Karlsruhe Institute of Technology

Resource Elasticity Benchmarking in Cloud Environments

Master Thesis of

Andreas Weber

At the Department of Informatics
Institute for Program Structures
and Data Organization (IPD)

Reviewer:	Prof. Dr. Ralf H. Reussner
Second reviewer:	Jun.-Prof. Dr.-Ing. Anne Koziolek
Advisor:	Dipl.-Inform. Nikolas R. Herbst
Second advisor:	Dr.-Ing. Henning Groenda

Duration: January 15th, 2014 – July 14th, 2014

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.

Karlsruhe, July 14th, 2014

.....

(**Andreas Weber**)

Acknowledgements

I would like to thank Amazon for providing a research grant that allowed me to evaluate the applicability of the benchmarking approach on a public cloud without any costs.

Special thanks go to my advisors Nikolas Herbst and Henning Groenda. Both supported me with ideas, inspiring discussions and detailed constructive feedback. Their outstanding supervision contributed invaluably to the successful planning and realization of this thesis.

I would like to thank Prof. Samuel Kounev for his advise and for encouraging me to present my work even in an early state at an ICPE 2014 Conference Workshop in Dublin. The Conference was a great experience and allowed me to enhance my work with feedback from professional researchers.

My thanks go to the research students of the FZI Student Lab with whom I spent a great time in course of the last half year. Particularly, I thank Jóakim v. Kistowski for various discussions about LIMBO, elasticity benchmarking and for helping me to develop creative benchmark and metric names.

Finally, my warm thanks go to my family and friends who supported me in all their individual ways.

Zusammenfassung

Anbieter von modernen Cloud-Diensten offerieren insbesondere auf Infrastrukturebene (Infrastructure-as-a-Service - IaaS) in den meisten Fällen die Möglichkeit, Resourcen an den aktuellen Bedarf des Kunden anzupassen. Die Fähigkeit eines Dienstes sich dynamisch an Lastschwankungen anpassen zu können, wird im Cloud-Kontext als Elastizität bezeichnet. Der Vergleich von Cloud-Diensten hinsichtlich der Qualität der Elastizität ist eine Herausforderung, da es bisher noch keine verlässlichen Messmethodiken und Metriken zur Bewertung der unterschiedlichen Aspekte von Elastizität gibt.

Diese Masterarbeit analysiert existierende Ansätze zur Messung von Elastitzität und stellt einen neuen Ansatz zum Benchmarken von elastischen Systemen vor. Dieser basiert auf der Idee das zu evaluierende System einem realistischen Lastintensitätsverlauf auszusetzen, um damit eine schwankende Nachfrage nach Resourcen zu induzieren. Zeitgleich werden die tatsächlich bereitgestellten Resourcen überwacht und im Anschluss an die Messung mit dem rechnerisch nötigen Resourcenbedarf verglichen. Der Vergleich erfolgt mittels Metriken, welche die Elastitzität hinsichtlich der Genauigkeit und des zeitlichen Verhaltens bewerten. Um einen fairen Vergleich von verschiedenen Systemen auch bei unterschiedlicher Effizienz der zu Grunde liegenden Resourcen zu ermöglichen, wird der Lastintensitätsverlauf vor der Messung systemspezifisch so angepasst, dass auf allen Systemen im Vergleich die gleichen Nachfragevariationen induziert werden.

Das Benchmarkkonzept untergliedert die Elastizitätsanalyse in vier Schritte: Zunächst wird im Rahmen einer *System Analyse* die zu evaluierende Plattform bezüglich ihres Skalierungsverhaltens und der Effizienz der zu Grunde liegenden Resourcen ausgewertet. Das Resultat wird dann in einer *Kalibrierung* genutzt, um ein gegebenes Lastintensitätsprofil systemspezifisch anzupassen. Im eigentlichen *Messschritt* wird eine variierende Last entsprechend des angepassten Lastintensitätsprofils generiert und die Resourcennutzung auf der zu evaluierenden Plattform überwacht. Die abschließende *Auswertung* beurteilt das beobachtete elastische Verhalten mittels der entwickelten Metriken.

Im Rahmen dieser Arbeit wird das Benchmarkkonzept mit der Entwicklung eines javabasierten Frameworks - genannt BUNGEE - zur Messung der Elastizität von IaaS-Cloud-Plattformen umgesetzt. Aktuell ermöglicht *BUNGEE* die Evaluation von Clouds, die virtuelle Maschinen horizontal skalieren und auf CloudStack oder Amazon Web Services (AWS) basieren.

In einer umfassenden Evaluation zeigt die Arbeit, dass die entwickelten Elastizitätsmetriken in der Lage sind, unterschiedliche elastische Systeme in eine ordinale und konsistente Ordnung zu bringen. Eine Fallstudie belegt darüber hinaus die Anwendbarkeit des Benchmarkkonzeptes in einem realitätsnahen Szenario unter Verwendung eines realistischen Lastintensitätsprofils, welches mehrere Millionen Anfragen modelliert. Die Fallstudie zeigt die Anwendbarkeit sowohl auf einer privaten als auch auf einer öffentlichen AWS basierten Cloud unter Verwendung von elf verschiedenen Konfigurationen von Elastizitätsregeln und vier verschieden effizienten Instanztypen von virtuellen Maschinen.

Abstract

Auto-scaling features offered by today's cloud infrastructures provide increased flexibility, especially for customers that experience high variations in the load intensity over time. However, auto-scaling features introduce new system quality attributes when considering their accuracy and timing. Therefore, distinguishing between different offerings has become a complex task, as it is not yet supported by reliable metrics and measurement approaches.

This thesis discusses the shortcomings of existing approaches for measuring and evaluating elastic behavior and proposes a novel benchmark methodology specifically designed for evaluating the elasticity aspects of modern cloud platforms. The benchmarking concept uses open workloads with realistic load intensity profiles in order to induce resource demand variations on the benchmarked system and compares them with the actual variation of the allocated resources. To ensure a fair elasticity comparison between systems with different underlying hardware performance, the load intensity profiles are adjusted to induce identical resource demand variations on all compared platforms. Furthermore, this thesis proposes new metrics that capture the accuracy of resource allocations and deallocations, as well as the timing aspects of an auto-scaling mechanism, explicitly.

The benchmark concept comprises four activities: The *System Analysis* evaluates the load processing capabilities of the benchmarked platform for different scaling stages. The *Benchmark Calibration* then uses the analysis results and adjusts a given load intensity profile in a system specific manner. Within the *Measurement* activity, the evaluated platform is exposed to a load varying according to the adjusted intensity profile. The final *Elasticity Evaluation* measures the quality of the observed elastic behavior using the proposed elasticity metrics.

A java based framework for benchmarking the elasticity of IaaS cloud platforms called *BUNGEE* implements this concept and automates benchmarking activities. At the moment, *BUNGEE* allows to analyze the elasticity of CloudStack and Amazon Web Service (AWS) based clouds that scale CPU-bound virtual machines horizontally.

Within an extensive evaluation, this thesis demonstrates the ability of the proposed elasticity metrics to consistently rank elastic systems on an ordinal scale. A case study that uses a realistic load profile, consisting of several millions of request submissions, exhibits the applicability of the benchmarking methodology for realistic scenarios. The case study is conducted on a private as well as on a public cloud and uses eleven different elasticity rule configurations and four instance types assigned to resources with different levels of efficiency.

Publications and Talks

Refereed Workshop Paper

[WHGK14]

A. Weber, N. R. Herbst, H. Groenda and S. Kounev, “Towards a Resource Elasticity Benchmark for Cloud Environments", in *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, March 2014.

Invited Talk

“Towards a Resource Elasticity Benchmark for Cloud Environments", at the *SPEC RG Annual Meeting 2014*, Dublin. March 26th, 2014.

Contents

1. Introduction

Context

In course of the last years, the usage of cloud based services such as GoogleMail or Dropbox became part of the everyday life of many people. With an ongoing consumerization, the popularity of cloud based solutions in industry is increasing, too. The Cloud Accounting Institute yearly conducts a survey where accounting professionals are asked about their current and intended use of cloud solutions [Ins13]. Between 2012 and 2013, the percentage of respondents that claim to use cloud solutions increased from 52% to 75%. When asked for the expected benefits of using cloud solutions, more than half of the respondents mention reducing cost as one of the benefits.

Cloud providers nowadays offer their services with a “Pay-Per-Use” accounting model to increase flexibility and efficiency with respect to traditional offers. Customers can specify their demand and pay accordingly. When the demand is changing, the customer asks the provider for a scaled version of the service and uses them for an adjusted price. A further step is providing *elasticity*. Elasticity means dynamic scaling of resources over time according to the recent demand. With an elastic cloud service, the customer does not have to specify his demand himself. The provider dynamically adapts the offered service according to the customer’s demand and the customer pays for the actually consumed resources. This business model is referred to as "Utility Computing" [AFG+10].

Motivation

Researchers have proposed various elasticity strategies that define adaptation processes for cloud systems as summarized and compared in the surveys of Galante et al. [GB12] and of Jennings and Stadler [JS14]. These elasticity strategies can be rather simple and rule based or use advanced techniques such as load forecasting in order to provision resources in time. A benchmark can help to evaluate the realized elasticity and allows to compare different strategies against each other.

Cloud providers often offer tools that allow customers to implement scaling rules to define the elastic behavior. Varying the parameters of these rules leads to different behaviors. Finding the optimal parameter configuration is not trivial. A benchmark can help customers to compare these parameter configurations in an objective manner.

Besides the used elasticity strategy and its parameter configuration, elasticity is also influenced by other factors such as the underlying hardware, the virtualization technology

or the used cloud management software. These factors vary across providers and are often unknown to the cloud customer. Therefore, even when cloud providers offer the same strategy and the customer configures them identically, the quality of the elastic behavior can be very different. Again, a benchmark allows to evaluate and compare the resulting elasticity.

State of the Art

Previous works [BKKL09, LYKZ10, DMRT11, LOZC12, CGS13] in the field of analyzing elasticity often evaluate elasticity only to a limited extend. For example, they just measure the elasticity aspect timing but not the accuracy aspect or vice versa. Additionally, elasticity is often not measured as a distinct attribute but is mixed up with efficiency. Furthermore, the employed load profiles for benchmarking do not reflect a realistic variability of the load intensity over time.

Other approaches [Wei11, FAS+12, ILFL12, Sul12, MCTD13] take a business perspective when evaluating elasticity. They analyze the financial impact of choosing between different elastic cloud solutions. This is a valid approach for a customer who must make cost based decision between alternative cloud offerings. However, this approach mixes up the evaluation of (i) the business model, (ii) the performance of underlying resources and (iii) the technical property elasticity.

Approach

This thesis focuses on the evaluation of the technical property elasticity in the Infrastructure as a Service (IaaS) context. To stress that scaling in the IaaS context is realized by scaling of the underlying resources, the term resource elasticity will be used throughout the thesis. The thesis refines and extends an existing concept [Her11] for evaluating resource elasticity. In addition, it presents the benchmarking framework BUNGEE that implements the concept and allows to benchmark real cloud platforms.

The main idea for evaluating resource elasticity bases on comparing a changing resource demand over time with the actual allocation of resources that is triggered by an elasticity mechanism. The varying resource demand is induced by resource specific workloads. To allow the usage of workloads with a realistic variation in load intensity, the framework incorporates the modeling of characteristic load variations. Different levels of hardware efficiency on the compared systems has effects on their scaling behavior and can hamper an objective evaluation. This issue is tackled by analyzing the benchmarked systems with respect to the efficiency of their underlying resources and their scaling behavior. The results of this analysis are used to adjust the load intensity in a way that all systems are stressed in a comparable manner. Hence, the induced resource demand is equal on the compared systems.

Based on previous research [Her11], this thesis proposes simple intuitive and effective metrics for characterizing the elasticity of a system. These metrics compare the system specific resource allocation curve with an system independent resource demand curve. Different metrics allow to measure different aspects concerning accuracy and timing, separately. In addition, this thesis discusses how the developed metrics can be used to compare the elasticity of a targeted system to a baseline system.

The benchmarking approach is evaluated on a private cloud as well as on public AWS based cloud. The evaluation analyzes the reproducibility of the *System Analysis* and the effect of using a simplified analysis version. The metrics are evaluated towards their ability to consistently rank different degrees of elasticity on an ordinal scale. In a case study, the benchmarking capabilities for a realistic scenario are demonstrated. The study

uses a realistic load profile, consisting of several millions of request submissions, and is conducted using virtual machine (VM) instance types that differ in terms of the levels of efficiency of the resources assigned to them.

1.1 Goals and Research Questions

This section lists the main goals for the envisioned thesis. The different aspects of the goals are specified as research questions that have to be answered in order to accomplish the goal.

Goal 1: Identify the key characteristics of elasticity and important properties for an elasticity benchmark.

- RQ 1.1: What are the prerequisites for a meaningful comparison of different elastic behaviors?
- RQ 1.2: What are the relevant aspects of resource elasticity?
- RQ 1.3: What are important properties of a benchmark that targets the measurement of resource elasticity?

Goal 2: Analyze existing approaches for measuring elasticity and their limitations.

- RQ 2.1: What is the focus of existing measuring and benchmarking approaches?
- RQ 2.2: What are the limitations of existing measurement methodologies?

Goal 3: Develop a concept for evaluating resource elasticity of IaaS cloud platforms.

- RQ 3.1: How can workloads suitable for elasticity benchmarking be modeled?
- RQ 3.2: How can a matching function that maps load intensities to resource demands be derived for a cloud system under test (CSUT)?
- RQ 3.3: How can a modeled load intensity curve be adjusted in a way that it induces the same resource demands over time on systems with different levels of efficiency?
- RQ 3.4: How can the resource demand that was induced by exposing the system to a load be extracted?
- RQ 3.5: How can the amount of allocated resources, the resource supply, be monitored?

Goal 4: Measure elasticity by comparing the actual resource supply with the resource demand that a realistic dynamic load induces.

- RQ 4.1: Which metrics can be derived to measure the different aspects of elasticity?
- RQ 4.2: How can the metric results be used in order to create a ranking within a group of different CSUTs?

Goal 5: Build an elasticity benchmarking framework which allows to evaluate the elasticity of IaaS cloud platforms that scale CPU-bound resources horizontally.

This goal is not connected to specific research questions but it includes known software engineering tasks such as selecting an appropriate architecture and design, specification of interfaces as well as documentation and testing.

Goal 6: Evaluation of the *System Analysis* and the elasticity metrics.

- RQ 6.1: Is the *System Analysis* reproducible?
- RQ 6.2: How big is the deviation between the real resource demand and an linearly extrapolated resource demand when the test system uses more than one resource unit?
- RQ 6.3: Do the developed metrics allow to rank the benchmarked systems on an ordinal scale?

1.2 Thesis Structure

The remainder of this thesis is structured according to the main gain goals as follows:

Chapter 2 describes several foundations for the context of resource elasticity benchmarking. The foundations include a blue print for an elastic cloud architecture, the definition and discrimination of important terms, information about the variety of existing elasticity strategies and explanations of requirements for elasticity targeted benchmarks.

Related work in the field of evaluating elasticity is analyzed in Chapter 3.

Chapter 4 describes the concept of the benchmark in greater detail. After a coarse grained overview of the benchmark, this chapter describes the main components of the benchmark: The modeling and generation of realistic workloads, the *System Analysis* and the *Benchmark Calibration* as a way for overcoming different levels of hardware efficiency and the extraction of resource demand and supply during the *Measurement*.

The metrics which are used to measure elasticity in the final *Elasticity Evaluation* are discussed separately in Chapter 5. It explains metrics for the different elasticity aspects and discusses ways for aggregating them into a single elasticity measure. Furthermore, metrics which have been considered but were rejected for the benchmark are discussed.

Chapter 6 outlines the architecture and the design of the benchmarking framework *BUNGEE* which was developed based on the benchmarking concept in course of this thesis.

Chapter 7 evaluates the *System Analysis* as well as the elasticity metrics and illustrates the applicability of the benchmark within a case study.

Possible future extensions and evaluations are discussed in Chapter 8, before Chapter 9 concludes the thesis.

2. Foundations

This chapter provides some relevant background for elasticity benchmarking and thus addresses the first goal mentioned in Section 1.1. It starts with a description of the architecture of elastic cloud systems and a definition of the (cloud) system under test in Section 2.1. This description is followed by Section 2.2 which explains terms commonly (mis-)used in the cloud context. After this differentiation, resource elasticity is analyzed more detailed in Section 2.3. The final Section 2.4 presents requirements for benchmarking in the context of measuring resource elasticity.

2.1 Elastic Cloud System Architecture

Figure 2.1 shows a blueprint architecture of a simple elastic cloud system. Elastic cloud systems typically consist of two components: The scalable infrastructure and a management system.

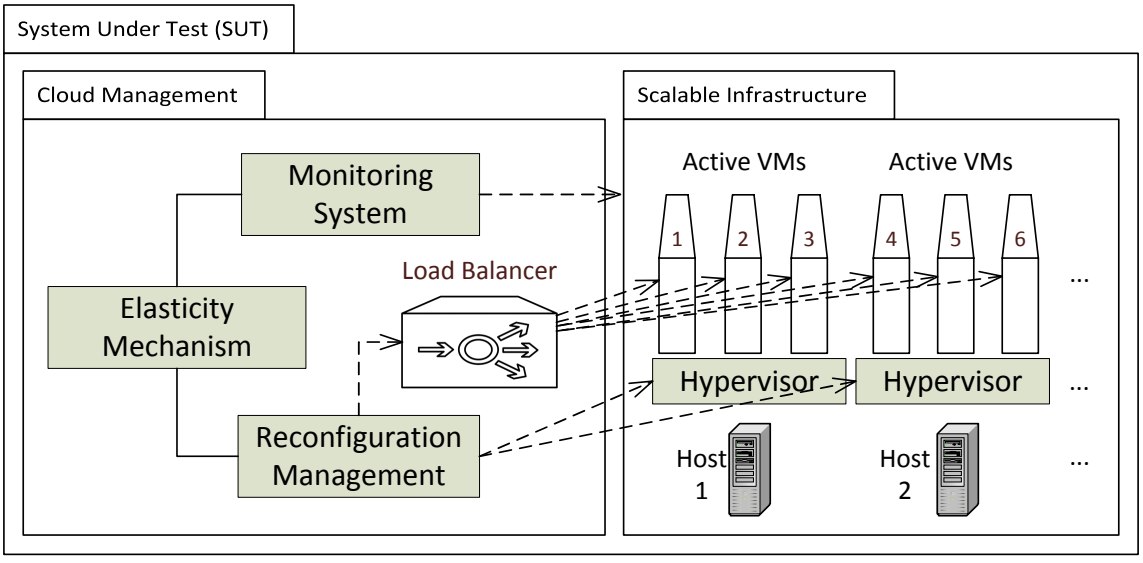


Figure 2.1: Blueprint architecture of a resource elastic system

As a basic service, cloud providers offer infrastructure to their customers in form of VMs with network access and storage. This service is called IaaS [MG11]. The VMs are

hosted on a hypervisor which acts as virtualization layer that allows a shared usage of the underlying physical hardware. When customers need more resources they have - depending on the provider - at least one of two options. They can either ask the provider to assign more resources to their VMs (scale up) or request additional VM instances (scale out). Sometimes even a combination of both methods is possible. The first option is limited by the amount of resources the underlying hardware can provide. As soon as multiple instances are available, incoming load must be distributed. This task is performed by a *load balancer*. It forwards incoming requests according to a configured scheme, e.g., round robin, to the VM instances.

The scaleable infrastructure is managed by a *cloud management server*. It offers different services via modules. The *reconfiguration management* module supports the creation of new VMs and allows starting and stopping them. A *monitoring* module allows the collection of monitoring data about the VMs and about the underlying physical infrastructure. The *load balancer* can be part of the *cloud management server* but can also be an external module. Often, the *cloud management server* also offers an *elasticity mechanism*. This mechanism uses monitoring data and triggers reconfigurations of the scalable infrastructure according to an *elasticity strategy*. It also reconfigures the *load balancer* if this is required due to a reconfiguration of the elastic system. Thus, the system adapts itself according to the demand and the customer does not need reconfigure the system himself everytime his demand changes. The software running on the cloud management server is called cloud management software.

Cloud System Under Test

The *cloud system under test (CSUT)* defines the boundaries of the system evaluated by an elasticity benchmark. The *CSUT* for the benchmark developed in course of this thesis includes the following components that impact the resulting elastic behavior:

- The scalable infrastructure
- The load balancer
- The cloud management server, including
  - The reconfiguration management
  - The monitoring system
  - The elasticity mechanism

2.2 Terms and Differentiation

In the context of cloud computing the terms efficiency, scalability and elasticity are commonly used without a clear distinction by referring to a precise definition . Although these terms are related to each other, they describe different properties. This section explains the meaning of each property in the context of cloud computing and the relations between them.

2.2.1 Efficiency

The Oxford Dictionary [OED14a] defines efficiency for the context of systems and machines as “achieving maximum productivity with minimum wasted effort or expense”. The way productivity and wasted effort are measured strongly depends on the context. For computing systems the term efficiency is tightly coupled with performance and can be split up into cost efficiency, energy efficiency or resource efficiency.

Cost Efficiency describes to what degree a system is able to achieve maximum productivity with minimum costs.

Energy Efficiency describes to what degree a system is able to achieve maximum productivity with minimum energy consumption.

Resource Efficiency either describes to what degree a system is able to achieve maximum productivity with minimal use of resources (system property), or describes the level of efficiency of an underlying resource unit (resource property).

For efficiency measurements, black box approaches are commonly used.

2.2.2 Scalability

The term scalability is used in various contexts and often in a way that important aspects of scalability get lost. To gain a better understanding, the next paragraph presents some general insights about scalability before the term is examined in the cloud context.

General Findings

Scalability describes the degree to which a subject is able to maintain application specific quality criteria when it is applied to large situations. Although the term is frequently used, statements about scalability often lead to just a vague impression about the analyzed subject [DRW06]. Many authors have tried to overcome this issue by proposing own definitions or systematic ways to analyze scalability. The most important insights that are shared by several authors are summarized in the following paragraphs.

Scalability is fulfilled within a range according to a specific quality. Therefore sentences like "The system is scalable" do not provide much insight. Every system is scalable to some extend. Discriminating is the range within and the quality to which it is scalable. Whereas the range is typically specified by an upper scaling bound, the quality usually describes the growth of a measured quality criteria. Possible qualities include linear or exponential growth, for example.

Scalability refers to *input variables* that are scaled. Scalability describes how the subject reacts when one or more input variables, sometimes referred to as attributes [vSVdZS98] or independent variables [DRW06], are varied. Examples for such *input variables* are problem size, number of concurrent users or number of requests per second.

Scalability is measured by evaluating at least one quality criteria. To measure how the subject reacts, one or more quality criteria have to be observed while input variables are varied. These quality criteria are sometimes referred to as performance measures [vSVdZS98] or dependent variables [DRW06]. Examples for quality criteria are memory consumption, I/O device usage, or response time.

Scalablity in Clouds

With the help of the above explained terms *input variable* and *quality criteria*, scalability in the cloud context can be described more precisely than commonly practiced.

Input Variable

Typically the *input variable* for scalability analysis of cloud systems is load intensity. It describes how much work a system has to handle in a given time span. Load intensity can be varied either by different work unit sizes or by varying the arrival rate of work units.

