>2.71</td>
<td>2.29</td>
</tr>
</tbody>
</table>
Figure C.9.: Demand, supply and requests evaluation of Adapt with Wikipedia trace on multi-tier application.
Figure C.10.: Demand, supply and requests evaluation of React with Wikipedia trace on multi-tier application.
Figure C.11.: Demand, supply and requests evaluation of Hist with Wikipedia trace on multi-tier application.
Figure C.12.: Demand, supply and requests evaluation of ConPaaS with Wikipedia trace on multi-tier application.
Figure C.13.: Demand, supply and requests evaluation of Mixed Auto-Scalers with Wikipedia trace on multi-tier application.
C.1. Large Setup

Table C.4.: All results of the multi-tier evaluation with Large Setup on the Wikipedia.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0059</td>
<td>0.1358</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td>0.1603</td>
<td>0.0780</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.0334</td>
<td>0.3850</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.8223</td>
<td>0.3324</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.1170</td>
<td>0.2076</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>11.30</td>
<td>8.78</td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>59</td>
<td>136</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.86</td>
<td>0.87</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyU</td>
<td>0.0065</td>
<td>0.2077</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyO</td>
<td>0.1209</td>
<td>0.1146</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareU</td>
<td>0.0510</td>
<td>0.5659</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareO</td>
<td>0.8260</td>
<td>0.3277</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.2641</td>
<td>0.3264</td>
</tr>
<tr>
<td>2</td>
<td>AverageAmount</td>
<td>18.36</td>
<td>13.66</td>
</tr>
<tr>
<td>2</td>
<td>AmountAdapt</td>
<td>106</td>
<td>251</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.92</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyU</td>
<td>0.0046</td>
<td>0.3058</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyO</td>
<td>0.1615</td>
<td>0.1280</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareU</td>
<td>0.0278</td>
<td>0.6909</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareO</td>
<td>0.6861</td>
<td>0.2136</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.3926</td>
<td>0.1926</td>
</tr>
<tr>
<td>3</td>
<td>AverageAmount</td>
<td>7.94</td>
<td>4.99</td>
</tr>
<tr>
<td>3</td>
<td>AmountAdapt</td>
<td>29</td>
<td>118</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.93</td>
<td>0.91</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>982.87</td>
<td>905.41</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>984.00</td>
<td>984.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.06</td>
<td>0.29</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.71</td>
<td>2.73</td>
</tr>
</tbody>
</table>
C.2. Reproducibility

Table C.5.: All of the multi-tier evaluation with Wikipedia Day Comparison.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Day 1</th>
<th>Day 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0068</td>
<td>0.0003</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td>0.1690</td>
<td>0.0620</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.0335</td>
<td>0.0014</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.6939</td>
<td>0.2508</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.0620</td>
<td>0.0620</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>5.15</td>
<td>4.46</td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>20</td>
<td>12</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.81</td>
<td>0.66</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyU</td>
<td>0.0069</td>
<td>0.0013</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyO</td>
<td>0.0997</td>
<td>0.0511</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareU</td>
<td>0.0450</td>
<td>0.0100</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareO</td>
<td>0.6191</td>
<td>0.3153</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.0884</td>
<td>0.0884</td>
</tr>
<tr>
<td>2</td>
<td>AverageAmount</td>
<td>7.96</td>
<td>7.02</td>
</tr>
<tr>
<td>2</td>
<td>AmountAdapt</td>
<td>18</td>
<td>25</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.81</td>
<td>0.71</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyU</td>
<td>0.0174</td>
<td>0.0005</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyO</td>
<td>0.1507</td>
<td>0.0656</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareU</td>
<td>0.0463</td>
<td>0.0015</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareO</td>
<td>0.4450</td>
<td>0.1698</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.0488</td>
<td>0.0488</td>
</tr>
<tr>
<td>3</td>
<td>AverageAmount</td>
<td>3.69</td>
<td>3.14</td>
</tr>
<tr>
<td>3</td>
<td>AmountAdapt</td>
<td>14</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.74</td>
<td>0.61</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>984.76</td>
<td>984.73</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>984.00</td>
<td>984.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.04</td>
<td>0.02</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.36</td>
<td>1.97</td>
</tr>
</tbody>
</table>
### D. Evaluation Multi-Tier BibSonomy