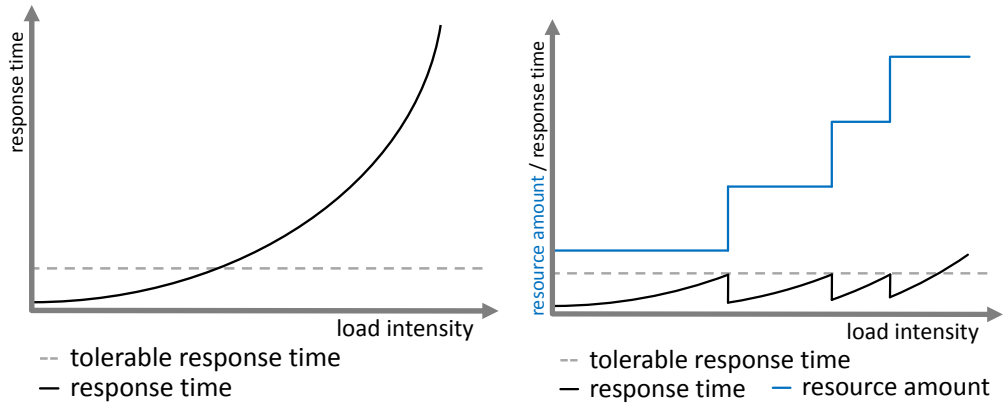
Quality Criteria

There are two kinds of *quality criteria* for cloud systems: service levels and used resource amount.

Service Levels: A service level can be described by measures like response time or abort rate. Cloud customers usually specify service level objectives (SLOs) which define the minimal acceptable service level for their application. Service levels are normally specified with the help of probabilities for or probability distributions over the measures. For example: "95% of all response times should be below one second". SLOs are often part of a service level agreement (SLA) that contains multiple SLOs.

Resource Amounts: Resources are required means to conduct certain types of work. The amount of consumed resources can be measured for different resource types and at different abstraction levels. Different types of physical resources are: processing resources like Central Processing Units (CPUs) or Graphics Processing Units (GPUs), memory resources like random access memory or storage resources like hard disk drives. Resources can also be software resources like server instances, threads or locks. Different abstraction levels cater for different granularities. For processing resources for example, the resource amount can be measured by the number of used CPU cycles, physical CPUs, VMs. The latter one is a special case as a VM is a container resource, that contains several other resources.

Cloud customers typically want to offer their end users a constant service level which is independent of the *input variable* load intensity. Thus, *quality criteria* that are defined in SLOs should always be satisfied. This means the used resource amount characterizes the scaling behavior, as it has to increase when the load intensity increases. To emphasize that the scaling behavior of a cloud system is based on scaling of underlying resources the term *resource scaling* will be used throughout this thesis when referring to such systems.



(a) System fixed resource amount

(b) System with resource scaling

Figure 2.2: Resource scaling allows cloud systems to comply with predefined service levels even for increased load intensity

Figure 2.2 illustrates the difference between a system that uses resource scaling and one that does not. Here a maximal tolerable response time is defined as service level. However, other measures that define a service level are possible, too. In case the amount of resources for a system is fixed, the system's response time will increase

when the load intensity increases. As soon as the response time exceeds the predefined threshold the system is not usable anymore. The scalablity of this system with respect to response time is therefore very limited.

In contrast, the system whose underlying resources can be scaled is able to comply with the maximum tolerable response time even for a higher load intensity. The scalability with respect to response time of this system is higher compared to the system without resource scaling. Still, the scalability is limited - as the maximum amount of underlying resources is limited.

Note that the exponential increase of the response time is just exemplary. Other growth characteristics are also possible. Moreover, other measures that measure a service level could be put in place of response time, too.

Scalability is a Static Property

It is important to understand that scalability does not contain any temporal aspect. In the context of cloud computing, scalability does not make any assumption about when the resources are scaled. Scalability just describes how much additional resources a system needs when the load increases to be able to offer a constant service level. Thus, scalability does not provide any information about the system's ability to scale resources on demand in a fast and accurate manner, it even does not make any assumptions about the existence of an - automated - scaling mechanism.

Scaling Method

Resource scaling can be achieved in two different ways, often referred to as scaling dimensions:

Vertical Scaling or scaling up/down refers to varying the amount of resources by adding/ removing resources to an existing resource node. Looking at computing resources for instance, scaling up can mean adding CPU time slices shares or additional CPU cores to a node. As the underlying physical hardware is limited, vertical scaling is only possible to some extend. This is true for low-level resources like CPUs but also for high level resources like threads in a thread pool, whose maximum pool size is a given parameter of the underlying hardware.

Horizontal Scaling or scaling out/in refers to varying the amount of resources by adding/removing resource nodes to a cluster. One example is the allocation of an additional VM. The added VM can be located at the same physical location like previous ones or at another remote location. Horizontal scaling typically is more expensive than vertical scaling since the allocation of new nodes and additional communication causes significant overhead. Depending on the application and the scaling architecture, scaling out can in some cases even lead to a *decreasing* service level.

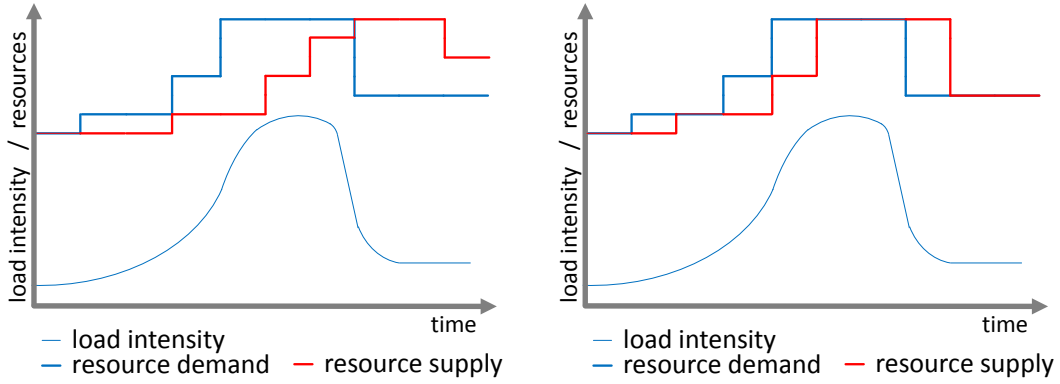
Migration is mentioned in [GB12] as a third scaling method. It describes the transference of a VM from one physical location to another for global infrastructure or locality optimization. Since the number of assigned resources typically changes but the number of virtual instances does not, migration can be treated as a special case of vertical scaling.

2.2.3 Elasticity

Elasticity is known in physics and likewise in economics. In physics [OED14b], elasticity is a material property that describes to which degree a material returns to its original state after being deformed. In economics, elasticity describes the responsiveness of a dependent variable to one or more other variables [CW09]. On a high level of abstraction one can argue elasticity captures how a subject reacts to changes that occur in its environment.

For the context of cloud computing elasticity was previously analyzed in [HKR13]. This thesis builds upon this work and further refines it. While scalability - in the cloud context - describes the degree to which a system is able to adapt to a varying load intensity by using a scaled resource amount, elasticity reflects the quality of the *adaptation process* in relation to load intensity variations over time. Thus, elasticity adds a temporal component to scalability. As elasticity describes properties of an adaptation process, elasticity requires the existence of a mechanism that controls the adaptation.

Before analyzing resource elasticity in detail in Section 2.3, the effect of different degrees of resource elasticity in cloud systems is illustrated by a simple example.



(a) System A (b) System B

Figure 2.3: Different degrees of elasticity due to different elasticity mechanisms

Figure 2.3 shows the behavior of two systems that are equal except for their elasticity mechanisms. In particular, their underlying resources have the same efficiency and the scalability of both systems is equal as well. Thus, for an arbitrary load intensity, both systems require the same amount of resources to comply with predfined SLOs. Thus, both systems have the same resource demand. In this example the second system exhibits a higher degree of elasticity. The red curve - resource supply - matches the blue curve - resource demand - better comparing System A to System B. System A's adaptation process reacts faster and more precise to changes in load intensity than the one of System B. To compare the elasticity of both systems in a quantitative manner metrics are required. The metrics which have been developed in course of this thesis are explained in Chapter 5.

Comparing elasticity in this simple case is easy. It becomes more complex, when the system's underlying resources have different levels of efficiency or exhibit different scaling behaviors. To cope with these difficulties elasticity is analyzed in detail in Section 2.3.

2.2.4 Relation and Differentiation

Efficiency is a term that can be applied to both, a part of a system, e.g., a single resource (resource property), or an entire system (system property). In any case it reflects the ability of the subject to process a certain amount of work with smallest possible effort.

Improving efficiency of underlying resources normally results in a better efficiency for the whole system.

Scalability describes the degree to which a system is able to adapt to a varying load intensity by using a scaled resource amount to maintain a predefined service level. Improving scalability normally means reducing scaling overhead and therefore leads to improved efficiency (system property). In contrast, an improved efficiency of the underlying resources does not necessarily result in an improved scalability, e.g., quality attributes such as the response time can still increase exponentially even for an improved efficiency of the underlying resources.

Elasticity reflects the sensitivity of a system's scaling process in relation to load intensity variations over time. Thus, scalability is a prerequisite for elasticity. Normally, a higher degree of elasticity results in higher efficiency (system property) since a high degree of elasticity implies appropriate resource allocation and usage. The other way around this implication is not necessarily given. No direct implications exist between scalability and elasticity or vice versa.

The fact that efficiency and scalability do not determine elasticity entirely, strengthens the consideration to treat elasticity as an individual property of a cloud computing environment.

2.3 Resource Elasticity

This section presents a definition for resource elasticity and explains it. Afterwards, Subsection 2.3.2 illustrates the prerequisites for measuring elasticity and thereby answers RQ 1.1. The following Subsection 2.3.3 explains the core aspects of elasticity and thus addresses RQ 1.2. Finally, Subsection 2.3.4 gives a brief overview about existing elasticity strategies that can be used when implementing elasticity mechanisms.

2.3.1 Definition

In [HKR13] the following definition for resource elasticity was proposed:

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

Several important aspects can be derived from this definition. Previous informal definitions included them to some extend but not with respect to all points:

Elasticity is

"... the degree to which ..." As true for scalability, elasticity is not a feature which is fulfilled or not. Elasticity is measurable and therefore it should be possible to compare the degree elasticity for different systems to each other. Nevertheless, some prerequisites have to be fulfilled in order to allow elasticity comparisons. These prerequisites are discussed in the next section.

"... a system is able to adapt ... (in an autonomic manner)" A system which is able to adapt needs a defined adaptation process. This process specifies how and when the system adapts. Normally, the process should be automated to ensure a consistent adaptation behavior.

"... to load changes ..." In a realistic cloud scenario, load intensity changes over time. Thus, a benchmark that measures elasticity should model the variability of load intensity in a realistic way to enforce a realistic behavior of the evaluated elastic systems.

"... by provisioning and deprovisioning resources ..." Elasticity includes both: Provisioning resources when demand increases *and* deprovisioning them when demand decreases.

"... resources match the current demand as closely as possible." As a close match between resource demand and availability is desired, comparing both is the central point for evaluating elasticity.

2.3.2 Prerequisites

Before evaluating resource elasticity several prerequisites should be checked beforehand cf. [HKR13].

Autonomic Scaling: Elasticity is the result of an adaptation process that scales resources according to the load intensity. Evaluation of elasticity therefore requires that this process is specified. The adaptation process is usually realized by an automated mechanism. However, the adaptation process could also contain manual steps. A notable aspect in the latter case is that repeatability of measurements in that case may be limited.

Resource Type: Elastic systems scale resources. The type of resources can be quite different: There are base resources like CPU, memory or disk storage and there are container resources, which comprise several base resources and are very common in cloud systems. To avoid comparing apples to oranges when evaluating elasticity, systems should be compared that use the same resource types.

Resource Scaling Unit: The amount of used resources can be measured in different units, e.g., CPU time slice shares, processors or VMs. If elasticity is analyzed by comparing resource demands to actual resource consumption, it is crucial to use the same units when comparing different systems.

Scaling Method: The different scaling methods are explained in Section 2.2.2. Comparing elastic systems that are based on different scaling dimensions may be desirable. Nevertheless, this should be done with care as the choice about the scaling method may have side effects such as different resource scaling units.

Scalability Bounds: The scalability of every system is limited. The scalability bounds depend on the maximum amount of available physical resources and on the service level constraints that are specified in SLOs. Elasticity comparisons should be performed within a scaling range that is supported by all compared systems.

2.3.3 Core Aspects

Definition 2.3.1 states that elasticity measures the degree to which a system is able to (de-)provision resources in a way that demand and provided resources "match as closely as possible". This definition helps to understand the meaning of perfect elasticity. By changing the perfect elastic behavior, it is possible to gain insights of the core aspects of elasticity.
Figure 2.4 illustrates an artificial system A with perfect elasticity. The curves for resource demand and allocated resources are equal and thus the property "match as closely as possible" is perfectly fulfilled. To illustrate different aspects of elasticity the curve for allocated resources is now deformed, and therefore the elastic behavior is changed systematically.

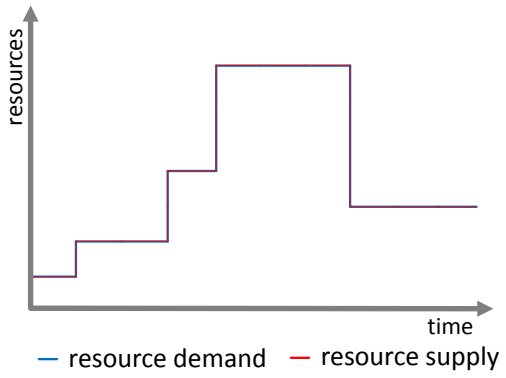
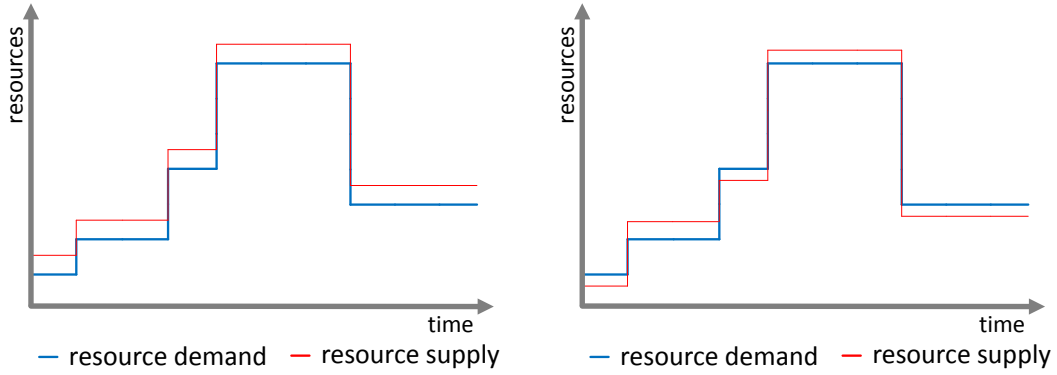


Figure 2.4: System A: Ideal elasticity

Accuracy



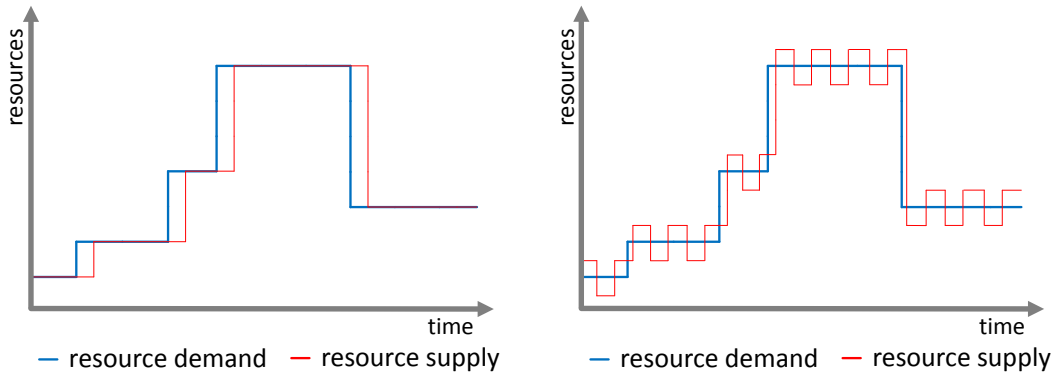
(a) System B (b) System C

Figure 2.5: Systems with imperfect accuracy

Subfigure 2.5(a) shows a system that over-provisions at all times. This could be due to a very conservative adaptation process, aiming to never violate SLOs. Although this system B reacts very fast, it does not match the demand as closely as possible and should therefore be considered less elastic than the ideal elastic System A. Subfigure 2.5(b) shows another System C that also always adapts at the exact points where the demand changes. But, in contrast to system B it over-provisions and under-provisions. Systems B and C have in common that they seem to react immediately when demand changes. But although they react fast, both systems do not match the demand very accurately. Thus, *accuracy* can be seen as one core aspect of elasticity.

Timing

Another way how the curve for available resources can be deformed is illustrated in Figure 2.6. Subfigure 2.6(a) shows the behavior of a hypothetical system D, which is able to match the resource demand, but with some delay. System D could be a system that needs some time to perform its adjustments after the resource demand changes. Similar, one can imagine a system that performs allocation activities in advance before the demand actually changes. Such a system foresees changes too early. A further way how the curve for available resources can be modified is shown in Subfigure 2.6(b). Whereas, the curve for available resources generally matches the curve for the resource demand, the available



(a) System D (b) System E

Figure 2.6: Systems with imperfect timing

resources seem to be updated with - an unnecessary - high frequency. It can be argued that systems D and E have a timing behavior that is not ideal. Therefore, the *timing* of the adaptation process can be seen as a second core aspect.

It is valid to argue that system E not only has a bad *timing* but also its *accuracy* is not optimal. Although *accuracy* and *timing* are no orthogonal dimensions, these core aspects help to describe and compare different elastic behaviors in a structured way.

Metrics that capture the core aspects of elasticity are presented in Chapter 5 and evaluated in Section 7.3.

2.3.4 Strategies

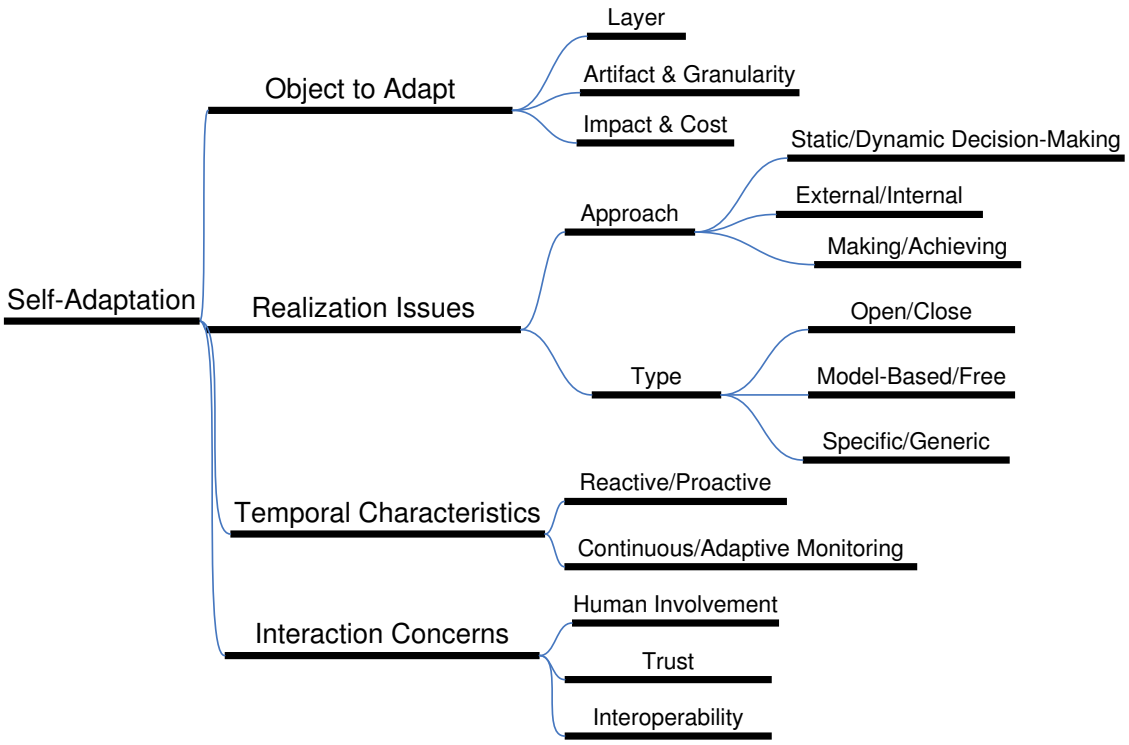
This section gives a short overview of existing elasticity strategies which can be used when implementing elasticity mechanisms and shows how they can be classified according to a taxonomy. The broad variety of different elasticity strategies warrants the need for a benchmark to evaluate the quality of different strategies.

A cloud system with resource elasticity is a self-adaptive system. Resources - as part of the system - are allocated according to a changing demand. In their journal article "Self-Adaptive Software: Landscape and Research Challenges"[ST09] Salehie and Tahvildari present a taxonomy of self-adaptive systems. The taxonomy is shown in figure 2.7(a). Although Salehie and Tahvildari target self-adaptive systems on a high abstraction level, most variation points are applicable to systems with elastic resource scaling.

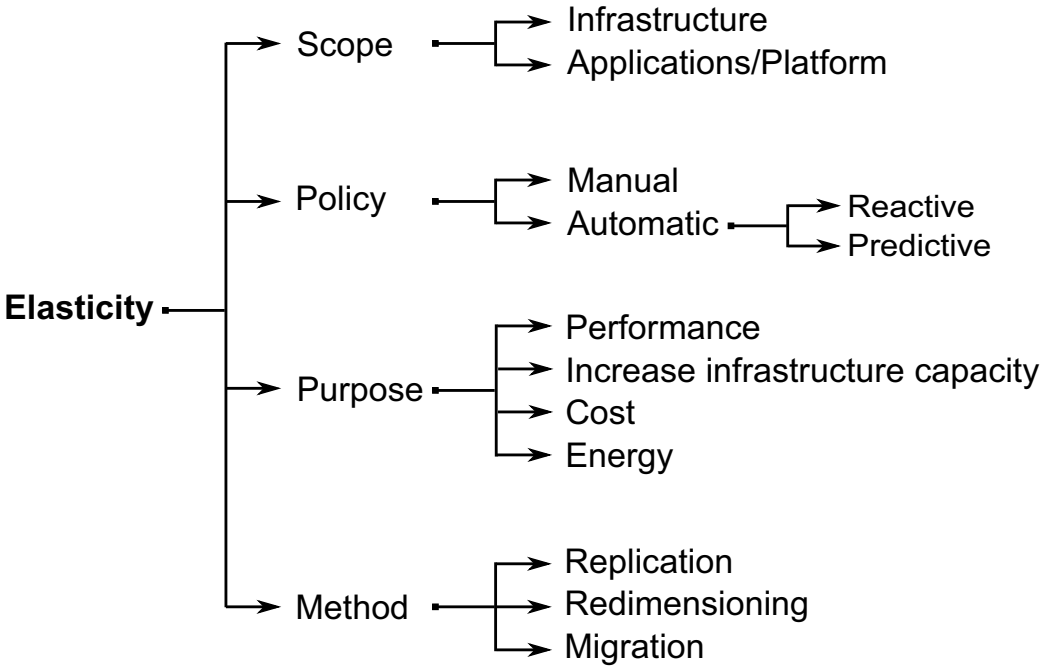
Galante and de Bona present in their survey about cloud computing elasticity [GB12] a comparable taxonomy targeted at resource elasticity. This taxonomy is shown in figure 2.7(b).

Without going into too much detail or explicitly picking advantages of an individual strategy, some relevant aspects that appear in at least one of the taxonomies are highlighted in the following. Hereby, aspects limiting the comparability as well as aspects that motivate the need for a benchmark are emphasized.

The target *abstraction layer* for elasticity strategies, i.e., IaaS, PaaS, can be different. This is one reason why the *unit* of the scaled resources or even the *type* of the considered unit can be different. Elasticity strategies can make use of different *scaling methods* to adjust the amount of available resources. As outlined in Section 2.3.2 resource type, resource unit and *scaling method* can limit the comparability of elasticity.



(a) Self-adaptive systems [ST09]



(b) Elastic Systems [GB12]

Figure 2.7: Taxonomies for (a) self-adaptive systems and (b) elastic systems

Elasticity strategies can be *reactive* or *proactive/predictive*. Reactive strategies start the adaptation process as soon as they detect a changed demand. Due to the time needed for the adaptation itself, available resources match the demand only after some delay. Predictive systems extend reactive ones. They try to foresee demand changes in order to provision the correct amount of resources in time. By intuition, strategies that contain predictive elements should perform better than others that are purely reactive. A benchmark which evaluates elasticity helps to substantiate the intuition and to bring different strategies into an order.

Apart from these temporal characteristics of elasticity strategies many variants exist for different methodical *realization issues*, compare [ST09, p.13ff]. All of them have their own advantages and disadvantages. A benchmark can reveal their impact on elasticity.

2.4 Benchmark Requirements

Before creating a new benchmark, it is important to know about the properties of a good benchmark. The paper "The Art of Building a Good Benchmark" [Hup09] is one of the first approaches to capture the characteristics of a good benchmark. Huppler mentions relevance, repeatability, fairness, verifiablity and economic efficiency as main characteristics of a benchmark. In a later paper [Hup12], Huppler addresses some new challenges that occur when developing benchmarks for cloud systems.

These five characteristics of a benchmark can - although structured differently - also be found in the benchmark requirements defined by Folkerts et. al. in their paper "Benchmarking in the Cloud: What it Should, Can, and Cannot Be"[FAS+12]. Folkerts et. al. arranged their benchmark requirements according to three groups: general requirements, implementation requirements and workload requirements. The following paragraph uses the requirement structure proposed by Folkerts et. al. to explain benchmark requirements and mention eventual implications for an elasticity benchmark. This way, RQ 1.3 is answered.

1. General Requirements

   a) *Strong Target Audience*
   One precondition for the success of a benchmark is a target audience of a considerable size. In the cloud context, possible target audiences are cloud customers who want to use cloud services and cloud providers who want stand out of their competition. In the case of elasticity benchmarking, researchers represent another target audience as they need a tool for evaluating developed elasticity strategies.

   b) *Relevant*
   Generally, a benchmark should measure the performance of operations that are typical within the targeted domain. Elasticity benchmarking is not targeted to a narrow domain. It should rather be possible to benchmark elastic systems that are open for different domains. To achieve relevance, typical operations that stress elasticity - provisioning and deprovisioning - should be triggered.

   c) *Economical*
   Running the benchmark should be affordable. For the sake of relevance, an elasticity benchmark has to trigger provisioning or deprovisioning operations at different scales. This can be expensive when comparing different public cloud systems. However, for the evaluation of different elasticity strategies in research it may be sufficient to them on a private cloud or a cheap public cloud.

d) *Simple*
A benchmark with a highly complex structure is often difficult to understand and hard to trust. If people do not trust a benchmark, they will not use it. Benchmarks should therefore be as simple as possible. Necessary complexity can be explained in a benchmark documentation.

2. Implementation Requirements

a) *Fair and Portable*
Fairness is an intuitive property of any benchmark. However, this does not mean that fairness is easy to establish. A benchmark can ensure fairness either by taking care of certain properties of different systems or by limiting the participant systems in way that the remaining systems are evaluated fair. When elasticity is benchmarked, fairness is an important issue when comparing systems whose underlying resources have different levels of granularity or efficiency. Comparing elastic systems which use different scaling methods can also be difficult with respect to fairness.

b) *Repeatable*
Benchmark results should be reproducible. Without reproducibility it is difficult to create trust for a benchmark.

c) *Realistic and Comprehensive*
This requirement is similar to the requirement *relevant* and means that the typical features used in the major classes of target applications should be exercised.

d) *Configurable*
The workload used to exercise the benchmarked system should be configurable. This true in particular for elasticity benchmarks, as depending on the targeted domain, the type of needed elasticity can vary. In some domains, elasticity is necessary to compensate seasonal patterns, in others it is important to react properly to high variations due to short bursts.

3. Workload Requirements

a) *Representativeness*
Representative workloads are important as the system should be stressed in realistic way. Configurable workloads help to customize the benchmark in a way that fits to the targeted domain.

b) *Scalable*
Scalability of workloads should be supported by a benchmark. In the context of elasticity benchmarking, scalability plays an inherent important role.

c) *Metric*
The metric used for a benchmark should be meaningful and understandable. For elasticity benchmarking this means the metrics should preferably reflect the different aspects of elasticity in an easy-to-grasp manner.

Fulfilling all these requirements completely is hard since some them, i.e., simplicity and fairness, tend to conflict with each other. Nevertheless, having these requirements and the corresponding implications in mind is important when developing a new benchmark.

3. Related Work

This section analyzes existing approaches in the context of measuring and modeling elasticity and thus addresses the second goal mentioned in Section 1.1. The approaches are grouped according to their focus (compare RQ 2.1) and are analyzed with respect to their limitations (compare RQ 2.2).

3.1 Early Elasticity Measurement Ideas and Approaches

Binning et al. [BKKL09] present initial ideas for measures that capture different aspects of cloud systems. Although the authors do not use the term elasticity, one of the discussed aspects is related to it: The ability of a system to adapt to peak loads. Binning et al. suggest to measure the adaptability as the ratio between the number of requests that are answered within a given response time and the total number of issued requests. It stays unknown, if the peak was big enough to enforce an adaptation or if the peak was so big that even at the upper scaling bound the system is not able to handle the request within the response time. Still, this can be seen as an early approach for measuring elasticity based on response time variability.

Ang Li et al. [LYKZ10] introduce the *Scaling Latency* metric. It measures the time between a manual request of an resource instance and its availability for use. Ang Li et al. further split up the *Scaling Latency* into the time necessary to make the instance available and power it on (*Provisioning Latency*) and the time between powering it on and its availability for use (*Booting Latency*). These time spans are one aspect that influences the elasticity of a system. In addition, the scaling behavior strongly depends on the elasticity mechanism that triggers the creation or removal of instances. This influence cannot be measured with the *Scaling Latency* metric.

Zheng Li et al. [LOZC12] present a catalog of metrics for various cloud aspects. For elasticity, they identify the aforementioned *Scaling Latency* and additionally the *Resource Release Time* and a metric called *Cost and Time Effectiveness* as elasticity measures. The latter takes the granularity of resources into account. Li et al. argue that using small instances offers higher elasticity than using big ones because the customer is billed according to a fine grained resource usage. All three metrics measure static system properties. As discussed in the previous paragraph, the dynamic behavior of a System also influenced by other factors such as the ability of the elasticity mechanism to detect or foresee demand changes.

The SPEC Open Systems Group (OSG) [CCB+12] defines four elasticity metrics in their *Report on Cloud Computing*. The first metric, *Provisioning Interval*, is equal to the *Scaling Latency* metric mentioned above. With an *Agility* metric the SPEC OSG measures the sum of over- and provisioned resources, normalized with an quality of service dependent resource demand. The remaining two elasticity metrics *Scaleup/Down* and *ElasticSpeedup* measure scalability not elasticity. The first two metrics however, already capture the accuracy and the timing aspect of elasticity to some extend.

Herbst [Her11] proposes four elasticity metrics and demonstrates their use for analyzing the elasticity of thread pools. Elasticity is evaluated by measuring the reaction time between demand and corresponding supply changes, by analyzing the distribution of reconfiguration effects, by comparing the reconfiguration frequency of demand and supply and by evaluating the dynamic time warping (DTW) distance between the demand and the supply curve.
Herbst et al. [HKR13] extend those metrics with *speed* and *precision* as further elasticity metrics. Both metrics capture the scale up and the scale down behavior of a system separately. The scale up/down speed metric measures the average time to switch from an under-/over-provisioned state to an optimal or over-/under-provisioned state. The scale up/down precision metric measures the average amount of under-/over-provisioned resources during a measurement period.
Furthermore, Herbst et al. state the importance of not mixing up elasticity with other system properties like efficiency and scalability when comparing the elasticity of systems. They sketch the idea of inducing equal demand curves on systems with different scaling behaviors or different levels of efficiency of underlying resources in order to allow a fair elasticity comparison.
This presents a matured concept for benchmarking resource elasticity based on this idea and refines, extends, and evaluates the metrics.

Coutinho et al. [CGS13] propose based on the work of Herbst et al. [HKR13] metrics to support the analysis of elastic systems. Coutinho et al. use the term *underprovisioned state* to refer to the state of a system in that it is adding resources. The term *underprovisioned state* is used accordingly for the removal of resources. Additionally, a *stable state* is defined as a system state where instances are neither added nor removed. A further *transient state* is not clearly defined. The proposed metrics measure the time spent within these states and the amount of resources allocated within them. Sample metric values are computed for two experiments. The authors name the refinement and the interpretation of these metrics as future work. None of the provided metrics measures the accuracy of elastic behavior.

3.2 Elasticity Models and Simulating Elastic Behavior

Shawky and Ali [SA12] measure elasticity of clouds at the infrastructure level in analogy to the definition of elasticity in physics as the ratio of stress and strain. Hereby, stress is modeled by the ratio of required and allocated computing resources. Strain is modeled as the product of the relative change of the data transfer rate and the time required to scale up or down one resource. In simulated experiments the modeled elasticity decreases with the total number of VMs. No experiments for scaling down are presented.

Brebner [Bre12] presents an approach to model and predict the elasticity characteristics of cloud applications. The approach models the essential components of cloud platforms: The incoming load, the load balancer, the elasticity mechanism and the VMs together with a deployed application. The behavior of the cloud platform is simulated using a discrete event simulator in order to predict compliance with response time SLOs and

costs. In contrast to a classical benchmark, this approach predicts the behavior instead of measuring it.

Similarly to Brebner, Suleiman et al. [SV13] present analytic models that emulate the behavior of elasticity rules. The models allow to predict metrics such as CPU utilization or response time for given elasticity rules and a statistically modeled number of concurrent users. Since a model of the evaluated system is required, measuring the elasticity of systems with unknown elasticity mechanisms or other system internals is hardly possible.

Bersani et al. [BBD+14] formalize concepts and properties of elastic systems based on a temporal logic. The approach leverages the automatic verification whether proposed constraints hold during the execution of a workload. A benchmark in contrast, does not evaluate constraints that are either true or false but measures the quality of different elasticity aspects. Although the perspective of the approach of Bersani et al. is different from benchmarking, some of the constraints can be transformed into useful metrics. For example, a constraint that restricts the amount of under- or over-provisioned resources or one that limits an oscillating behavior can be transformed into metrics that measure the amount of under- or over-provisioned resources (compare the *accuracy* metrics, Section 5.1) or respectively into a metric that measures the frequency of oscillations (compare the *jitter* metric, Section 5.2.2).

3.3 Business Perspective Approaches

Many proposed approaches use a business perspective when evaluating elasticity. They measure elasticity indirectly by comparing the financial implications of alternative platforms or strategies. This may be a valid approach from a cloud customer perspective, but is often hard to implement because it is difficult to derive necessary cost or penalty functions. Furthermore, such approaches mix up the evaluation of the technical aspects of elasticity and the business model. Nevertheless, the following paragraphs explain some of the business oriented approaches for the sake of completeness.

Folkerts et al. [FAS+12] propose a simple cost oriented approach to evaluate the financial impact of elasticity. They suggest to measure elasticity by running a varying load and comparing the resulting price with the price for the full load. A reduced price for varying load is a rough indicator for elasticity, but it does not allow more detailed evaluation.

Weinman [Wei11] presents a metric very similar to the *Agility* metric of the SPEC OSG [CCB+12] and the *precision* metric of Herbst et al. [HKR13]. He compares the demand curve D(t) and the resource allocation curve A(t) for a computational resource with a loss function. The loss function measures the weighted sum of the financial losses for over- ($A(t) > D(t)$) and under-provisioning ($A(t) < D(t)$) periods. The paper also analyzes how different elasticity strategies influence the resulting loss. This approach evaluates the financial implications of the accuracy aspect of elasticity but does not evaluate the timing aspect explicitly.

Sharma et al. [SSSS11] present a concept for cost-aware resource provisioning. The approach accounts for infrastructure and transitioning costs and optimizes them using integer linear programming. However, the approach does not allow to compare different resource provisioning options with other metrics than infrastructure or transitioning costs.

Suleiman et al. [Sul12] propose a framework that allows to collect different cost and performance metrics and supports trade-off analysis. Initial results compare costs and the maximum latency for a simple step wise increasing load intensity. Of course, the maximum latency of requests is influenced by the elasticity of a system, but it cannot quantify the elasticity of a system alone.

Islam et al. [ILFL12] present a concept that allows cloud customers to evaluate the financial implications of choosing different elastic cloud providers. In contrast to many other works, this paper analyzes over- and under-provisioning. It also considers the fact, that the amount of allocated resources is not necessarily equal to the amount of resources the customer is charged for. Besides the costs for resource allocations, Islam et al. take the penalty costs for violating SLAs into account. The load profiles used for the evaluation is a set of simple mathematical functions including linear functions, exponential functions and sines containing plateaus of different lengths. These load profiles are one step towards a realistic variation of load intensity, but still the use of a workload model that captures the expected variability of load intensity better may be desired.

Moldovan et al. [MCTD13] propose MELA, a framework targeted at cloud service providers that allows to analyze the elasticity dimensions resource elasticity, cost elasticity and quality elasticity. The framework monitors low level data for every dimension and offers mechanisms to compose the monitored data to higher level metrics. For a set of metrics the framework can analyze the boundaries between that the metric values vary during the measurement. Additionally, relationships between metrics can be discovered by analyzing the rate of different metric value combination occurrences. The proposed framework is a generic monitoring tool and allows cloud providers to analyze different financial elasticity aspects. Currently, the framework does not allow to retrieve or monitor the resource demand as required for a technical analysis of resource elasticity.

3.4 Elasticity of Cloud Databases

Dory et al. [DMRT11] propose an approach to measure the elasticity for cloud databases. They analyze elasticity by measuring how a cluster of database nodes reacts after adding new nodes. The quality of the behavior is measured using the observed distribution of response times after triggering the scale up. The removal of database nodes as well as the influence of an elasticity mechanism that triggers the adaptations is not analyzed.

Almeida et. al. [ASLM13] present another response time based elasticity measurement methodology for cloud databases. Over-provisioning as well as under-provisioning are evaluated within the approach. For the over-provisioning case, the ratio of expected and actual response time is used to determine the degree of elasticity. Implicitly, this assumes that adding more resources will always result in a decreasing response time. For the most systems, this assumption does not hold for a low utilization of the underlying resources. Without this assumption, the approach may be able to evaluate if a system over-provisions, but it cannot quantify how much over-provisioning is occurring.

Tinnefeld et al. [TTP14] propose an approach to evaluate the elasticity of cloud database management systems by analyzing the financial implications of using a certain system. This approach bases on and is very similar to the approach of Islam et al. [ILFL12] discussed in Section 3.3.

3.5 Conclusions

Existing elasticity measurement approaches analyze elasticity only to a limited extend. Their metrics often cover only the elasticity aspect timing but not the accuracy aspect or vice versa. Many approaches evaluate the elastic behavior in scale up or scale out scenarios, but do not consider scenarios where resources are decreased. Approaches that analyze both behaviors often use simple workload models, where the load intensity is varied according to simple mathematical functions. For benchmarking purposes the usage of representative workloads is desirable [FAS+12]. All analyzed approaches but

[HKR13] neglect to take the levels of efficiency of underlying resources and the scaling behavior of a system explicitly into account in order to not mix up these properties with elasticity. Business perspective analysis approaches are important for customers who are interested in the financial implications of choosing between different cloud offerings. These approaches are often difficult to implement and mix up the evaluation of the technical property elasticity and of the business model of the cloud provider.

4. Resource Elasticity Benchmark Concept

This chapter addresses Goal 3 by explaining the benchmarking concept that was further developed and refined in course of this thesis based on previous research [Her11]. Section 4.1 outlines the scope and explains the limitations of the benchmarking approach. A general overview about the benchmark components and about the benchmarking workflow is then given in Section 4.2. The conceptual ideas for the essential benchmark components are discussed in own sections. The implementation of the concept is illustrated in Chapter 6.

4.1 Limitations of Scope

This thesis adopts a technical perspective on resource elasticity. Therefore, the developed benchmark targets researchers, cloud providers and customers interested in comparing elastic systems from a technical, not a business value perspective. As a result, this approach does not take into account the business model of a provider or the concrete financial implications of choosing between different cloud providers, elasticity strategies or strategy configurations.

Since this approach evaluates resource elasticity from a technical perspective, a strict black-box view of the CSUT is not sufficient. The evaluation bases on comparing the induced resource demand with the actual amount of used resources. To monitor the latter, access to the CSUT is required. Furthermore, the calibration requires to manually scale the amount of allocated resources. Since cloud providers usually offer APIs that allow resource monitoring and manual resource scaling, this limitation does not restrict the applicability of the benchmark.

Cloud services offer their customers different abstraction layers. These layers are commonly referred to as Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS) [MG11]. As this thesis focuses on resource elasticity, the target systems for elasticity comparisons are mainly systems that provide IaaS. In the SaaS context resources are not visible to the user. The user pays per usage quota instead of paying for allocated resources. SaaS systems are therefore not within scope of this thesis. Although this approach is not explicitly targeted at PaaS, the approached benchmark should also be applicable in the PaaS context as long as the underlying resource are transparent.

The workloads used for this approach are realistic with respect to load intensity. They are modeled as open workloads with uniform requests. The work units are designed

to specifically stress the scaled resources. Workloads that use a mixture of work unit sizes, stress a several (scalable) resources types at the same time or workloads modeled as closed workloads remain future work. The application that uses the resources is assumed to be stateless. Thus, a requests always consumes the same amount of resources. Furthermore, selecting appropriate load profiles is not in scope of this thesis. However, the thesis demonstrates how complex realistic load profiles can be modeled and adjusted in a system specific manner in order to allow fair comparisons of systems with different levels of efficiency of underlying resources and different scaling behaviors.

The range of different resources that a resource elastic system can scale is broad. This thesis focuses on processing resources such as CPUs but can also be applied to other physical resources. The evaluation will showcase a simple IaaS scenario where VMs are bound to processing resources. Thus, scaling the virtual machines corresponds to scaling the processing resources. Elasticity of resources on a higher level of abstraction, like thread pools, have been analyzed before [KHvKR11] and are not in scope of this thesis.

Elastic systems can scale resources horizontally, vertically or even combine both methods to match the resource demand. This thesis focuses on comparing systems that scale VMs horizontally.

4.2 Benchmark Overview

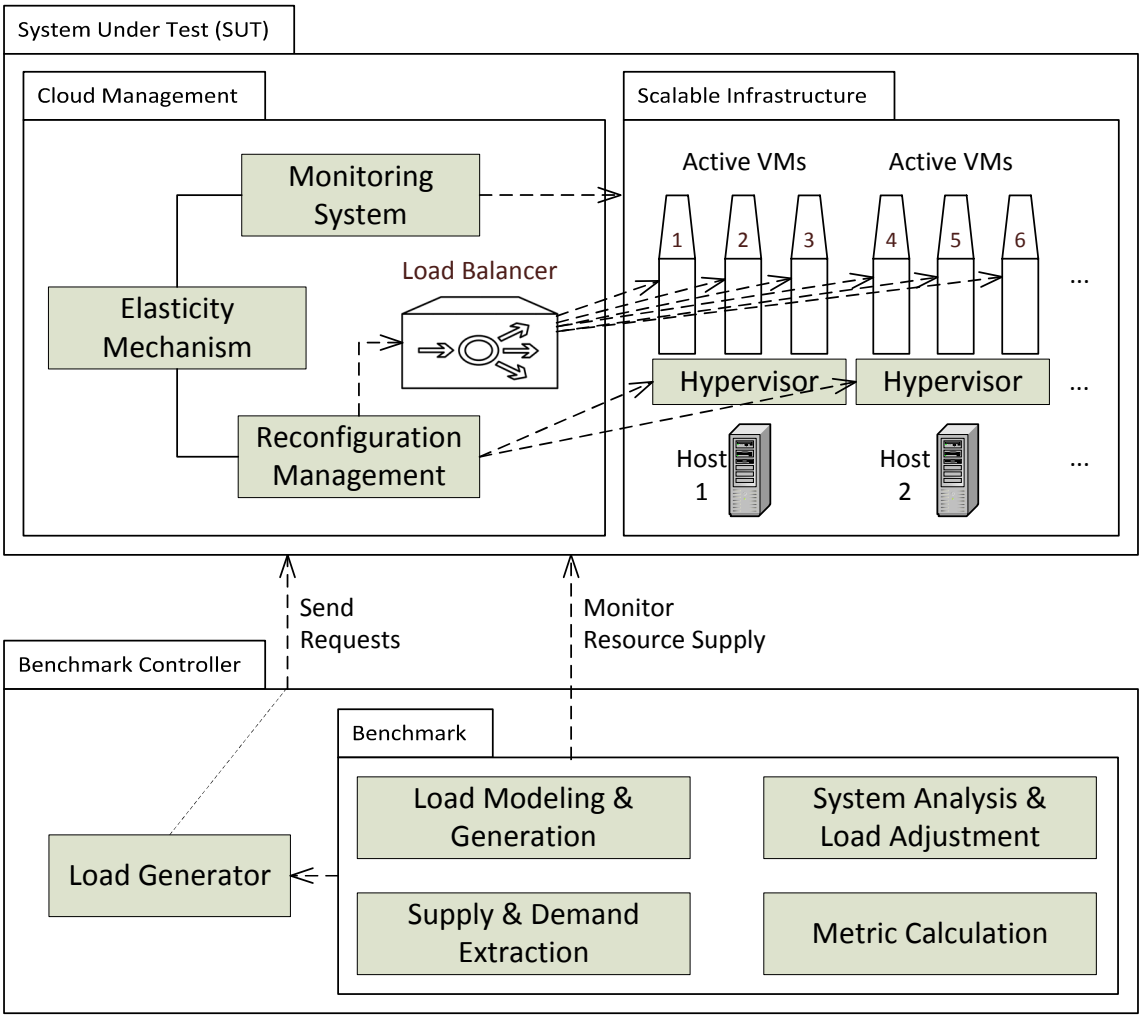


Figure 4.1: Blueprint for the CSUT and the benchmark controller

This section presents the structure of the benchmark concept. Figure 4.1 shows an extended version of the cloud architecture blueprint that was presented in Chapter 2.4. The extended version additionally contains the *benchmark controller*, which runs the benchmark. The benchmark components facilitate the process for benchmarking resource elasticity that is depicted in Figure 4.2.

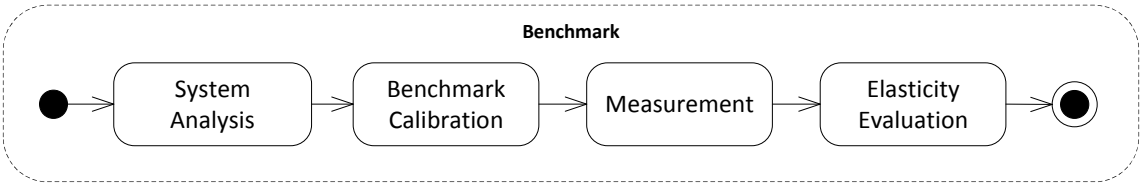


Figure 4.2: Activity diagram for the benchmark work flow

The benchmarking process comprises four activities:

1. **System Analysis**
   The benchmark analyzes the CSUT with respect to the efficiency of underlying resources and its scaling behavior.
2. **Benchmark Calibration**
   The analysis result is used to adjust a load intensity profile in a way that it induces the same resource demand on all compared systems.
3. **Measurement**
   The load generator exposes the CSUT to a load varying according to the adjusted load profile. The benchmark extracts the induced resource demand as well as the actual resource allocations (resource supply) on the CSUT.
4. **Elasticity Evaluation**
   Metrics compare the curves for resource demand and resource supply with respect to different elasticity aspects.