Table D.6.: All results of the multi-tier evaluation with BibSonomy.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Chameleon</th>
<th>Adapt</th>
<th>React</th>
<th>Reg</th>
<th>Hist</th>
<th>ConPaaS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0353</td>
<td>0.0960</td>
<td><strong>0.0034</strong></td>
<td>0.0462</td>
<td>0.0273</td>
<td>0.1324</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td><strong>0.2061</strong></td>
<td>0.2105</td>
<td>0.5833</td>
<td>0.2109</td>
<td>0.2938</td>
<td>0.3472</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.1114</td>
<td>0.2923</td>
<td><strong>0.0109</strong></td>
<td>0.1525</td>
<td>0.1067</td>
<td>0.3469</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.4618</td>
<td><strong>0.3494</strong></td>
<td>0.8936</td>
<td>0.4277</td>
<td>0.5612</td>
<td>0.3599</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.2772</td>
<td>0.4509</td>
<td>0.1904</td>
<td>0.2008</td>
<td><strong>0.1790</strong></td>
<td>0.2806</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>3.72</td>
<td>3.42</td>
<td>4.75</td>
<td>3.63</td>
<td>3.88</td>
<td>3.78</td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>148</td>
<td>199</td>
<td>45</td>
<td>47</td>
<td>29</td>
<td>128</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.87</td>
<td>0.94</td>
<td>0.92</td>
<td><strong>0.84</strong></td>
<td>0.86</td>
<td>0.91</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyU</td>
<td>0.0489</td>
<td>0.1286</td>
<td><strong>0.0128</strong></td>
<td>0.0975</td>
<td>0.0389</td>
<td>0.2126</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyO</td>
<td>0.1770</td>
<td>0.1554</td>
<td>0.3670</td>
<td><strong>0.1284</strong></td>
<td>0.2767</td>
<td>0.1434</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareU</td>
<td>0.1870</td>
<td>0.4540</td>
<td><strong>0.0769</strong></td>
<td>0.3014</td>
<td>0.2038</td>
<td>0.5651</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareO</td>
<td>0.4622</td>
<td>0.3069</td>
<td>0.7696</td>
<td>0.3562</td>
<td>0.6007</td>
<td><strong>0.2474</strong></td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.4570</td>
<td>0.4450</td>
<td>0.3364</td>
<td>0.3242</td>
<td><strong>0.2967</strong></td>
<td>0.4309</td>
</tr>
<tr>
<td>2</td>
<td>AverageAmount</td>
<td>5.77</td>
<td>5.00</td>
<td>6.65</td>
<td>5.22</td>
<td>6.09</td>
<td>4.74</td>
</tr>
<tr>
<td>2</td>
<td>AmountAdapt</td>
<td>204</td>
<td>181</td>
<td>106</td>
<td>86</td>
<td>34</td>
<td>134</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.94</td>
<td>0.96</td>
<td>0.94</td>
<td><strong>0.90</strong></td>
<td>0.92</td>
<td>0.97</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyU</td>
<td>0.0545</td>
<td>0.1663</td>
<td><strong>0.0000</strong></td>
<td>0.0439</td>
<td>0.0614</td>
<td>0.2303</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyO</td>
<td><strong>0.1245</strong></td>
<td>0.1452</td>
<td>0.6903</td>
<td>0.1348</td>
<td>0.1532</td>
<td>0.1474</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareU</td>
<td>0.1401</td>
<td>0.4016</td>
<td><strong>0.0000</strong></td>
<td>0.1370</td>
<td>0.1904</td>
<td>0.4938</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareO</td>
<td>0.2379</td>
<td>0.1563</td>
<td>0.9346</td>
<td>0.2257</td>
<td>0.2672</td>
<td><strong>0.1293</strong></td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.1978</td>
<td>0.2615</td>
<td>0.1195</td>
<td><strong>0.1136</strong></td>
<td>0.1145</td>
<td>0.2530</td>
</tr>
<tr>
<td>3</td>
<td>AverageAmount</td>
<td>2.58</td>
<td>2.19</td>
<td>3.81</td>
<td>2.56</td>
<td>2.56</td>
<td>2.27</td>
</tr>
<tr>
<td>3</td>
<td>AmountAdapt</td>
<td>105</td>
<td>136</td>
<td>25</td>
<td>11</td>
<td>15</td>
<td>95</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.79</td>
<td>0.87</td>
<td>0.92</td>
<td><strong>0.74</strong></td>
<td>0.77</td>
<td>0.88</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>985.11</td>
<td>970.80</td>
<td>981.42</td>
<td>969.36</td>
<td>972.88</td>
<td>965.27</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>985.00</td>
<td>985.00</td>
<td>984.00</td>
<td>984.00</td>
<td>984.00</td>
<td>985.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.07</td>
<td>0.18</td>
<td><strong>0.02</strong></td>
<td>0.11</td>
<td>0.05</td>
<td>0.24</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.61</td>
<td>2.77</td>
<td>2.79</td>
<td><strong>2.49</strong></td>
<td>2.55</td>
<td>2.76</td>
</tr>
</tbody>
</table>
Figure D.14.: Demand, supply and requests evaluation of Adapt with BibSonomy trace on multi-tier application.
Figure D.15.: Demand, supply and requests evaluation of React with BibSonomy trace on multi-tier application.
Figure D.16.: Demand, supply and requests evaluation of Hist with BibSonomy trace on multi-tier application.
Figure D.17.: Demand, supply and requests evaluation of ConPaaS with BibSonomy trace on multi-tier application.
## D.1. Side-Evaluation Forecasting