The remainder of this chapter explains the benchmark components according to the following structure: Section 4.3 explains how workloads can be modeled and executed. Section 4.4 explains why analyzing the evaluated system and calibrating the benchmark accordingly is necessary and describes the concept for realizing both activities. Finally, Section 4.5 explains how the resource demand curve and the resource supply curve can be extracted during the measurement.

4.3 Workload Modeling and Generation

This section covers modeling and executing workloads suitable for elasticity benchmarking and thereby addresses RQ 3.1.

4.3.1 Worktype

A benchmark should stress the CSUT in representative way. Therefore, a benchmark which measures the performance of a system for example should execute a representative mix of different programs to stress the system. An elasticity benchmark however measures how a system reacts when the demand for specific resources changes. Thus, an elasticity benchmark must induce representative demand changes.

Varying demand is mainly caused by a varying load intensity. An elasticity benchmark should therefore vary load intensity in a representative way. Section 4.3.2 illustrates how the variation of load intensity is modeled in this approach.

In order to induce a processing demand, the work which is executed within each request is designed to be CPU-bound. In particular, for every request a *fibonacci numer* is calculated. To minimize the memory consumption, an iterative algorithm is used in lieu of a recursive one. Result caching is avoided by adding random numbers within each calculation step. Furthermore, the final result is returned as part of the response to prevent compiler optimizations that remove the whole execution.

The overhead for receiving requests is limited by using a lightweight web server. More details about how requests are handled and processed on the server side can be found in Section 6.2.

4.3.2 Load Profile Modeling

A good benchmark uses realistic load profiles to stress the CSUT in a representative manner. Workloads are commonly modeled either as closed workloads or as open workloads [SWHB06]. Whereas in closed workloads new job arrivals are triggered by job completions, arrivals in open workloads are independent of job completions. The elastic behavior of a system is usually triggered by a change in load intensity. Hence, for elasticity benchmarking it is important that the variability of the load intensity is modeled realistically. As this can be achieved with an open workload model, the developed benchmark will use an open workload model. Unnecessary complexity due to a closed workload model is avoided.

Workloads typically consist of a mixture of several patterns. These patterns can model linear trends, bursts that are characterized by a exponential increase, or patterns which model the general variability over a day, a week or a year. V. Kistowski et al. present in [vKHK14a] a meta-model that allows modeling of varying load intensity behaviors. They also offer the LIMBO toolkit described in [vKHK14b] to facilitate the creation of new load profiles that are either similar to existing load traces or contain different desired properties like a seasonal pattern and additional bursts. The usage of this toolkit and the underlying meta-model allows the creation of realistic load variations that are still configurable. Thus, the load profiles used for benchmarking can be adapted with low effort to suit the targeted domain.

4.3.3 Load Generation

In order to stress an elastic system reproducible, it is necessary to send accurately timed requests to the tested system. This subsection illustrates the concepts for the parallel submission of requests and shows how the timing accuracy of the request transmission can be evaluated. The implementation of the load driver is described in Section 6.1.3.

4.3.3.1 Parallel Submission and Response Handling

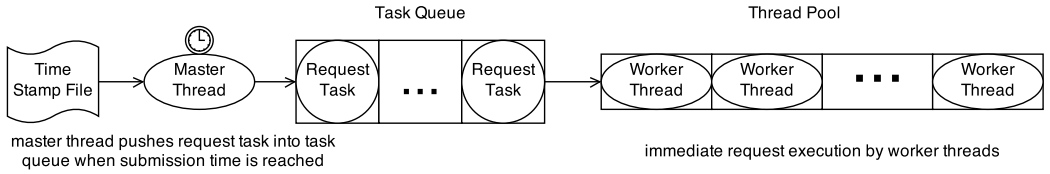
Partitioning Techniques

Depending on the request response time, the handling (sending of a request and waiting for the corresponding response) of consecutive requests overlaps and must therefore be done concurrently. Three different strategies which allow share the work of request submission and handling of the answers between threads have been developed in course of this thesis. One of them bases on static partitioning, two on dynamic partitioning.

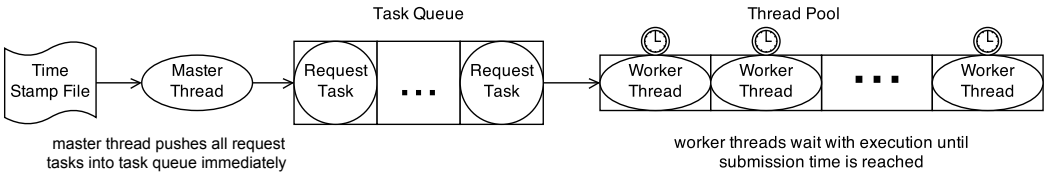
1. Static Partitioning - Round Robin
   Timestamps are assigned to the threads in a round robin approach. Every thread has its own list of timestamps which it processes one after another. For every timestamp, the thread first sleeps until the time of submission specified by the timestamp is

reached. Then, the thread sends a request and waits for the corresponding response. If a response is received delayed, the next request cannot be send in time anymore, although other threads may idle at the same time. However, a timeout which limits the maximum response time and a sufficient number of threads can solve this problem.

2. Dynamic Partitioning - Thread Pool Pattern
   The dynamic partitioning approaches base both on the thread pool pattern, which is also known as the replicated worker pattern [FHA99]. In the thread pool pattern, a number of threads performs a set of tasks concurrently. The tasks are typically produced by a master thread, who puts them into a data structure, such as a synchronized queue. Whenever a thread has completed a task, it takes a new one from the queue. If the queue does not contain any tasks, the threads wait until a new task is inserted into the queue. The two options explained in the following mainly differ in when tasks are added to the task queue. They are both illustrated in Figure 4.3.



(a) Waiting Master Thread



(b) Waiting Worker Threads

Figure 4.3: Alternative ways for using the *Thread Pool Pattern*

a) Waiting Master Thread
   The master first reads the list of timestamps. Then, the master waits until the submission time for the first request is reached. It pushes the task of sending this request into the queue. One of the free worker threads takes the task from the queue and immediately executes it. In the mean time, the master thread waits until the submission time for the next request is reached and again pushes it into the queue. Again, one of the free worker threads takes the task from the queue and executes it. This procedure is repeated for all timestamps.

b) Waiting Worker Threads
   Waiting for the time of submission is shifted from the master thread to the worker threads. The master thread pushes submission tasks into the task queue immediately after reading the timestamps. As in the first variant, the worker threads take tasks from the queue as soon as they are finished with the previous task. In contrast to the *waiting master thread* variant, now the worker threads wait until the point of submission for the request contained in their specific task is reached. When the submission time is reached, the request is sent. After the response has been received, the thread takes the next task from the queue.

Scale Request Submission across Hardware Nodes

For high load intensities, the capabilities of one hardware node may not be sufficient to send all requests in time. In this case, request submission must be scaled out. This means request are sent from different machines (nodes). One way to do so is using static partitioning to assign requests to the different nodes. On each node, the requests can then be sent concurrently using a set of threads and one of the above mentioned partitioning techniques.

Required Number of Parallel Threads

When the maximum response time is known - this is the case when a timeout is defined for the requests - the amount of threads which is necessary to handle all request without delays due to a lack of parallelism can be computed as follows:

Algorithm 1 Calculate Number of Threads

```
function numberOfThreads(timestamps,timeout)
    requiredThreads ← 0
    concurrentRequests ← {}
    for all timestamp ∈ timestamps do
        add timestamp to concurrentRequests
        while size(concurrentRequests) > 0 and
                concurrentRequests.first + timeout < timestamp do
            removeFirstElement(concurrentRequests)
        end while
        currentRequiredThreads ← size(concurrentRequests)
        requiredThreads ← max(requiredThreads,currentRequiredThreads)
    end for
    return requiredThreads
end function
```

These threads can either be located on one node or be split up across several nodes.

4.3.3.2 Request Transmission Accuracy Evaluation

Sending requests in time is a crucial aspect to ensure that resource demands are induced correct and repeatable over different runs. Evaluating the accuracy of the request submission times is therefore desirable.

A simple approach for this evaluation bases on comparing the planned submission times of the requests with the real ones. Therefore, real submission times have to be logged by the load driver which executes the request. Unfortunately, the order of the logged timestamps is not necessarily equal to the order of planned timestamps. In particular, the logged timestamps can be disordered when the record for the request is written only after the response has been received.

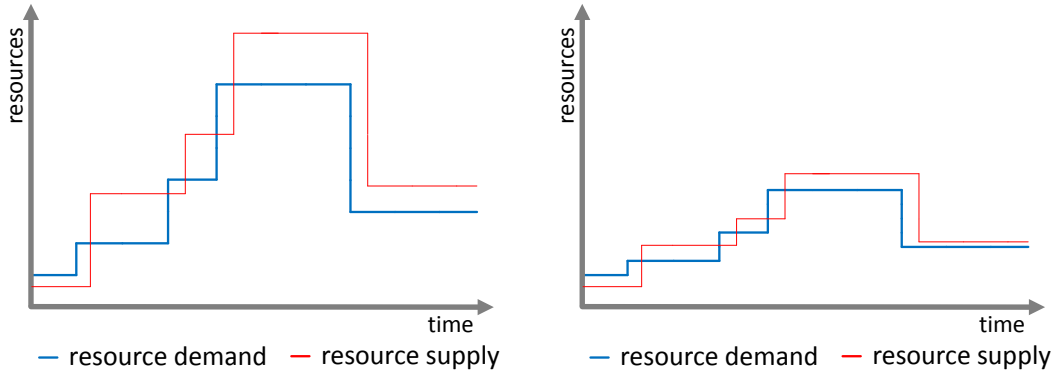
Thus, it is necessary to match the logged real submission times with the correct planned submission times. This can be achieved by assigning a unique identificator (UID) to every timestamp. When this UID is logged together with the submission time, it is easy to match the real and planned submission times.

If planned and real time of submission are known for all requests, simple statistical measures such as sample mean $\overline{X}$ and sample standard deviation S can be calculated for difference of planned and real submission times. These measures allow insights into

different aspects: $\overline{X}$ specifies how much requests are delayed on average. S specifies if the observed delay is rather constant or is varying. For accurately send requests both measures should be close to zero. Since especially $\overline{X}$ is highly affected by outliers another option for measuring the submission accuracy is evaluating the p percentile, e.g., the 95th percentile, of the difference between planned and real submission times.

4.4 Analysis and Calibration

The resource demand of a system for a fixed load intensity depends on two factors: The efficiency of a single underlying resource unit and the overhead caused by combining multiple resource units. Both aspects can vary from system to system and define distinct properties namely efficiency and scalability. Elasticity is a different property and should be measured separately. One way to do so relies on analyzing the load processing capabilities of a system before evaluating elasticity. After such an analysis, it is known how many resources a system needs to satisfy a given static resource demand. The resource demand can then be expressed as function of the load intensity: $resourceDemand = demand(intensity)$. With the help of this mapping function, which is specific for every system, it is possible to compare the resource demand to the amount of the actually allocated resources. In the following section RQ 3.2 is answered by explaining how the mapping function can be derived.



(a) System A (less efficient resources)

(b) System B (more efficient resources)

Figure 4.4: Different resource demands for equal load profiles

Figure 4.4 shows how two systems react when they are exposed to the same load profile. Since the resources of System B are more efficient than those of System A, System B can handle the load with less resources than System A. For both systems there exist some points in time where the systems over-provision and other points in time where they under-provision. Comparing their elasticity is difficult as the resource demands of the systems are different.

The idea of this approach is to adjust a given load profile in a way that the induced resource demand changes occur equal in amount and experiment time on all systems. By doing so, the possibly different levels of efficiency of underlying resources as well as different scaling behaviors are compensated. With an equal resource demand it is now easier to compare the quality of the adaptation process and thus the evaluation of elasticity can be done in a fair way. This adjustment is explained in detail in Section 4.4.2, which thereby answers RQ 3.3.

4.4.1 System Analysis

The goal of the *System Analysis* is to derive a function that maps a given load intensity to the corresponding resource demand. Hereby, the resource demand is the minimum resource amount that is necessary to handle the load intensity without violating a set given SLO. Therefore, the analysis assumes predefined SLOs. They have to be chosen according to the targeted domain.

Since this *System Analysis* should not evaluate any elastic behavior, scaling is controlled manually during the analysis.

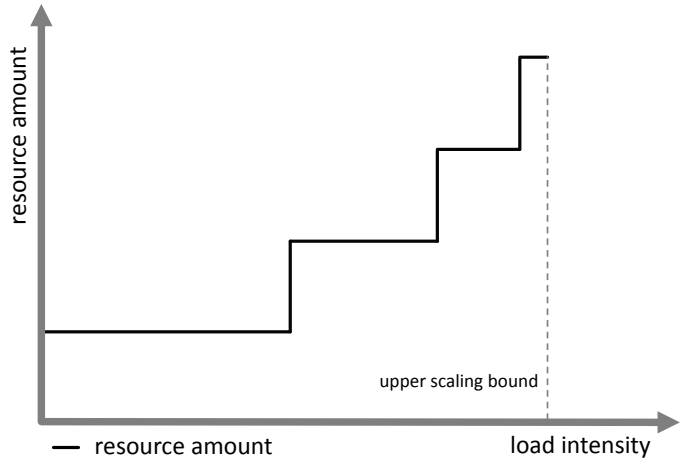


Figure 4.5: The result of a *System Analysis*: The mapping function *demand*(*intensity*)

The result of an hypothetical analysis is shown in figure 4.5. The dashed gray line marks the upper scaling bound and thus the maximal load intensity the system can cope without violating SLOs. This upper bound is either caused by a limited amount of available resources or by a limited scalability due to other reasons like a limited bandwidth or increased overhead. In the latter case, additional resources are available, but even after adding resources the SLO cannot be satisfied.

Figure 4.5 shows that the derived mapping function *demand*(*intensity*) is a step function. The function is characterized by the intensities where the resource demand increases. Thus, analyzing the system means finding those intensities. The search is realized within an iterative process that is formalized in Algorithm 2.

The analysis is starts by configuring the CSUT to use one resource instance. This includes configuring the load balancer to forward all requests to this instance. Now, the maximum load intensity that the system can withstand without violating the SLO has to be determined, because this the load intensity at which the resource demand increases. In this thesis, the maximum load intensity is also referred to as the load processing capability.

Several load picking algorithms that are able to find the load processing capability are discussed in [SMC+08]. For this approach a binary search algorithm is applied, see Algorithm 3. Binary search consumes more time than a model guided search but is, in contrast to the latter, guaranteed to converge and still fast compared to a simple linear search.

Since the upper and lower search bounds are not known at the beginning, the binary search is preceded by an exponential increase/decrease of the load intensity to find those bounds. As soon as both bounds are known a regular binary search is applied.

Algorithm 2 Detailed System Analysis

```
function analyzeSystem(startIntensity)
    disableAutoScaling( )
    resourceAmount ← 1
    configureSystemAndLoadBalancer(resourceAmount)
    lastIntensity ← 0
    finished ← false
    while (!finished) do
        maxIntensity ← searchMaxIntensity(startIntensity)
        if (maxIntensity ≤ lastIntensity) then
            finished ← true
        else
            addToMappingFunction(resourceAmount, maxIntensity)
            if (furtherInstanceAvailable) then
                resourceAmount ← resourceAmount + 1
                configureSystemAndLoadBalancer(resourceAmount)
                lastIntensity ← maxIntensity
                startIntensity ← maxIntensity
            else
                finished ← true
            end if
        end if
    end while
    enableAutoScaling( )
    return function: demand(intensity)
end function
```

Algorithm 3 Binary Search for Maximum Intensity

```
function searchMaxIntensity(startIntensity)
    intensity ← startIntensity
    foundLower ← false
    foundUpper ← false
    while (!foundLower || !foundUpper || (upper − lower) < threshold) do
        comliant ← checkCompliance(intensity)
        if (comliant) then
            lower ← intensity
            foundLower ← true
            if (!foundUpper) then
                intensity ← intensity ∗ 2
            end if
        else
            upper ← intensity
            foundUpper ← true
            if (!foundLower) then
                intensity ← intensity/2
            end if
        end if
        if (foundLower && foundUpper) then
            intensity ← (lower + upper)/2
        end if
    end while
    return intensity
end function
```

Once the load processing capability for one resource is known, the system is reconfigured to use an additional resource unit and the load balancer is reconfigured accordingly. After the reconfiguration, the system should be able to comply with the SLOs again. The load processing capability is again searched with algorithm 3. This process is repeated until either there are no additional resources that can be added to the system, or even after adding a new resource the load processing capability does not increase further. In both cases the upper scaling bound is reached and the analysis is finished.

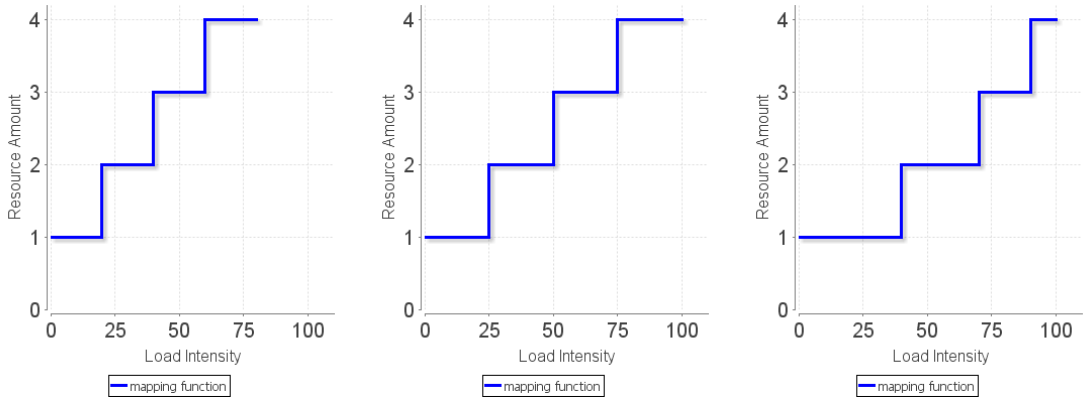
Simple and Detailed System Analysis

The analysis approach described above evaluates all scaling stages of a CSUT separately. When it is known that the resource demand of the analyzed CSUT is increasing linearly with the load intensity, the analysis can be simplified. In this case, it is sufficient to analyze the load processing capabilities for the first scaling stage only. The load processing capabilities for other scaling stages can then be extrapolated. For this thesis, analyzing the system at each scaling stage separately is referred to as *Detailed System Analysis*. In contrast, the simplified version is referred to as *Simple System Analysis*.

4.4.2 Benchmark Calibration

The goal of the *Benchmark Calibration* activity is to induce the same demand changes at the same points in time on every compared system. To do so, the load intensity curve has to be adapted for every system to overcome different levels of efficiency of underlying resources and different scaling behaviors.

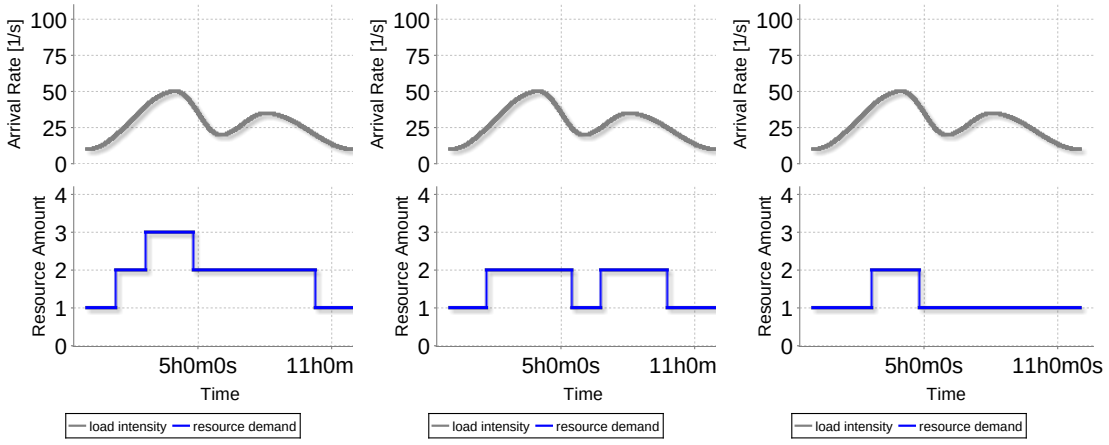
The mapping function *demand*(*intensity*) that was derived in the previous *System Analysis* activity, contains information about both: The efficiency of underlying resources and the scaling behavior. Figure 4.6 illustrates what impact different levels of efficiency and different scaling behaviors have on the mapping function. Compared to System A, the underlying resources of System B are more efficient. Hence, the length of steps of the mapping function is bigger for System B than for System A. For both systems there is no overhead when adding further resources. The resource demand is increasing linearly with the load intensity and the length of the steps for one system is therefore independent of the load intensity. For System C in contrast, the overhead increases when additional resources are used. As a result of this non linear increasing resource demand, the length of the steps of the mapping function decreases for increasing load intensities.



(a) System A (b) System B (c) System C

Figure 4.6: Different mapping functions

As illustrated in Figure 4.6, the resource demand variations on the Systems A-C are different when they are exposed to the same load profile, although they offer same amount of resources. The difference is caused by different scaling behaviors and different levels of efficiency of the underlying resources.



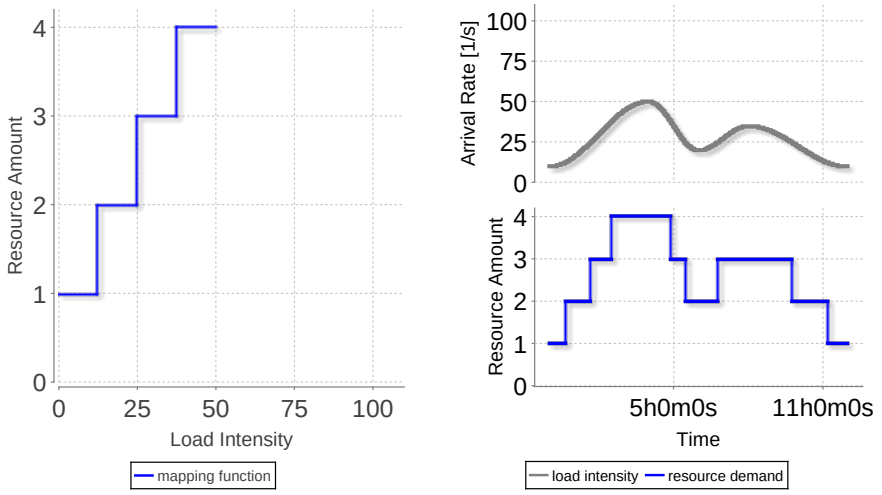
(a) System A (b) System B (c) System C

Figure 4.7: Resource demand induced by an unadjusted load profile

As basis for the transformation of load profiles, an artificial baseline system is used. The baseline system serves as reference system for demand changes. Thus, the resource demand on the compared systems is the same as of the baseline system when the resource

demand of the baseline system is induced by an unchanged load profile and the demand on the compared systems is induced by the respective transformed load profiles.

The resource demand of the baseline system is assumed to increase linearly with load intensity. Thus, the length of the steps of the mapping function is equal. Using this assumption, the mapping function $demand_{base}(intensity)$ of the baseline system can be characterized by two parameters: The number of steps n_{base} in the mapping function, which equals the assumed number of available resources, and the maximum load intensity $maxintensity_{base}$ that the base system can withstand using all resources. The first parameter is chosen as the maximum amount of resources that all systems support. The second parameter should be greater than or equal to the maximum load intensity which occurs in the load profile.



(a) Mapping function $demand_{base}(intensity)$ (b) Induced resource demand

Figure 4.8: Mapping function and resource demand on a baseline system for $n_{base} = 4$ and $maxintensity_{base} = 50$

Having defined the mapping function $demand_{base}(intensity)$ (compare Figure 4.8(a)), the load profile adjustment step finds for every benchmarked system k an adjustment function $adjustedintensity_k(intensity)$ such that the following applies:

$$\forall intensity \in [0, maxIntensity_{base}]:$$
$$demand_{base}(intensity) = demand_k(adjustedintensity_k(intensity))$$

The adjustment function $adjustedintensity_k(intensity)$ maps the steps from the $demand_{base}$-$(intensity)$ function to steps of the $demand_k(intensity)$ function. The result is a piecewise linear function, whereby every linear section represents the mapping of one step in the $demand_{base}(intensity)$ to the same step in the $demand_k(intensity)$ function. The parameters m_i and b_i which define the linear function $y_{k_i}(x) = m_i * x + b_i$ that maps the intensities belonging to the ith step of the $demand_{base}(intensity)$ function to the ith step of the $demand_k(intensity)$ function can be calculated as follows:

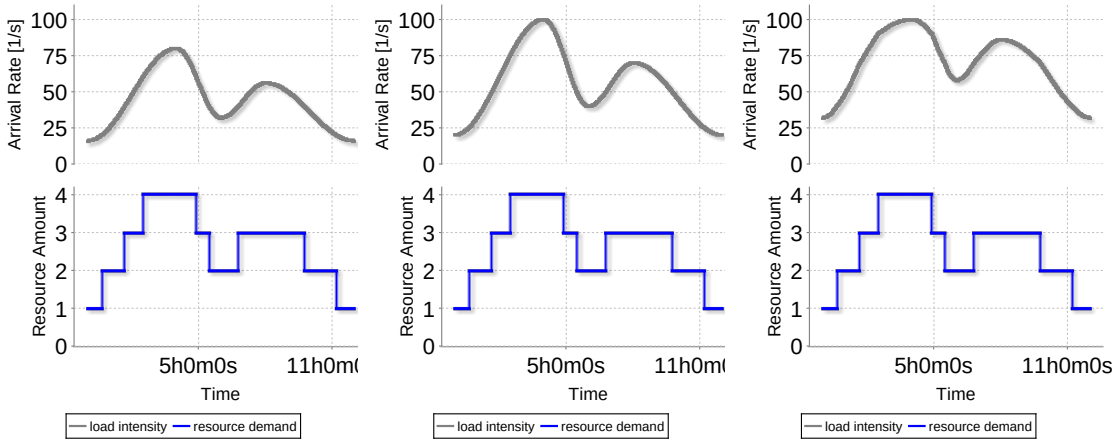
$$\begin{aligned}
startintensity_{k_i} &= \max\{intensity | demand_k(intensity) < i\} \\
endintensity_{k_i} &= \max\{intensity | demand_k(intensity) = i\} \\
steplength &= endIntensity - startIntensity \\
m_{k_i} &= steplength/(maxintensity_{base}/n_{base}) \\
b_{k_i} &= startintensity - steplength * (i-1)
\end{aligned}$$

and thus the function $adjustedintensity_k(intensity)$ can be expressed as:

$$adjustedintensity_k(x) = \begin{cases} y_{k_1} & \text{when } startintensity_{k_1} < x \leq endintensity_{k_1} \\ \vdots & \\ y_{k_{n_k}} & \text{when } startintensity_{k_{n_k}} < x \leq endintensity_{k_{n_k}} \end{cases} \tag{4.1}$$

Having calculated the adjustment function $adjustedintensity_k(intensity)$, this function can be applied onto the original load profile $intensity(t) = lp(t)$ in order to retrieve a system specific adjusted load profile $lp_k(t)$. This adjusted load profile can then be used in the actual benchmark run.

$$lp_k(t) = adjustedintensity_k(lp(t))$$



(a) System A (b) System B (c) System C

Figure 4.9: Induced resource demand for system specific adjusted load profiles

Figure 4.9 shows the induced load demand for Systems A, B and C using the adjusted load profiles. It can be seen that although the systems have underlying resources with different levels of efficiency and have different scaling behaviors, the induced resource demand variations are now equal for all compared systems.

4.5 Measurement: Demand and Supply Extraction

During a measurement run, the load driver induces a varying resource demand on the CSUT. At the same time, the CSUT tries to match this demand by adapting the amount of allocated resources, the resource supply, dynamically. This section describes how the resource demand and the resource supply can be extracted and thereby answers RQ 3.4 and RQ 3.5, respectively.

4.5.1 Resource Demand

The load on the CSUT is induced by requests which are sent according to a predefined load profile. The load profile $lp(t)$ defines the load intensity, i.e., the number of requests sent per time unit, over time. Using the mapping function $demand(intensity)$ derived in the *System Analysis* (compare Section 4.4.1), this information can be used to derive the resource demand. For every point in time t, the induced demand $resourceDemand(t)$ can be calculated as follows:

$$resourceDemand(t) = demand(lp(t))$$

4.5.2 Resource Supply

Measuring the Amount of Usable Resources

The resource supply is the amount of resources which is available in a usable state. Unfortunately, the resource supply is not necessarily the resource amount that a cloud provider bills its customer. There are two reasons for the discrepancy. The main reason is that cloud providers usually charge based on time intervals which are coarse grained compared to the provisioning time. For example, for a resource that is used for ten minutes only a provider may bills the same costs as for a resource used for one hour. The second reason is that a resource may not be ready to use immediately after the allocation. This is especially true for container resources like VMs. After allocation, a VM needs time to boot and start required applications. Cloud providers may even bill the VM when its creation was scheduled, although the customer cannot use it at this point. This approach does not evaluate the business model of cloud providers. Therefore, the benchmark compares the amount usable resources, the resource supply, with the amount of necessary resources, the resource demand, independently of what the customer is billed for. The benchmark user should be aware of this fact when implementing the resource supply monitoring.

For this thesis and for the presented benchmark, the amount of allocated or the resource supply refers to the amount of resource that are *currently available for use*.

Elastic cloud systems are usually managed by a cloud computing software which also provides monitoring tools. These monitoring tools can be used to retrieve the resource supply. Hereby can, depending on the capabilities of the monitoring tools, two different monitoring techniques be applied: Active Monitoring based on polling and passive monitoring based on log parsing.

Active Monitoring - Polling the System State

This method polls the information about the resource supply periodically. The granularity of this method depends on the frequency of the polling requests. The disadvantage of this method is the creation of unnecessary network traffic and CPU load on the benchmark and on the management node.

Passive Monitoring - Log Parsing

If the monitoring tool offers access to logs which contain information about when resources have been de-/allocated, this log can be parsed to obtain the desired information. Log parsing usually only offers information about relative changes of the resource supply not about its absolute level. Thus, it is necessary to poll the absolute the number of used resources additionally, once. Using this information about the absolute level of the resource supply, the information from the parsed log can be used to compute the resource supply for other points in time. If this method is feasible, it should be preferred over the active monitoring, since it requires less overhead.

5. Resource Elasticity Metrics

The last activity within the benchmarking process, the *Elasticity Evaluation*, measures the observed elastic behavior of the CSUT with the help of metrics. This section discusses the elasticity metrics that have been developed and evaluated in course of this thesis and thus addresses the fourth goal mentioned in Section 1.1.

Some previous works [LYKZ10, LOZC12, CCB+12] suggest to measure static system properties that influence the elastic behavior. A drawback of this approach is that these static properties cannot fully explain elasticity. One example for such a system property is the *Scaling Latency*, defined as the time necessary to make a new resource available after it has been requested. Knowing this time can be useful for understanding imperfect elastic behavior in some cases. With a reactive elasticity strategy in place for example, an increased *Scaling Latency* will most likely lead to a decreased elasticity. In contrast, with a proactive elasticity strategy in place, the elasticity can be more or less independent of the increased *Scaling Latency* since the strategy can incorporate a known *Scaling Latency* into the decision making process.

This thesis presents metrics that do not evaluate system specific factors, such as the underlying hardware, the virtualization technology, the used cloud management software or the used elasticity strategy and its configuration. Instead, the metrics evaluate the elastic behavior that is a result of these factors. As a consequence, this measurement methodology is applicable in situations where not all influencing factors are known, too.

All metrics use two curves as input: The demand curve, which defines how the resource demand varies during the measurement period, and the supply curve, which defines how the amount of actually used resource varies.

The development of the metrics pursued three main targets:

1. Metrics should reflect the core aspects - accuracy and timing - of elasticity illustrated in Section 2.3.3.
2. Metrics should measure criteria that are as understandable as possible and help the communication between cloud providers and customers.
3. Limit the amount of metrics to as few metrics as are necessary in order measure elastic behavior. A new metric is only necessary, when it is possible to create two systems behaviors that result in equal metric values for the existing metrics but still exhibit different degrees of elasticity.

The next two sections present metrics for the core aspects of elasticity and thus address the RQ 4.1. They are followed by Section 5.3 that discusses metrics which were considered when creating the benchmark but were finally rejected. The last Section 5.4 answers RQ 4.2 by illustrating how the elasticity can be compared within a group of cloud systems using the discussed metrics as a basis.

5.1 Accuracy

The first aspect of elasticity that was discussed in Section 2.3.3 is accuracy. There are two ways how the resource supply can deviate from the resource demand. The system can either provision to much resources - in an over-provisioned state - or provision not enough resources - in an under-provisioned state. Both deviations have a negative impact on the accuracy, but depending on the preferences of the cloud customer, he may want to weight the inaccuracy caused by under-provisioning different to inaccuracy caused by over-provisioning. Since under-provisioning means violating SLOs, it can be assumed that customers do want to use systems that are susceptible to under-provisioning at all. Thus, the challenge for providers is to ensure that enough resources are provided at any point in time, but at the same time beat the competitors by over-provisioning not to much. Considering this, separate accuracy measures for over-provisioning and under-provisioning help providers to communicate their elasticity capabilities and help customers to select a cloud provider according to their needs.

Herbst et al. proposed [HKR13] two metrics that capture the accuracy for over-provisioning periods and for under-provisioning periods separately. Since these metrics capture the core aspect accuracy very well, they are implemented and evaluated within this thesis.

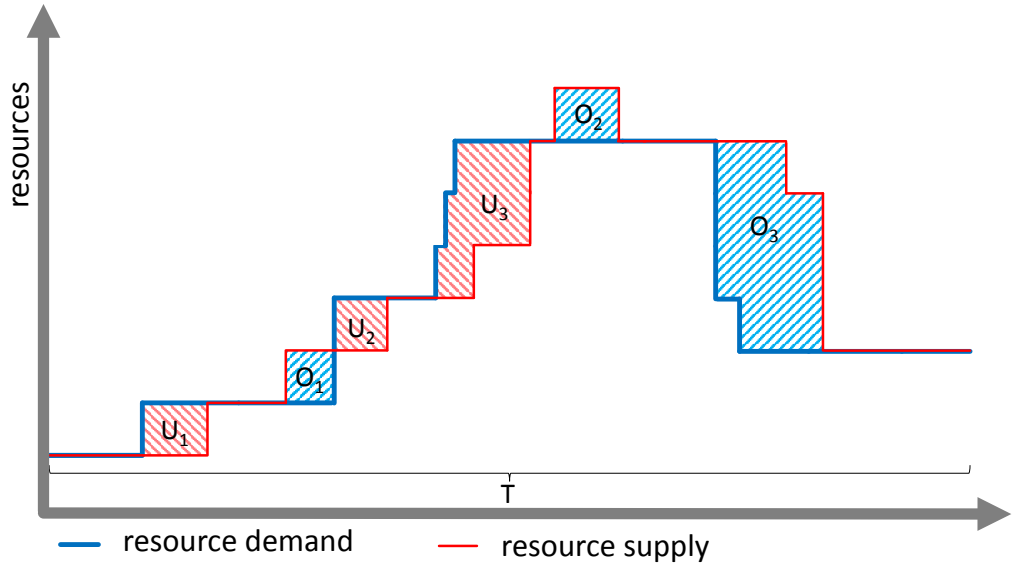


Figure 5.1: Measuring accuracy:
red areas: under-provisioning, blue areas: over-provisioning

The metrics are calculated by accumulating the amount of resources in under-provisioned states ($\sum U$, red areas) or over-provisioned ($\sum O$, blue areas) states and dividing them by

the length of the measurement period T.

$$\text{Accuracy for under-provisioning periods: } accuracy_U \text{ [resource units]} = \frac{\sum U}{T}$$
$$\text{Accuracy for over-provisioning periods: } accuracy_O \text{ [resource units]} = \frac{\sum O}{T}$$

Thus, $accuracy_U$ and $accuracy_O$[1] measure the average amount of resources that are over-provisioned during the measurement period. Of course, it is possible to combine these metrics to a global accuracy metric:

$$\text{global accuracy: } acc_{global} \text{ [resource units]} = acc_U + acc_O$$

As stated above, other weights for $accuracy_U$ and $accuracy_O$ are possible, too.

5.2 Timing

The second core aspect of elasticity discussed in Section 2.3.3 is timing. Timing is a very generic term. To be able to cover the different sub-aspects of timing, the benchmark includes different metrics which focus on the different sub-aspects.

5.2.1 Under- / Over-provision Timeshare

The accuracy metrics state how much resources are over- or under-provisioned on average during the measurement. These metrics do not show if the inaccuracy results from a few big deviations between demand and supply or if it is rather caused by a constant small deviation. Therefore, the following two metrics give more insights about how often under- or over-provisioning occurs.

The metric sums up the total amount time that was spend in an in under- ($\sum A$) or over-provisioned ($\sum B$) state and divides the sum by the duration of the measurement. Thus, it measures the timeshare spent in under- or over-provisioned states.

$$\text{Timeshare spent in under-provisioned state: } timeshare_U \text{ [\%]} = \frac{\sum A}{T}$$
$$\text{Timeshare spent in over-provisioned state): } timeshare_O \text{ [\%]} = \frac{\sum B}{T}$$

Today, cloud providers often advertise the availability of their infrastructure. Using these metrics or slightly transformed versions like

$$resource\_availability \text{ [\%]} = 1 - timeshare_U$$

allows providers to advertise their elasticity mechanism with statements like: "No under-provisioning occurs for 99,9% of the time".

As for the accuracy metric, the timeshare submetrics can be combined to a global metric:

$$\text{global timeshare: } timeshare_{global} \text{ [\%]} = timeshare_U + timeshare_O$$

This combined metric measures the timeshare where the CSUT is not able to match the demand because either under-provisioning or over-provisioning occurs. Furthermore, the it is possible to weight $timeshare_U$ and $timeshare_O$ differently according to user preferences.

[1] In [HKR13] these metrics are referred to as *precision*.

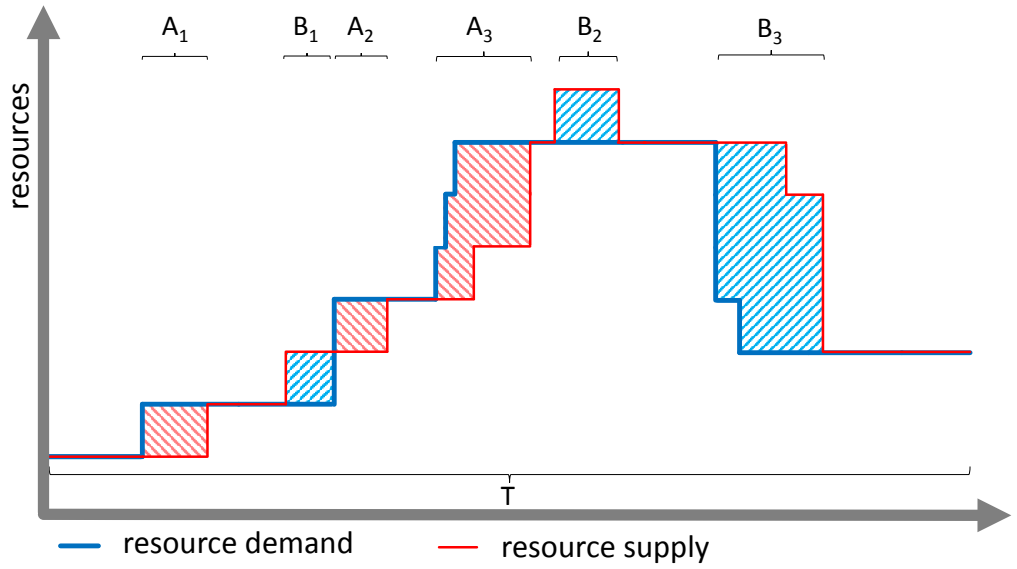


Figure 5.2: Measuring timing:
A_i: Time spent in under-provisioned state
B_i: Time spent in over-provisioned state

5.2.2 Jitter

Although the *accuracy* and *timeshare* metrics measure important aspects of elasticity, systems can still behave very different although they produce the same metric values for *accuracy* and *timeshare*. An example is shown in Figure 5.3.

Both systems have the same accuracy ($accuracy_U$, $accuracy_O$) and spend the same amount of time in the under-provisioned respectively over-provisioned states. However, the behavior of both systems is different. System B triggers a lot of unnecessary resource supply adaptations whereas System A triggers just a few. Within the developed benchmark this behavior is measured with a further metric.

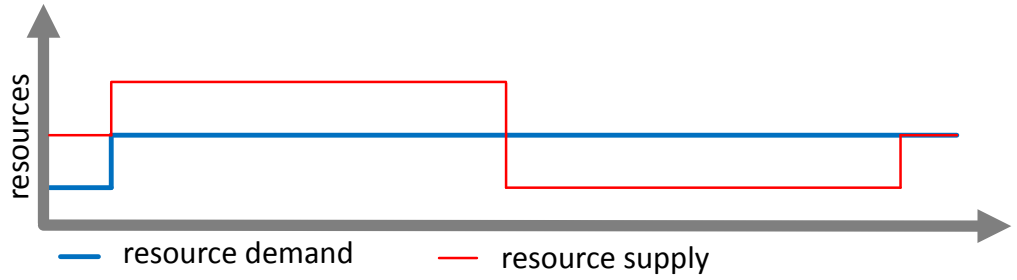
The *jitter* metric compares the amount of adaptations within the supply curve E_S with the number of adaptations within the demand curve E_D. The difference is normalized with the length of the measurement period T:

$$\text{Jitter metric: } jitter\left[\frac{\#adap.}{min}\right] = \frac{E_S - E_D}{T}$$

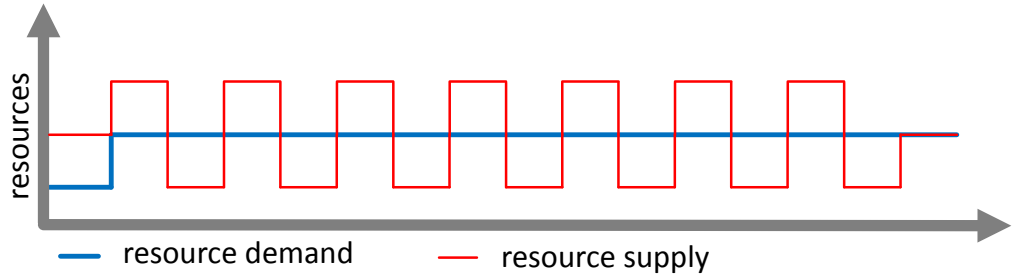
Big absolute values of *jitter* indicate that the system is not able to react on demand changes appropriately. When the *jitter* metric is negative that means that the system does not do enough allocations and behaves rather sluggish. A positive *jitter* metric means that the system does to do de-/allocations and tends to oscillate like Systems A (little) and B (heavily) in Figure 5.3.

Counting Adaptations

Figure 5.4 shows different possible behaviors after two successive demand changes. System C reacts by triggering two resource allocations of one resource. In contrast, System D triggers simply one supply change consisting of the allocation of two resource units. If the *jitter* metric simply counted supply and demand changes independently of the their size, this would result in a missing adaptation for System D and therefore a worse *jitter* metric compared to System C. This metric result would be unfair, because System D does not



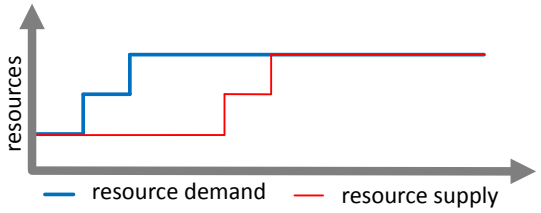
(a) System A



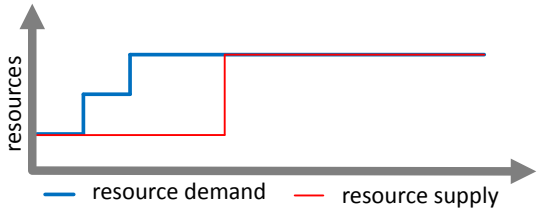
(b) System B

Figure 5.3: Systems with different elastic behaviors that produce equal results for *accuracy* and *timeshare* metrics

miss to allocate resources. To overcome this problem the *jitter* metric counts adaptations with respect to the resource granularity:
When the allocation of n resource units occurs at the same time within the resource supply or demand curve, this allocation is counted as n concurrently occurring adaptations. This way, the *jitter* metric evaluates to zero for System C and System D.



(a) System C



(b) System D

Figure 5.4: Systems C reacts with two small adaptations, System D with one big adaptation. Both systems do neither do superfluous adaptations nor miss to adapt the resource supply. Thus, the *jitter* metric should evaluate to zero for both.

5.3 Considered but Rejected Metrics

5.3.1 Delay

One way to characterize the timing behavior of a resource elastic system is to measure how fast the system reacts when the resource demand changes. Thus, a metric that measures something like the average time a system needs to react to a demand change with an according supply change, sounds useful. This paragraph discusses different approaches and explains why they have been rejected for the benchmark.

1. Use of a Speed [HKR13] Metric
   Herbst et al. [HKR13] proposed a speed metric which measures the average time that a system needs to return from an under- or over-provisioned state back to a state without under- or over-provisioning. In physics, speed is defined as distance divided by time. A speed metric which computes the average time to do something is therefore not intuitive. Additionally, this metric can not be derived when a mechanism either constantly over-provisions or constantly under-provisions. In particular, a constant (but small) over-provisioning is reasonable for a real conservative cloud provider. Furthermore, a small value for the speed metric is not necessarily an indicator for a high degree of elasticity. A system which provisions and deprovisions resources with an unnecessary high frequency - as shown in Figure 5.3(b) - may have a small value for the speed metric although its elasticity is suboptimal.
2. Measure Delay Between Demand and Supply Change
   Measuring the delay between demand and corresponding supply changes is difficult for several reasons. Depending on the elasticity strategy, resource allocations can occur in advance instead of after the corresponding demand change. In particular, this is true for proactive elasticity strategies. Thus, it is more appropriate to measure the time difference between demand changes and corresponding supply changes, instead of a delay.