Table D.7.: All results of the multi-tier evaluation of Telescope and TBATS.

<table>
<thead>
<tr>
<th>Tier</th>
<th>Metric</th>
<th>Telescope</th>
<th>TBATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ProvAccuracyU</td>
<td>0.0353</td>
<td>0.0198</td>
</tr>
<tr>
<td>1</td>
<td>ProvAccuracyO</td>
<td>0.2061</td>
<td>0.4308</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareU</td>
<td>0.1114</td>
<td>0.0623</td>
</tr>
<tr>
<td>1</td>
<td>ProvTimeShareO</td>
<td>0.4618</td>
<td>0.7209</td>
</tr>
<tr>
<td>1</td>
<td>Instability</td>
<td>0.2772</td>
<td>0.2166</td>
</tr>
<tr>
<td>1</td>
<td>AverageAmount</td>
<td>3.7</td>
<td>4.32</td>
</tr>
<tr>
<td>1</td>
<td>AmountAdapt</td>
<td>148</td>
<td>75</td>
</tr>
<tr>
<td>1</td>
<td>ASDeviation</td>
<td>0.87</td>
<td>0.90</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyU</td>
<td>0.0489</td>
<td>0.0403</td>
</tr>
<tr>
<td>2</td>
<td>ProvAccuracyO</td>
<td>0.1770</td>
<td>0.3571</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareU</td>
<td>0.1870</td>
<td>0.1416</td>
</tr>
<tr>
<td>2</td>
<td>ProvTimeShareO</td>
<td>0.4622</td>
<td>0.6991</td>
</tr>
<tr>
<td>2</td>
<td>Instability</td>
<td>0.4570</td>
<td>0.3508</td>
</tr>
<tr>
<td>2</td>
<td>AverageAmount</td>
<td>5.77</td>
<td>6.49</td>
</tr>
<tr>
<td>2</td>
<td>AmountAdapt</td>
<td>204</td>
<td>131</td>
</tr>
<tr>
<td>2</td>
<td>ASDeviation</td>
<td>0.94</td>
<td>0.95</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyU</td>
<td>0.0545</td>
<td>0.0396</td>
</tr>
<tr>
<td>3</td>
<td>ProvAccuracyO</td>
<td>0.1245</td>
<td>0.3203</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareU</td>
<td>0.1401</td>
<td>0.0963</td>
</tr>
<tr>
<td>3</td>
<td>ProvTimeShareO</td>
<td>0.2379</td>
<td>0.5802</td>
</tr>
<tr>
<td>3</td>
<td>Instability</td>
<td>0.1978</td>
<td>0.1662</td>
</tr>
<tr>
<td>3</td>
<td>AverageAmount</td>
<td>2.58</td>
<td>3.02</td>
</tr>
<tr>
<td>3</td>
<td>AmountAdapt</td>
<td>105</td>
<td>67</td>
</tr>
<tr>
<td>3</td>
<td>ASDeviation</td>
<td>0.79</td>
<td>0.86</td>
</tr>
<tr>
<td>overall</td>
<td>MeanRespTime</td>
<td>985.11</td>
<td>985.16</td>
</tr>
<tr>
<td>overall</td>
<td>MedianRespTime</td>
<td>985.00</td>
<td>985.00</td>
</tr>
<tr>
<td>overall</td>
<td>SLOViolations</td>
<td>0.07</td>
<td>0.07</td>
</tr>
<tr>
<td>overall</td>
<td>Deviation</td>
<td>2.61</td>
<td>2.71</td>
</tr>
</tbody>
</table>