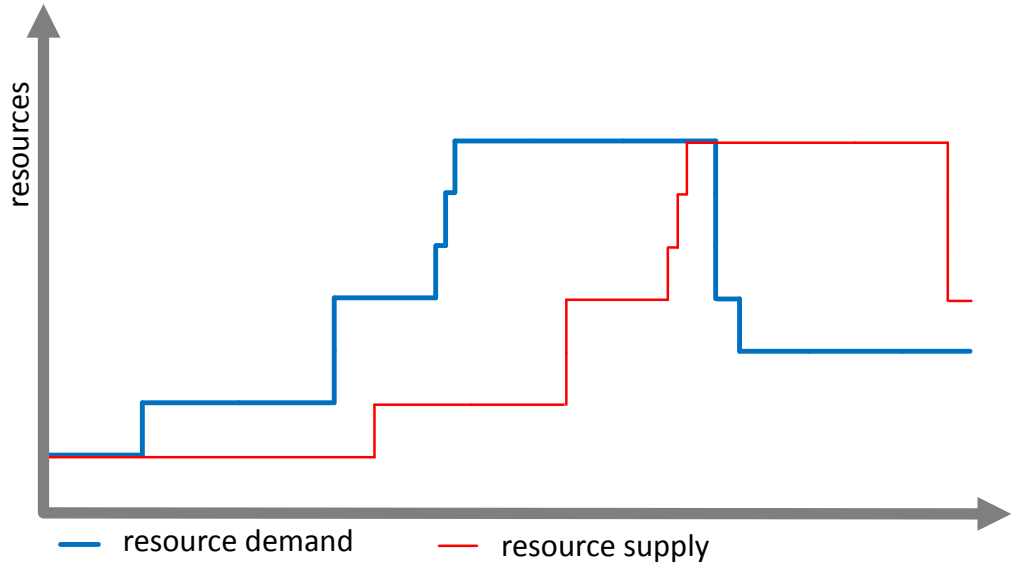


Figure 5.5: Matching supply change to demand changes is not trivial, when supply changes occur with a long delay.

However, it is not trivial to evaluate which supply change corresponds to which demand change. Supply changes can happen before or after their corresponding demand changes. In addition, the total number supply changes must not be equal to the number of demand changes. A simple way of creating those correspondences would be just assigning every supply change to the closest demand change. However, this would not produce the desired outcome in every case. An example where this assignment does not work is shown in Figure 5.5. More complex assignment strategies, that are based on a mapping created by a DTW algorithm for example may work for the scenario depicted in Figure 5.5. But still, there can be situations where resource allocations and deallocations happen without a correlation to a demand change. An example for such a behavior is shown in Figure 5.3(b). Filtering

out these oscillating behavior could be an option, but would lead to a metric which is not only difficult to measure but also not easy to understand.

5.3.2 Dynamic Time Warping Distance

The metrics within this benchmark are based on comparing two time series. The demand series, which contains points in time where the demand changes, and accordingly the supply series. A popular metric for comparing the distance between two timeseries is the dynamic time warping (DTW) distance [SC07]. Within the domain of automatic speech recognition it is often used to cope with different speaking speeds.

The DTW distance was used by Herbst [Her11] to measure the elasticity of thread pools. For this benchmark, the DTW distance was therefore also considered. The DTW distance is designed in way that the distance of two timeseries that describe curves with the same shape is equal, no matter if they match each other exactly or are shifted, as depicted in Figure 5.5 for example. However, a system which is able to match the demand all over the time, does not have the same elasticity as a system where the supply changes occur delayed. Thus, using the DTW distance as a measure for elasticity without modification seems not meaningful.

Another option is to use the DTW algorithm not for a direct elasticity measurement, but to create a mapping between demand and supply changes, to measure the average delay between demand and supply changes afterwards. Under the assumption, that the supply curve is mainly a shifted and or stretched demand curve, DTW seems to be applicable here. Unfortunately systems, can - as discussed in Section 5.3.1.2.: Measure Delay Between Demand and Supply - provision and deprovision resources without a correlation to demand changes. In this case, DTW creates mappings even between supply and demand changes that are not correlated.

For this reasons, DTW is not used within this benchmark at the moment.

5.4 Compare Different Systems Using Metrics

The sections above explained five different metrics, which allow to measure different aspects of elasticity: $accuracy_U$, $accuracy_O$, $timeshare_U$, $timeshare_O$ and $|jitter|$. All metrics have in common that smaller values signify higher degrees of elasticity. One option for comparing the elasticity of different CSUTs is to choose a single metric and compare the metric results for this single metric. However, benchmarks typically provide a measure that aggregates different metrics to a single number. This makes comparing the overall performance easier.

When aggregating the elasticity metrics to a single number metric, it is necessary to consider that the measurement units and therefore the scales of the metrics are different:

- $accuracy_U$ and $accuracy_O$ measure average resource amount deviations
- $timeshare_U$ and $timeshare_O$ measure ratios of time periods
- $|jitter|$ measures deviations of adaptation event amounts normalized with time

Thus, simply averaging the metric results for one system and comparing the result to the averaged metric result for another system does not allow a fair comparison.

The following subsections discuss three ways for creating an aggregated elasticity measure.

5.4.1 Distance Based Aggregation

One way to create a single elasticity measure bases on measuring the distance between system behaviors. The elastic behavior of a system is characterized by its metric vector m:

$$m = (accuracy_U, accuracy_O, timeshare_U, timeshare_O, |jitter|)$$

The metric values for an ideal elastic behavior are known to be zero for every element of the vector.

$$m_{ideal} = (0, 0, 0, 0, 0)$$

Different elastic behaviors can now be compared with the help of their *distance* to the ideal behavior m_{ideal} as a single number measure. The distance must be computed using an appropriate distance measure that accounts for the different scales of the metrics. Normalizing the metric would allow to use the *euclidean distance*, but normalization without distortion is not possible for $accuracy_O$ and $|jitter|$ since no upper bound exists for them.

A possible distance measure is the *mahalanobis distance* [Mah36]. It requires a covariance matrix which describes the variance of the metrics and the correlation between them. The matrix can be obtained using the metric results for a set of sample behaviors. As a drawback, adding new sample behaviors changes the covariance matrix and therefore potentially the metric results.

5.4.2 Speedup Based Aggregation

Performance benchmarks often use a baseline system and rank systems according to their performance speedup with respect to the baseline system. This thesis presents a similar approach that ranks systems according to their *elastic speedup*. As a limitation, this approach includes the *accuracy* and the *timeshare* metrics only, not the *jitter* metric. It consists of the following steps:

1. Aggregate the accuracy and timeshare sub metrics into a weighted accuracy and a weighted timeshare metric, respectively.
2. Compute elasticity speedups for both of the aggregated metrics using the metric values of a baseline system.
3. Use the geometric mean to aggregate the speedups for *accuracy* and *timeshare* to a *elasticspeedup* measure.

The resulting *elasticspeedup* measure can be used to compare systems without having to compare each elasticity metric separately.

Each of the three steps is now explained more detailed:

1. The $accuracy_U$ and $accuracy_O$ metrics are combined to a weighted accuracy metric $accuracy_{weighted}$:

$$accuracy_{weighted} = w_{acc_U} \cdot accuracy_U + w_{acc_O} \cdot accuracy_O$$

with

$$w_{acc_U}, w_{acc_O} \in (0, 1), \quad w_{acc_U} + w_{acc_O} = 1$$

In the same way, the $timeshare_U$ and $timeshare_O$ metrics are combined to a weighted timeshare metric $timeshare_{weighted}$:

$$timeshare_{weighted} = w_{ts_U} \cdot timeshare_U + w_{ts_O} \cdot timeshare_O$$

with

$$w_{ts_U}, w_{ts_O} \in (0, 1), \quad w_{ts_U} + w_{ts_O} = 1$$

2. Let x be a vector that stores the metric results:

$$x = (x_1, x_2) = \left(accuracy_{weighted}, timeshare_{weighted}\right)$$

For a metric vector x_{base} of a given baseline system and a metric vector x_k of a benchmarked system k, the speedup vector s_k is computed as follows:

$$s_k = \left(s_{k_{accuracy}}, s_{k_{timeshare}}\right) = \left(\frac{x_{base_1}}{x_{k_1}}, \frac{x_{base_2}}{x_{k_2}}\right)$$

3. The elements of s_k are aggregated to an unweighted *elastic speedup*$_{unweighted_k}$ measure using the geometric mean:

$$elastic\ speedup_{unweighted_k} = \sqrt{s_{k_{accuracy}} \cdot s_{k_{timeshare}}}$$

The geometric mean produces consistent rankings, no matter how measurements are normalized and is the only correct mean for normalized measurements [FW86]. Thus, the ranking of the systems according to *elastic speedup*$_k$ is consistent, regardless of the chosen baseline system. The geometric mean is also used in some popular performance benchmarks such as the SPEC CPU2006 in order to aggregate the results for different workloads [RR10].

Furthermore, different preferences concerning the elasticity aspects can be taken into account by using the weighted geometric mean for computing the *elastic speedup*$_{weighted_k}$:

$$elastic\ speedup_{weighted_k} = {s_{k_{accuracy}}}^{w_{acc}} \cdot {s_{k_{timeshare}}}^{w_{ts}}$$

with

$$w_{acc}, w_{ts} \in [0,1],\ \ w_{acc} + w_{ts} = 1$$

The following equation summarizes the three steps:

$$elastic\ speedup_{weighted_k} = \left(\frac{w_{acc_U} \cdot accuracy_{U_{base}} + w_{acc_O} \cdot accuracy_{O_{base}}}{w_{acc_U} \cdot accuracy_{U_k} + w_{acc_O} \cdot accuracy_{O_k}}\right)^{w_{acc}} \cdot \left(\frac{w_{ts_U} \cdot timeshare_{U_{base}} + w_{ts_O} \cdot timeshare_{O_{base}}}{w_{ts_U} \cdot timeshare_{U_k} + w_{ts_O} \cdot timeshare_{O_k}}\right)^{w_{ts}} \tag{5.1}$$

with

$$w_{acc_U}, w_{acc_O}, w_{ts_U}, w_{ts_O} \in (0,1),\ \ w_{acc}, w_{ts} \in [0,1],$$
$$w_{acc_U} + w_{acc_O} = 1,\ \ w_{ts_U} + w_{ts_O} = 1,\ \ w_{acc} + w_{ts} = 1$$

5.4.2.1 Elasticity Metric Weights

The single number elasticity measure shown in Equation 5.1 can be adjusted according to the preferences of the target audience by using different weights. For example, the accuracy weights $w_{acc_U} = 0.2,\ \ w_{acc_O} = 0.8$ allow to amplify the influence of the amount of over-provisioned resources compared to the amount of under-provisioned resources. A reason for doing so could be that over-provisioning leads to additional costs, which mainly depend on the amount of over-provisioned resources. The cost for under-provisioning in contrast does not depend that much on the amount of under-provisioned resources but more on how long the system under-provisions. This observation can be taken into account by using timeshare weights like: $w_{ts_U} = 0.8,\ \ w_{ts_O} = 0.2$. Finally, when combining the *accuracy* and *timeshare* speed ups, the weights $w_{acc},\ w_{ts}$ can help to prioritize different elasticity aspects. Here, weights like $w_{acc} = \frac{1}{3},\ \ w_{ts} = \frac{2}{3}$ for example would double the importance short under- and over-provisioning periods compared to the importance of small under- or over-provisioning amounts.

5.4.2.2 Including the Jitter Metric

The jitter metric indicates whether a system tends to either be overly responsive exhibiting many unnecessary resource de-/allocations, or to react rather sluggish and slow exhibiting not enough allocation changes. In order to include it into the *elastic speedup*, deriving a speed up for the *jitter* metric is desirable. This speedup can then be included in the geometric mean in the third step describe above. In contrast to the weighted *accuray* and *timeshare* metrics, the *jitter* metric can be zero even for systems with imperfect elasticity. Computing a speed up $s_{k_{jitter}}$ by dividing the *jitter* metric for the baseline system $jitter_{base}$ by *jitter* metric for the evaluated system $jitter_k$ is therefore not sensible.

A possible alternative is not to calculate the speedup for the *jitter* metric directly, but with a helper method $h(x)$:

$$s_{k_{jitter}} = \frac{h\left(jitter_{base}\right)}{h\left(jitter_k\right)} \qquad \text{with } \forall x : h(x) > 0$$

Possible options for $h(x)$ are $h(x) = 1 + x$ or $h(x) = e^x$. The drawback of this approach is, that the speedup is distorted.

5.4.3 Cost Based Aggregation

Evaluating the financial impacts of different elastic behaviors is not in focus of this thesis. However, the presented metrics measure elasticity aspects that can be mapped to costs. Aggregating these costs allows creating a single number measure that can be used to rank different elastic systems or different elasticity mechanism configurations. The following equation illustrates an exemplary cost based aggregation:

$$\begin{aligned} cost_{imperfect\_elasticity} = {} & (accuracy_O - accuracy_U) \cdot time\_period_{ref} \cdot cost_{resource} && \text{(i)} \\ & + timeshare_U \cdot time\_period_{ref} \cdot cost_{underprovisioning} && \text{(ii)} \\ & + cost_{jitter}(jitter) && \text{(iii)} \end{aligned}$$

The measure $cost_{imperfect\_elasticity}$ evaluates the costs caused by an imperfect elastic behavior during a reference time period $time\_period_{ref}$. These costs can be calculated by adding (i) the costs caused by unnecessary resource allocations and (ii) the costs of underprovisioning (i.e., violating SLOs) for a certain amount of time. Furthermore, (iii) costs for superfluous adaptations are included in the last addend. Including these costs makes sense when the billing granularity with respect to time is rather coarse grained. Some providers (e.g., Amazon) bill their customers for example for five instance hours even when the customer simply requested one instance for a three minutes five times in a row.

For comparisons between different providers, the costs caused by imperfect elasticity $cost_{imperfect\_elasticity}$ must be added to the absolute costs $cost_{ideal\_elasticity}$ for a perfect elastic behavior in order to allow a meaningful comparison.

While using the *Benchmark Calibration*, which adjusts load profiles, is important for evaluating the technical aspects of elasticity, it must not be used for evaluating financial impacts of elasticity. Comparing two systems with a load that requires up to 20 instances on one system but up to 25 instances on another system with the help of a system specific adjusted load profile that induces a demand of up to 20 instances on both systems is fair for evaluating the technical property elasticity with respect to using 20 instances. It is not fair for evaluating the financial impact of elasticity, since the second system requires 25 instances and thus a comparison must reflect the costs for using them. Therefore, comparisons with respect to financial costs must be conducted with unadjusted load profiles. The benchmark concept as well as the developed benchmarking framework are applicable for evaluating the financial impact of elasticity by omitting the *Benchmark Calibration* activity.

6. BUNGEE - An Elasticity Benchmarking Framework

This chapter explains how the resource elasticity benchmark concept is implemented in *BUNGEE*, the benchmarking framework that has been developed in course of this thesis. The development of *BUNGEE* addresses the fifth goal mentioned in Section 1.1.

BUNGEE consists of two parts:

- *The benchmark harness*, which runs outside of the CSUT and performs different benchmarking tasks like calibration, measurement and metric evaluation is presented in Section 6.1.
- *The cloud-side load generation application*, which utilizes resources within the CSUT is presented in Section 6.2.

6.1 Benchmark Harness

The benchmark harness is a java application that allows to analyze a CSUT, to adjust load profiles in a system specific manner, and to measure the resource allocations on a CSUT while it is exposed to a load. Additionally, it supports the evaluation of elasticity metrics by comparing resource demand with resource supply curves.

Section 6.1.1 explains the benchmark harness on architectural level and explains its dependencies. The different components of the benchmark harness are then illustrated in Sections 6.1.2 - 6.1.9.

6.1.1 Architectural Overview

In the following, the benchmark harness is explained with the help of two different views. Firstly, an activity diagram explains the benchmarking workflow. Secondly, a package diagram visualizes the package structure of the harness.

Activity Diagram

In the concept Chapter 4, Figure 4.2 illustrated a coarse grained benchmarking workflow. Figure 6.1 illustrates the process of benchmarking a CSUT more detailed. For all four activities the control flow and the object flow are shown. The names of the parameter nodes are equal to the class/interface names of the corresponding java entities.

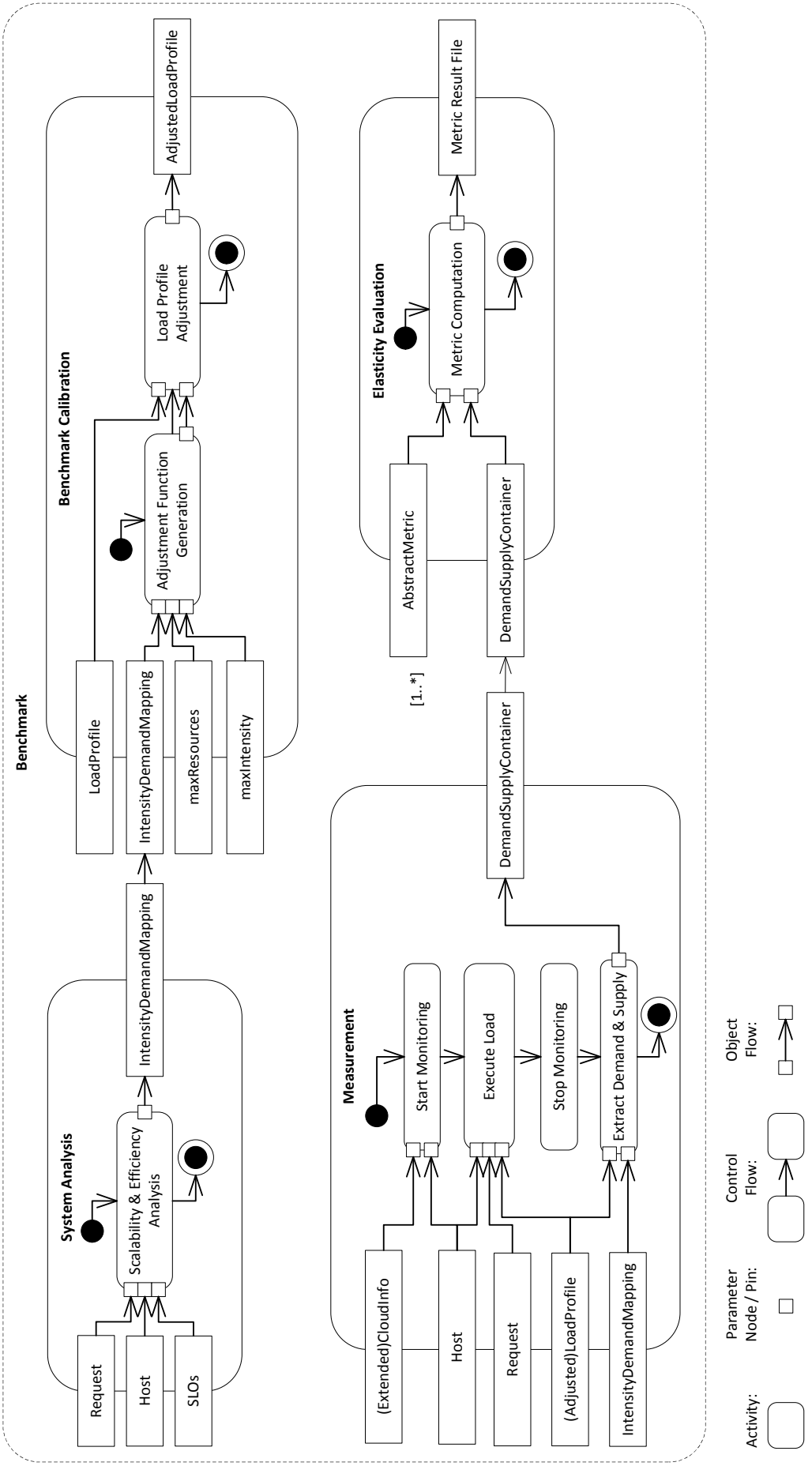


Figure 6.1: Control and object flow between and within the benchmarking activities

In the following, the different activities as well as their input parameters and return values are explained.

1. **System Analysis**
   The *System Analysis* evaluates the CSUT's load processing capabilities at different scaling stages and thereby its scaling behavior and the level of efficiency of its underlying resources. The analysis requires the specification of a `Request`, a `Host` and a number of `ServiceLevelObjectives` as input. When finished, it returns an `IntensityDemandMapping` that specifies the mapping of load intensities to resource demands. More details on the implementation about the analysis can be found in Section 6.1.4.

2. **Benchmark Calibration**
   During the *Benchmark Calibration*, a given load profile is adjusted in a system specific manner. In a first sub-activity, an `AdjustmentFunction` is generated. This `AdjustmentFunction` is then applied on a `LoadProfile` in a second sub-activity. In order to generate an `(Adjusted)LoadProfile`, a `LoadProfile` as well as an `IntensityDemandMapping` of a preceding *System Analysis* are required. Additionally, the parameters `maxResources` and `maxIntensity` are necessary. Implementation details about the calibration can be found in Section 6.1.5.

3. **Measurement**
   During a *Measurement* run, the resource allocations on the CSUT are monitored while the CSUT is exposed to a load varied according to an `(Adjusted)LoadProfile`. Apart from the `LoadProfile` a `Host` and a `Request` specification are required as input. To derive the demand, an `IntensityDemandMapping` is a further requirement. Finally, the monitoring of resource allocation requires access to the cloud management system via the `CloudInfo` interface. As a result, the activity returns a `DemandSupplyContainer`, which contains the demand and the supply curve. Implementation details about the load generation and about monitoring cloud systems can be found in Section 6.1.3 and Section 6.1.7, respectively.

4. **Elasticity Evaluation**
   The *Elasticity Evaluation* activity uses a `DemandSupplyContainer` and a set of `AbstractMetrics` as input. It evaluates the metrics and write the results into a file. Implementation details about the metrics can be found in Section 6.1.8.

Package Diagram

The package diagram shown in Figure 6.2 illustrates the packages in which the elasticity benchmark harness is organized.

- Package `loadprofile`:
  Models load profiles. More information and a class diagram can be found in Section 6.1.2.

- Package `loadgeneration`:
  Contains classes which control the load generation via JMeter. More information and a class diagram can be found in Section 6.1.3.2.

- Package `slo`:
  Contains classes for evaluating responses with respect to SLOs. More information and a class diagram can be found in Section 6.1.3.3.

- Package `analysis`:
  Contains classes that allow analyzing the CSUT at different scaling stages. More information and a class diagram can be found in Section 6.1.4.

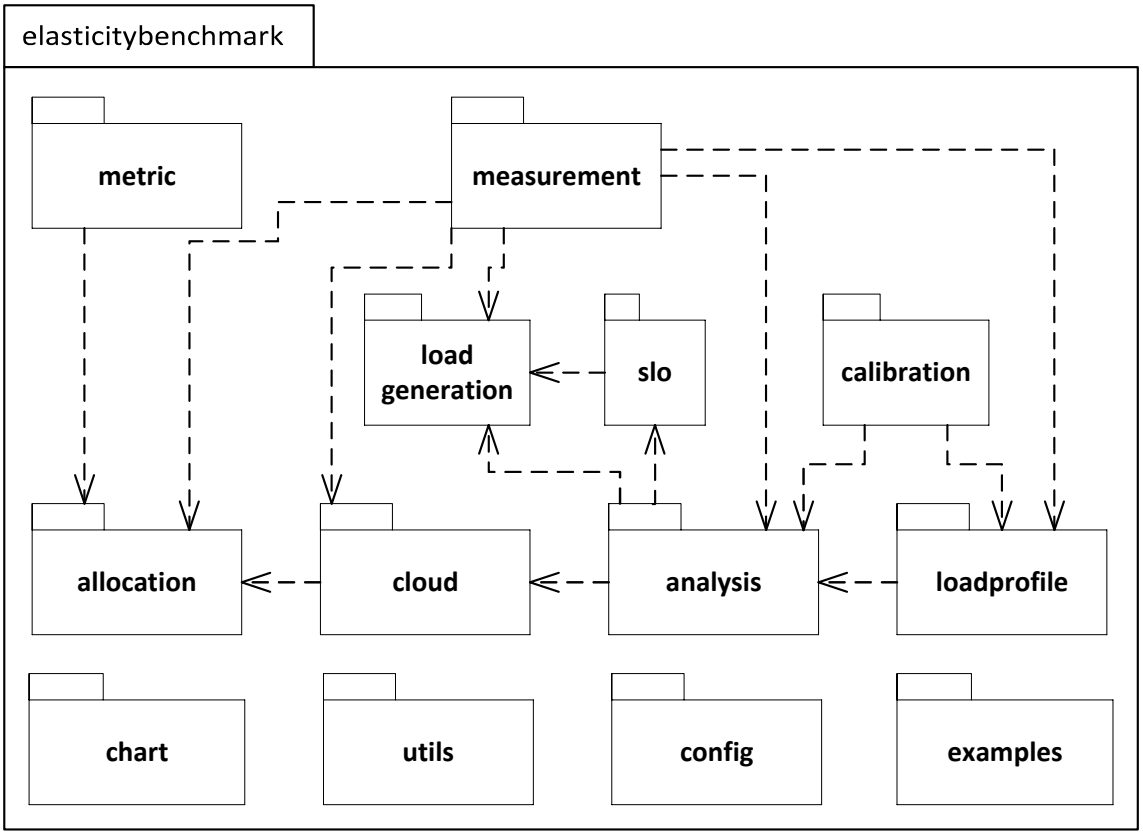


Figure 6.2: Package diagram of the elasticity benchmark framework

- Package `calibration`:
  Contains classes that allow the benchmark to adjust load profiles. More information and a class diagram can be found in Section 6.1.5.
- Package `allocation`:
  Contains data structures for series of demand or supply allocations. More information and a class diagram can be found in Section 6.1.6.
- Package `allocation`:
  Contains interfaces that define the cloud information access and control options. More information and a class diagram can be found in Section 6.1.7.
- Package `metrics`:
  Contains classes that implement different elasticity metrics. More information and a class diagram can be found in Section 6.1.8.
- Package `measurement`:
  Contains the class *MeasurementRunner* which implements the *Measurement* activity.
- Package `utils`:
  Contains utility classes mainly for date and file functions. Incoming dependencies are not shown for the sake of clarity.
- Package `config`:
  Contains parameter data structures such as *Host* or *Request*. Incoming dependencies are not shown for the sake of clarity.
- Package `chart`:
  Allows to generate graphical representations of measurements. Outgoing depen-

dencies are not shown for the sake of clarity. More information can be found in Section 6.1.9

- Package `examples`:
  Contains small examples that demonstrate how the benchmarking activities *System Analysis*, *Calibration*, *Measurement* and *Evaluation* can be triggered. Additionally, the package contains examples which demonstrate how load profiles, mapping functions or resource allocation curves can be illustrated using the `chart` package. Outgoing dependencies are not shown for the sake of clarity.

Dependencies

To induce a varying load on a CSUT, the benchmark needs a load profile that defines how the load intensity varies over time. The benchmark harness uses **Descartes Load Intensity Model (DLIM)** [vKHK14a] instances for this purpose. In order to process the model instances the **LIMBO** [vKHK14b] toolkit is used. LIMBO publicly available [1].

During analysis and during measurement runs, **JMeter**[Hal08] is used as a load generator. Running the benchmark therefore requires a JMeter installation.

The benchmark harness offers different options for visualizing the measured data. For the graph creation the java chart library **JFreeChart**[2] is used.

Currently, **CloudStack** and **Amazon Web Services (AWS)** are supported cloud management softwares. *BUNGEE* therefore depends on the *cloudstack-api-java*[3] and on the *aws-sdk-java*[4].

Support of other Cloud Platforms

In order to use *BUNGEE* for evaluating other cloud platforms the implementation of at least the `CloudInfo` interface in the `cloud` package is required:

- Interface `CloudInfo`:
  Implementation is required for benchmarking the platform.
- Interface `ExtendedCloudInfo`:
  Required to support passive monitoring in addition to active monitoring (compare Section 4.5.2).
- Interface `CloudManagement`: Required to support the *Detailed System Analysis* in addition to the *Simple System Analysis* (compare Section 4.4.1).

All interfaces are explained in Section 6.1.7, which explains the cloud package.

6.1.2 Load Profiles

Load profiles model the variation of load intensity over time. As illustrated in Figure 6.3, the interface `LoadProfile` is used to represent a load profile within the benchmark. The `LoadProfile` interface is realized by the class `DlimAdapter`. This class uses a DLIM [vKHK14a] instance (the `rootSequence` member of type `Sequence`), to store the load profile. To evaluate the DLIM instance, functionality implemented in the LIMBO [vKHK14b]

[1] LIMBO http://se2.informatik.uni-wuerzburg.de/mediawiki-se/index.php/Tools#LIMBO:_Load_Intensity_Modeling_Tool
[2] JFreeChart http://www.jfree.org/jfreechart
[3] cloudstack-api-java https://code.google.com/p/cloudstack-api-java
[4] aws-sdk-java http://aws.amazon.com/sdkforjava

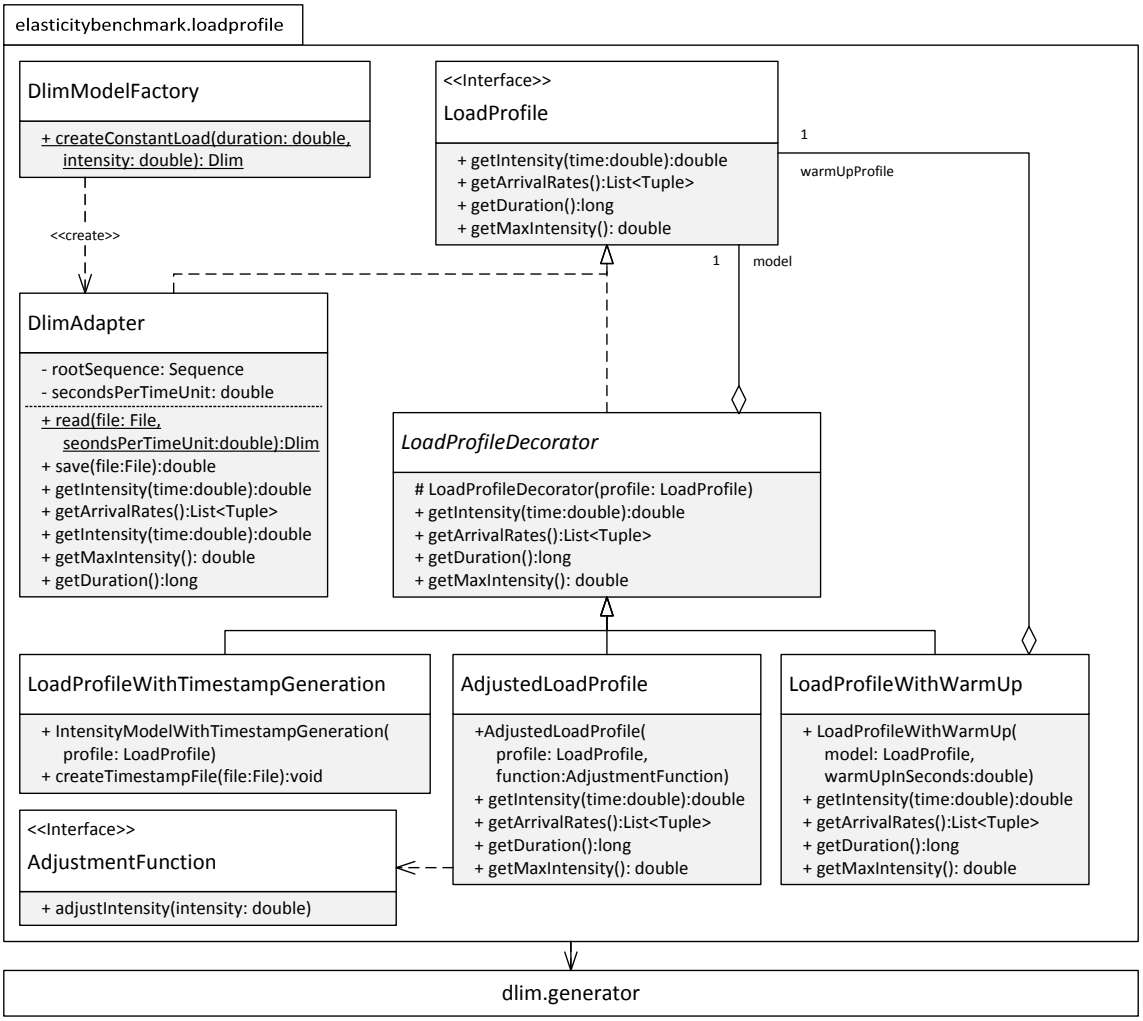


Figure 6.3: Class diagram of the `loadprofile` package

toolkit (package: `dlim.generator`) is used. DLIM instances model load intensity as arrival rate per time unit. The time unit itself is not stored within the model. Thus, the `DlimAdapter` class has a `secondsPerTimeUnit` member to store this information.

`Dlim` instances can be created either by reading a given *.dlim file or by using the `DlimModelFactory` to dynamically create constant load profiles of a predefined intensity. The first option is useful for measurement runs, while the second facilitates the *System Analysis* which requires the generation of constant loads.

`LoadProfiles` can easily be enhanced with the help of the decorator pattern. For example, it is possible to add a warm up period at the beginning of a profile with the help of the `LoadProfileWithWarmUp` class. Furthermore, the `AdjustedLoadProfile` class adjusts a given profile by applying an `AdjustmentFunction`. This is very helpful within the calibration part of the benchmark. The `LoadProfileWithTimestampGeneration` class adds functionality to create a timestamp file. The timestamp file generation uses the LIMBO toolkit. Hereby, equal distance sampling is configured to be used as sampling method. LIMBO supports other sampling methods, such as uniform distribution sampling, as well. The generated timestamp files is then used as input for JMeter [Hal08], an external load generation tool.

6.1.3 Load Generation and Evaluation

Most load generation tools, like JMeter [Hal08], httperf [MJ98], Faban[5] or Rain [BLY+10] mainly target the execution of closed workloads. In contrast, this benchmark bases on an open workload model. This means that requests are sent independently at fixed points in time. Additionally, for the benchmark it is necessary to vary the frequency of request submissions - the load intensity - over time. Faban currently offers in version 1.2 an experimental feature which allows to vary the load intensity indirectly by varying the number of used threads. Rain lacks proper documentation and is therefore not considered further. JMeter and httperf allow to replay webserver logs, but they do not use the timestamps when replaying the logs. However, JMeter can be extended easily. Therefore, the benchmark uses JMeter together with a newly developed extension: The *TimestampTimer* JMeter plugin[6].

The next Subsection 6.1.3.1 explains this plugin. Subsection 6.1.3.2 illustrates how the extended version of JMeter is used within the benchmark. Finally, the Subsection 6.1.3.3 illustrates how the benchmark evaluates the SLO compliance of processed requests.

6.1.3.1 JMeter Extension: TimestampTimer Plugin

The TimestampTimer plugin allows to delay requests according to a list of timestamps specified in a timestamp file. In order to do so, it provides a new timer element. In contrast to already existing JMeter timers, which delay the execution of a thread for a fixed period of time or according to a distribution, this new timer delays a thread according to timestamps. Within this benchmark the timestamp file is created automatically from a load profile. The plugin however can also be used for other use cases, where the timestamp file is created differently.

Since one thread is not sufficient - except for very low intensities and short response times - to submit all requests in time, the submission and response handling for different requests must be assigned to different threads. As discussed in Section 4.3.3.1 different techniques are possible for this assignment.

The dynamic assignment (Strategy 2) exhibits a greater flexibility compared to the static assignment (Strategy 1) and was therefore implemented.

Depending on how the timer element is used within JMeter, it is possible to realize both of the dynamic assignment strategies described in Section 4.3.3.1:

- Waiting Master Thread (Strategy 2a)
  For the *Waiting Master Thread* variant, two thread groups are created. The first one contains just the master thread and the timer, the other one contains the worker threads which handle the actual requests synchronously. The master thread in the first thread group sends requests for submissions to the second thread group over a communication mechanism provided by JMeter. One of the worker threads receives the request inquiry and executes the request immediately. The timer delays the master thread in way that requests for submission are send to the second thread group according to the submission times specified in the timestamp file.
- Waiting Worker Threads (Strategy 2b)
  The *Waiting Worker Threads* variant requires just one thread group, which contains the timer. The threads take timestamps from the central queue and execute the corresponding requests only after being delayed by the timer according to corresponding timestamps.

[5] Faban http://faban.org
[6] TimestampTimer http://github.com/andreaswe/JMeterTimestampTimer

The *Waiting Worker Threads* variant creates less overhead because no communication between thread groups is necessary. It is therefore used in the benchmark.

The JMeter TimestampTimer plugin is publicly available on GitHub[6]. Besides the plugin itself, the repository contains test plan files (*.jmx) defining JMeter test plans. These test plans demonstrate how the timer can be used to create a *Waiting Worker Threads* or a *Waiting Master Thread* assignment.

6.1.3.2 Automated Load Execution and Validation

General Approach

Whenever the benchmark needs to expose the CSUT to a load according to a load profile, the benchmark first generates a timestamp file. JMeter is then started to execute the load.

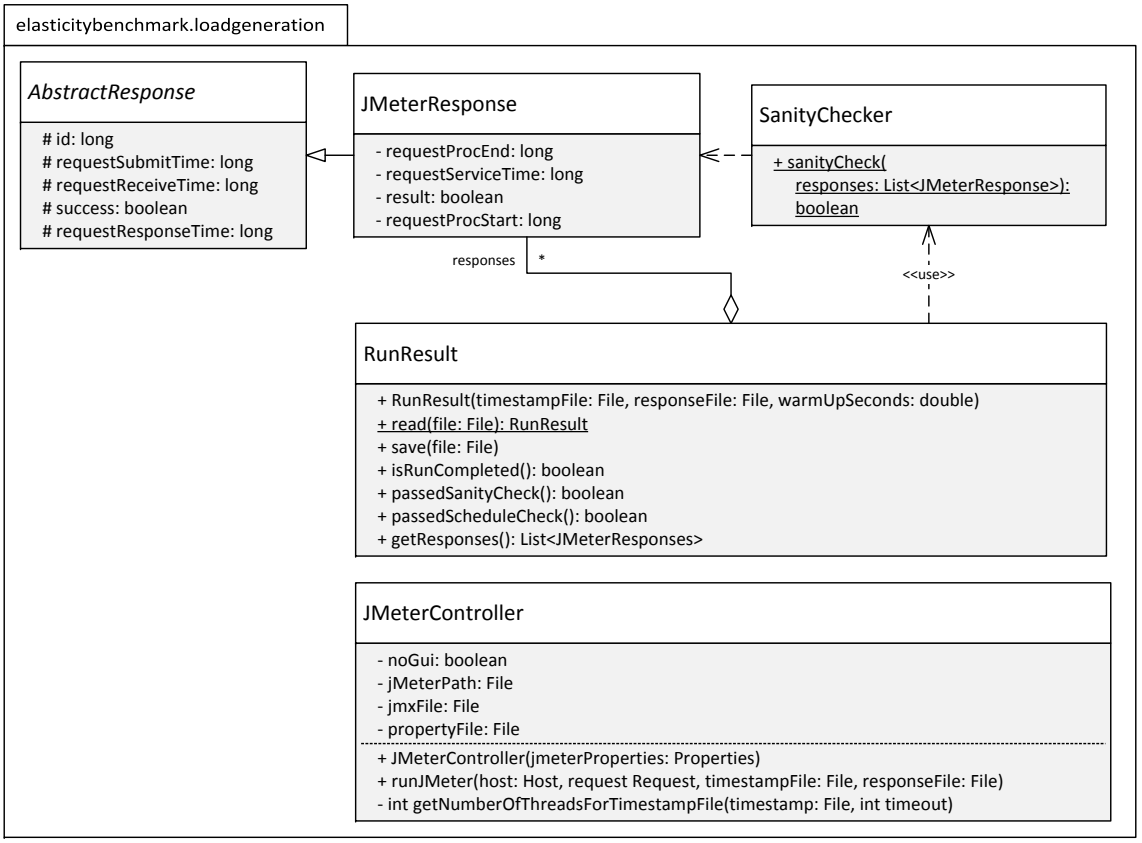


Figure 6.4: Class diagram of the `loadgeneration` package

Figure 6.4 illustrates the class diagram for the `loadgeneration` package. The class `JMeterController` is responsible for executing a load according to a given timestamp file. The request results are stored in a response file.

To reduce the overhead produced by a graphical user interface[7], JMeter is started in the non-gui mode. A test plan which contains the TimestampTimer, a prepared HTTP-Request and a file writer to persist information about the requests is passed to the JMeter in the form of a test plan file. Additionally, parameters which specify the CSUT target, such as hostname and port (parameter `Host`) or request specific parameters, like *problemSize* or *requestTimeout* (parameter `Request`), are passed as command line parameters.

[7] http://www.ubik-ingenierie.com/blog/jmeter_performance_tuning_tips

The file writer element that is contained in the test plan, logs and writes for every request the following data into a response file: unique identificator (UID), request submission time, response receive time, response time (= response receive - request submission), request success. For requests that have been received within the specified timeout, the following additional data is parsed from the response and is logged into the response file:

- request processing start time
- request processing end time
- request service time (request processing end time - request processing start time)
- request result
- internal ip address of the responding VM

The number of threads that are created by JMeter to execute the load, is calculated with Algorithm 1 (compare Section 4.3.3.1) in the `getNumberOfThreadsForTimestampFile()` method in the `JMeterController` class. In initial experiments, I observed that in rare cases requests are aborted only after exceeding the specified timeout about a factor of four. In order reduce the likelihood that this behavior results in an inaccurate timing, JMeter is configured to use five times more than the calculated number of required threads according to Algorithm 1.

Request Submission and Response Evaluation

After JMeter has finished sending the requests, the benchmark uses the response file and the timestamp file to analyze the quality of the request submission.

Since the JMeter logs the information for a request after either the corresponding response has been received or the request timed out, the information is not ordered the same way as in the timestamp file. Therefore, the request information is sorted by the UID.

The class `RunResult` holds the all information about a JMeter run: The timestamps used for sending the requests as well as the logged request response information.

Sanity Checks

Within a first evaluation step, the following sanity checks are performed for each request to test if the logged data is valid:

$$
\begin{aligned}
&\text{for each request } i, i = i..n\text{:}\\
&\qquad (\text{submission time})_i \leq (\text{response receive time})_i\\
&\qquad (\text{processing start time})_i \leq (\text{processing end time})_i\\
&\qquad (\text{service time})_i \leq (\text{response time})_i\\
&\qquad (\text{submission time})_i \leq (\text{processing start time})_i + \text{allowed delay}\\
&\qquad (\text{processing end time})_i \leq (\text{response receive time})_i + \text{allowed delay}
\end{aligned}
$$

Hereby, n is the total number of requests and *allowed delay* specifies the maximal allowed deviation between the CSUT clock and the load driver clock. If one of the last two checks fails, this is an indicator, that the clock of the load driver is not synced correctly with the clock of the CSUT.

The sanity check is implemented in the `RunResult` class and can be queried with the `passedSanityCheck()` method.

Accuracy of Request Submission

In a second evaluation step, the accuracy of the request submission is evaluated as previously described in Section 4.3.3.2. The accuracy evaluation is also implemented in the `RunResult` class.

6.1.3.3 Load Evaluation

In the *System Analysis* the CSUT is tested with different amounts of allocated resources whether it can sustain a certain load intensity without violating predefined SLOs, as explained in detail in Section 4.4.1. Figure 6.5 illustrates the class diagram for the `slo` package, which contains classes to perform this evaluation.

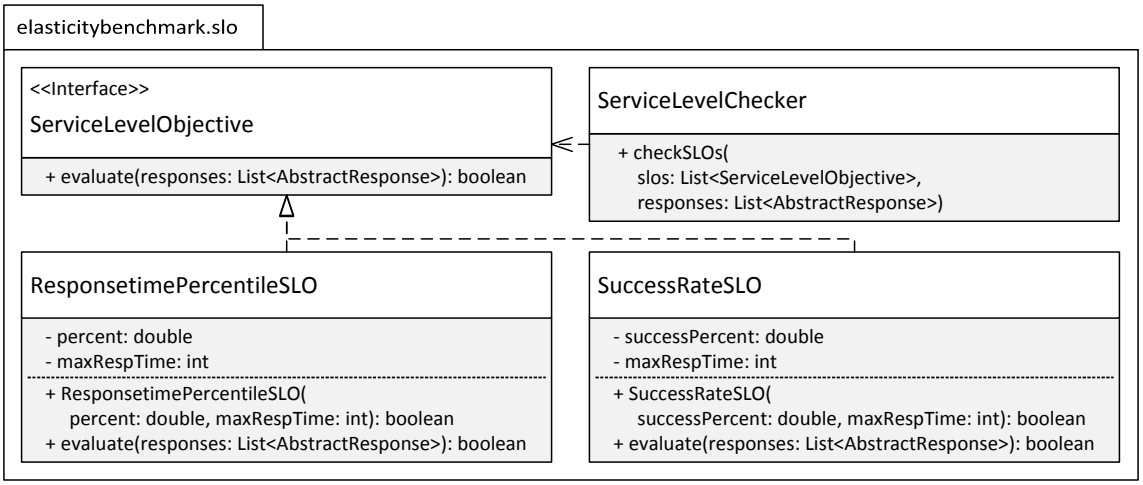


Figure 6.5: Class diagram of the `slo` package

Concrete SLOs are realizations of the `ServicelLevelObjective` interface. For this thesis, `ResponsetimePercentileSLO` and `SuccessRateSLO` were implemented as examples for SLOs.

The `ResponsetimePercentileSLO` evaluates, if the `percent`-percentile for the response time is smaller than the maximum response time `maxRespTime`. Hereby, `percent` and `maxRespTime` are configurable parameters.

The `SuccessRateSLO` additionally considers the successful completion of requests. It evaluates, if at least a share of `successPercent` of the issued requests is answered successfully within a given maximum response time `maxRespTime`.

The class `ServiceLevelChecker` evaluates, if a list of responses complies with a list of SLOs.

Due to the simple interface, the benchmark can be easily extended to support other SLOs.

6.1.4 System Analysis: Evaluation of Load Processing Capabilities

Figure 6.6 shows a class diagram for the classes in the `analysis` package. The benchmark offers two different ways of analyzing the load processing capabilities of a CSUT. Both analysis functions are implemented in subclasses of the abstract `SystemAnalysis` class.

The `SystemAnalysis` class allows to specify a maximal resource amount, up to which the system is analyzed. Furthermore, it defines the abstract function `analyzeSystem()`,

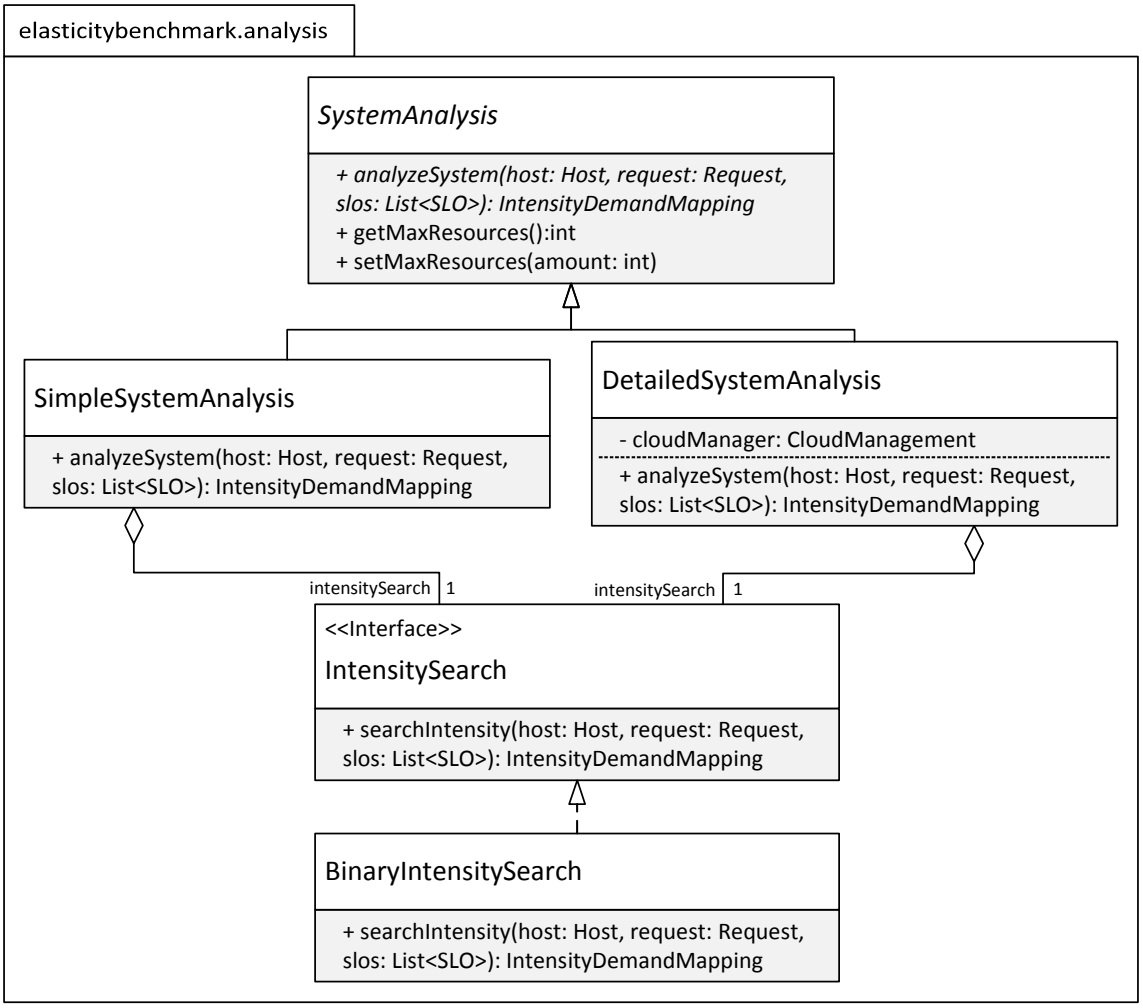


Figure 6.6: Class diagram for the `analysis` package

which must be implemented by subclasses. The return type `IntensityDemandMapping` represents the mapping function *demand*(*intensity*) (compare Section 4.4.1).

Both subclasses of `SystemAnalysis` use the `IntensitySearch` interface to analyze the load processing capabilities of a scaling stage. As discussed in Section 4.4.1, different algorithms can be used for the intensity search. The `BinaryIntesitySearch` class implements Algorithm 3.

The flexibility of the design allows to add new intensity search algorithms or additional analysis implementations, easily.

The following paragraphs explain the two different analysis implementations.

Simple System Analysis

The *Simple System Analysis* implemented in the `SimpleSystemAnalysis` class assumes that the resource demand increases linearly with the load intensity. It analyzes the load processing capabilities of the CSUT only for using one resource. The load processing capabilities for using more resources is then extrapolated linearly. The resulting mapping function *demand*(*intensity*) contains therefore always steps of equal length.

The benefit of this simplified analysis approach is that there is no need to control the cloud programmatically. Since only the first scaling stage is analyzed, the cloud must

not be reconfigured to use more resources. Thus, it is not necessary to use the API of the cloud management software. This reduces the overhead for benchmarking new cloud systems. Additionally, analyzing the CSUT at one scaling stage only, is faster and saves costs compared to analyzing all scaling stages. However, one should be aware of the mentioned linearity assumption when using the *Simple System Analysis*. If this assumption is not true for the evaluated system, the accuracy of the analysis results deteriorates.

Detailed System Analysis

The *Detailed System Analysis* implemented in the `DetailedSystemAnalysis` class realizes Algorithm 2. It evaluates each scaling stage separately until either adding a new resource does not result in increased load processing capabilities, or no additional resources are available. Furthermore, the analysis stops when the maximal resource amount specified by `setMaxResources()` is reached.

The *Detailed System Analysis* requires control over a cloud and has therefore a reference to a `cloudManager`, an instance of the `CloudManagement` interface. This interface is described in Section 6.1.7.

6.1.5 Benchmark Calibration: Load Profile Adjustment

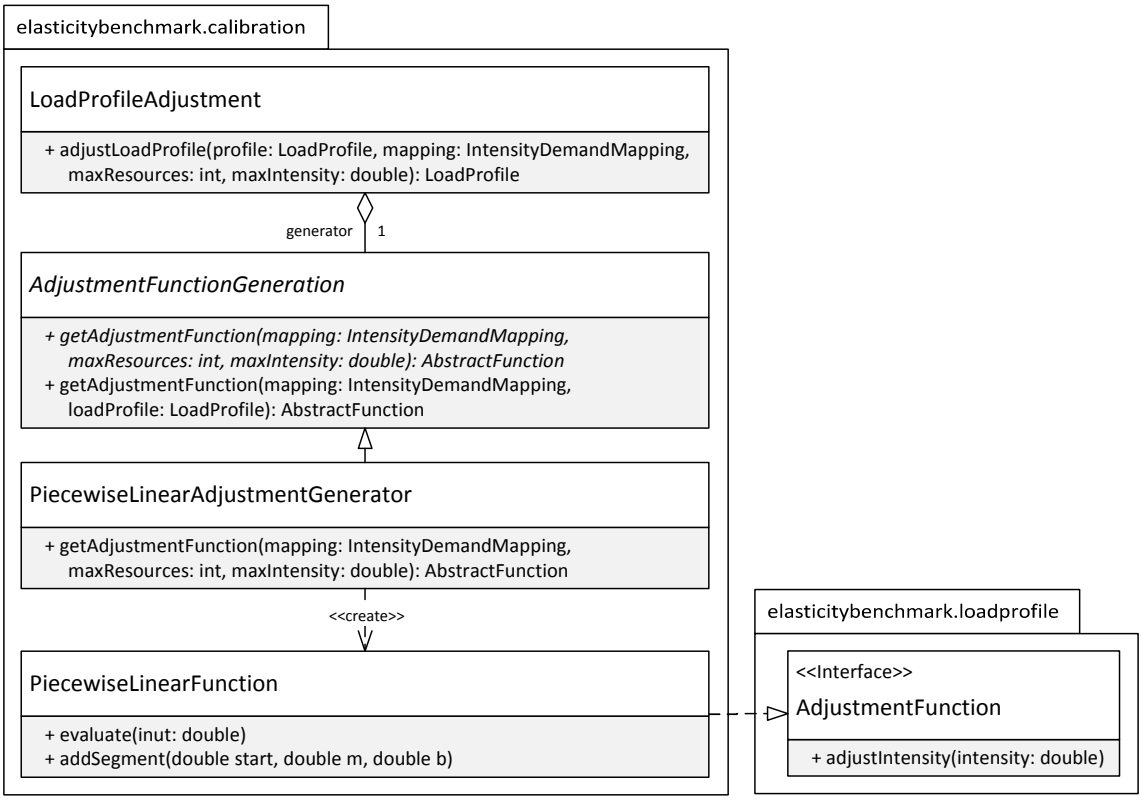


Figure 6.7: Class diagram for the `calibration` package

The load profile adjustment during the calibration assures, that the same resource demand is induced on different systems even when their underlying resources have different levels of efficiency or different scaling behaviors. Section 4.4.2 illustrated the concept for adjusting the intensities of a load profile in a system specific manner. As described in the mentioned section, the adjustment of a load profile for a specific system k can be specified by an adjustment function $adjustedintensity_k(intensity)$. Formula 4.1 shows how this function can be defined as a stepwise linear function.

The class `PiecewiseLinearAdjustmentGenerator` illustrated in Figure 6.7 implements the generation of a stepwise linear adjustment function according to Formula 4.1. Hereby, the parameters `mapping`, `maxResources` and `maxIntensity` of the `getAdjustmentFunction()` method correspond to $demand_k(intensity)$, n_{base} and $maxintensity_{base}$ in Formula 4.1.

The abstract super class `AdjustmentFunctionGeneration` additionally defines a helper method `getAdjustmentFunction()` which only needs a `mapping` and a `loadProfile` as parameters. This method calls the three parameter version of `getAdjustmentFunction()` and sets `maxResources` to the maximum resource amount specified in the `IntensityDemandMapping` and `maxIntensity` to the maximum intensity that occurs in the load profile described by the `intensityModel`. This is useful, when one want to adjust a load profile in a way, that all available resources have to be used for the intensity peaks of a load profile.

The design allows to extend the benchmark easily with respect to new kinds of adjustment functions and their generation.

6.1.6 Resource Allocations

Figure 6.8 shows a class diagram for the `allocation` package. This package contains data structures for storing series of allocations.

The base class `ResourceAllocation` stores the amount of resources allocated at a specific point in time. The abstract class `AllocationSeries` has a reference to a list of `ResourceAllocations`. It represents an allocation curve. Allocation curves can either be demand curves (class `DemandSeries`) or supply (class `SupplySeries`) curves. An allocation curve stored in a `SupplySeries` can be supplemented with a describing name. This is useful when different supply sources are monitored during a measurement.

The class `SeriesContainer` stores a demand curve and a list of supply curves. A `startMeasurement` member allows to define when the measurement started (the point in time when the warm up period ended). Thus, a `SeriesContainer` can cut allocation changes which occurred within the warm up period and return only allocation changes which occurred during the actual measurement period.

6.1.7 Cloud Information and Control

During measurement runs, the benchmark monitors the resource allocations of the CSUT. In order to do so, the benchmark requires an interface that allows to retrieve this information. Additionally, the `DetailedSystemAnalysis` class requires limited control over the CSUT. This again requires a standardized interface that allows to abstract from concrete cloud deployments and their cloud management software.

The `cloud` package, which is illustrated in Figure 6.9 depicts these interfaces and their implementations. The interface `CloudInfo` is the minimum interface that must be implemented in order to benchmark a cloud system. It just contains the method `getNumberOfResources()`. This method returns the current number of resources assigned to a CSUT. Using this method allows to monitor the resource allocations on the CSUT actively by calling it repeatedly during the measurement (*Active Monitoring*, compare Section 4.5.2). This monitoring behavior is implemented in the `ResourceMonitor` class.

Some cloud management systems log events about when the amount of allocated resources changes. If this event log is accessible, the resource allocations during a measurement run can be reproduced by taking just one absolute measurement with the `getNumberOfResources()` method at the beginning of a measurement. The allocation changes during the run itself can be parsed from the event log, when the measurement has

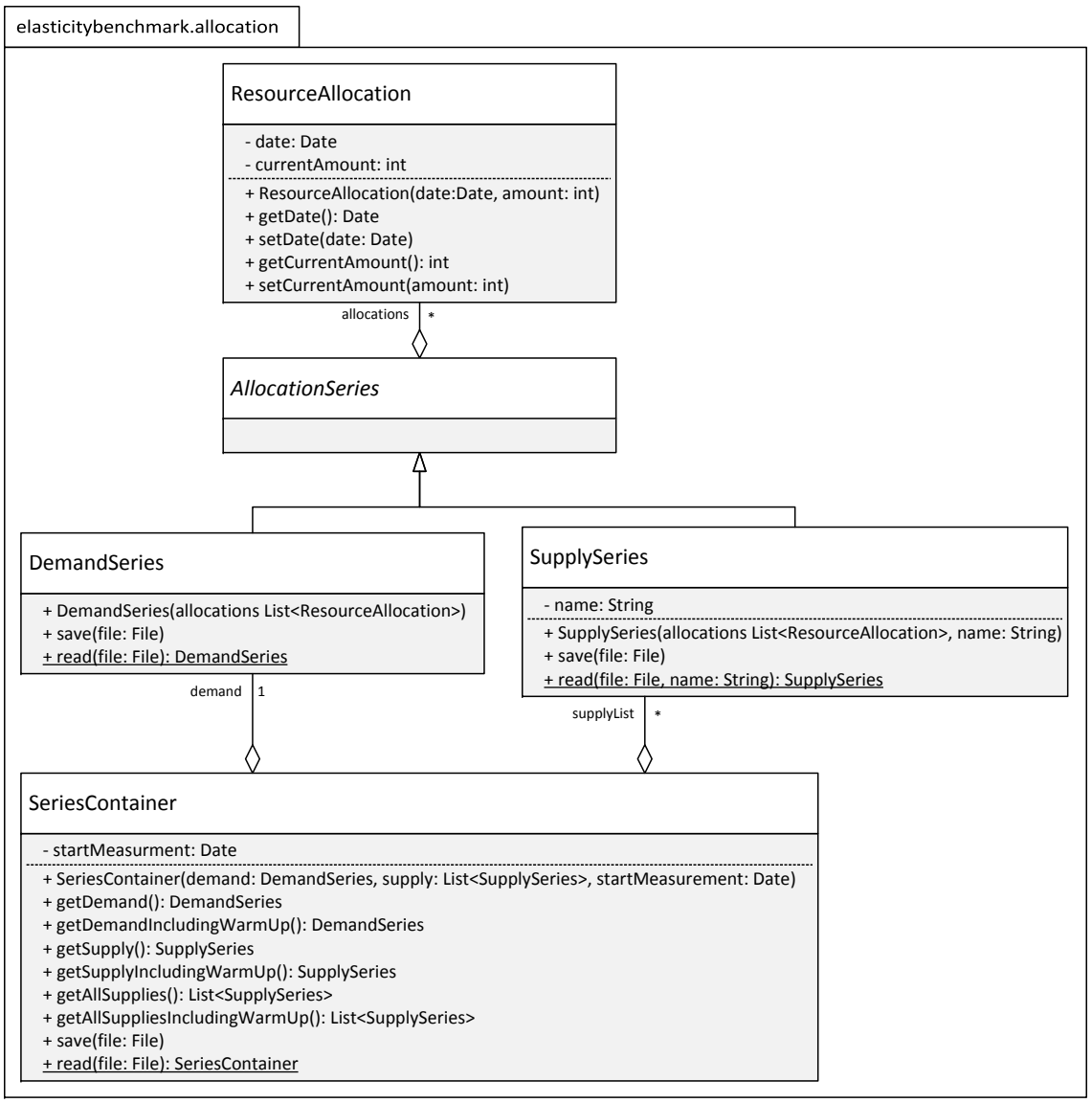


Figure 6.8: Class diagram of the `allocation` package

finished (*Passive Monitoring*, compare Section 4.5.2). This kind of measurement requires the implementation of the `ExtendedCloudInfo` interface. Since in some cases, there exist several events for the same resource allocation the `getResourceAllocations()` method returns a list of `SupplySeries`. A system which supports several event types can return separate `SupplySeries` for every event type.

In order to support the `DetailedSystemAnalysis`, the `CloudManagement` interface must be implemented. It allows to configure the minimum and maximum amount of resources that the CSUT is allowed to use.

At the moment, the benchmark harness supports to benchmark cloud systems that are based on either CloudStack or AWS. The benchmark can be extended to use other cloud management software by implementing at least the `CloudInfo` interface.

6.1.7.1 CloudStack

The class `CloudstackResourceInfo` in the `cloud.cloudstack` package implements the interface `ExtendedCloudInfo` and thereby allows to use the *Active Monitoring* technique as

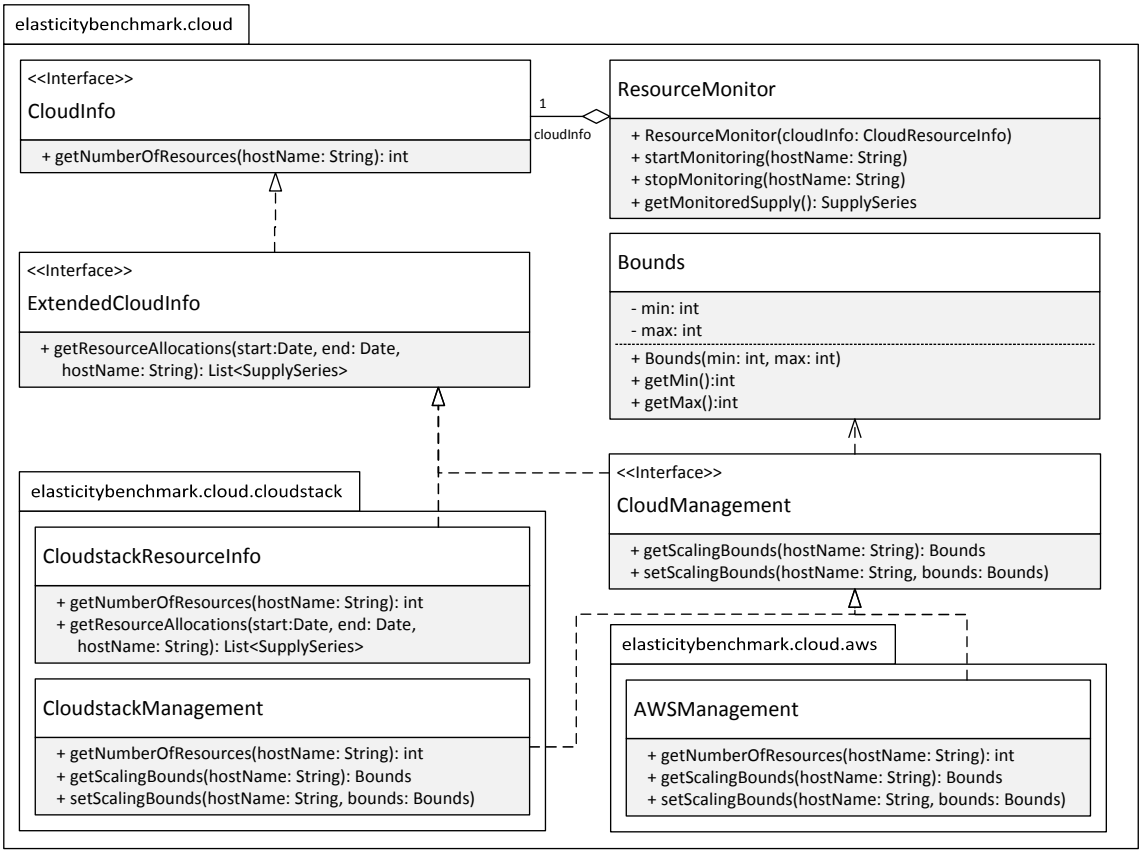


Figure 6.9: Class diagram for the `cloud` package

well as the *Passive Monitoring* technique during measurement runs. The implementation of the `CloudManagement` interface in the `CloudstackManagement` class allows to use the *Detailed System Analysis* in addition to the *Simple System Analysis* on CloudStack based CSUTs.

Monitoring Resource Suppply on CloudStack Clouds

Figure 6.10 illustrates how a resource allocation change is monitored differently when different monitoring techniques are applied and different supply event types are compared. The topmost graph shows how the resource demand increases after one minute and decreases again after eleven minutes. The undermost graph illustrates how the response time varies over time. The graphs in between show the monitored resource supply:

- The graph labeled *MONITORED* illustrates how the resource allocations on the cloud system changed according to the polled monitoring data. This graph shows the fastest reactions to the demand changes.
- The graph labeled *VM_SCHEDULED* illustrates how the resource allocations on the cloud system changed according to the CloudStack events that signal the scheduling of the creation/deletion of a VM instance.
- The graph labeled *VM_COMPLETED* illustrates how the resource allocations on the cloud system changed according to the CloudStack events that signal the completion of the creation/deletion of a VM instance.
- The graph labeled *LB_RULE_ADAPTION* illustrates how the resource allocations on the cloud system changed according to the CloudStack events that signal the adaptation of the load balancer.

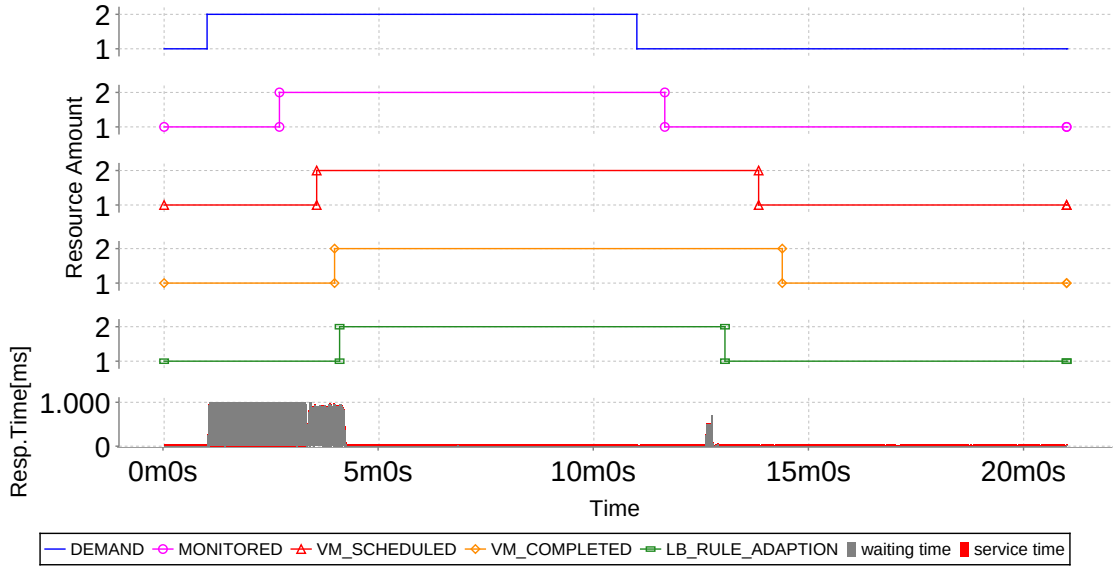


Figure 6.10: Resource demand (topmost graph) and different monitored CloudStack resource supply types

When evaluating the elasticity of the system, using different supply types leads to different metric results. In the opinion of the author, using the *LB_RULE_ADAPTION* curve is most appropriate for evaluating elasticity. A resource is only effective if the complete CSUT - this includes the load balancer - can make use of the resource. If an instance is created or even only scheduled, but the load balancer does not use it, the observed system behavior does not change. The response time graph confirms this: The response times decreases to a normal level just very shortly after the new instance was added to the load balancer.

6.1.7.2 Monitoring Resource Suppply on AWS Clouds

For AWS based clouds, the *Detailed System Analysis* as well as *Active Monitoring* is supported. The required interfaces are implemented in the `AWSManagement` class in the `cloud.aws` package.

AWS Supply Adaptation Events

The AWS API allows to query two different system properties that can be used to monitor the resource supply on AWS based CSUTs:

1. Number of VMs assigned to the load balancer
2. Number of VMs passing the load balancer's health check

Experiments showed that in scale up scenarios, monitoring the resource supply by querying Property 1 results in a supply curve which adapts well before the corresponding change of the response time is observed. This means VM instances are assigned to the load balancer before they are ready to use. Thus, the CSUT cannot make use of instances directly after their assignment to the load balancer.

When Property 2 is queried for monitoring the resource supply, the response time changes shortly after the supply curve adaptation. This observation for AWS based clouds is similar to that made for CloudStack based clouds when using the *LB_RULE_ADAPTION* event for monitoring the resource supply. Furthermore, monitoring the resource supply by querying Property 2 leads to a resource supply of zero instances as soon as no instance

passes the health check. In cases where it is known that instances are allocated, a supply of zero instances might seem wrong firstly. However, when no instance passes the health check, no request is forwarded to an instance by the load balancer and thus the CSUT cannot use the instances. Hence, querying Property 2 is reasonable for elasticity evaluations and is therefore used in *BUNGEE*.

6.1.8 Metrics

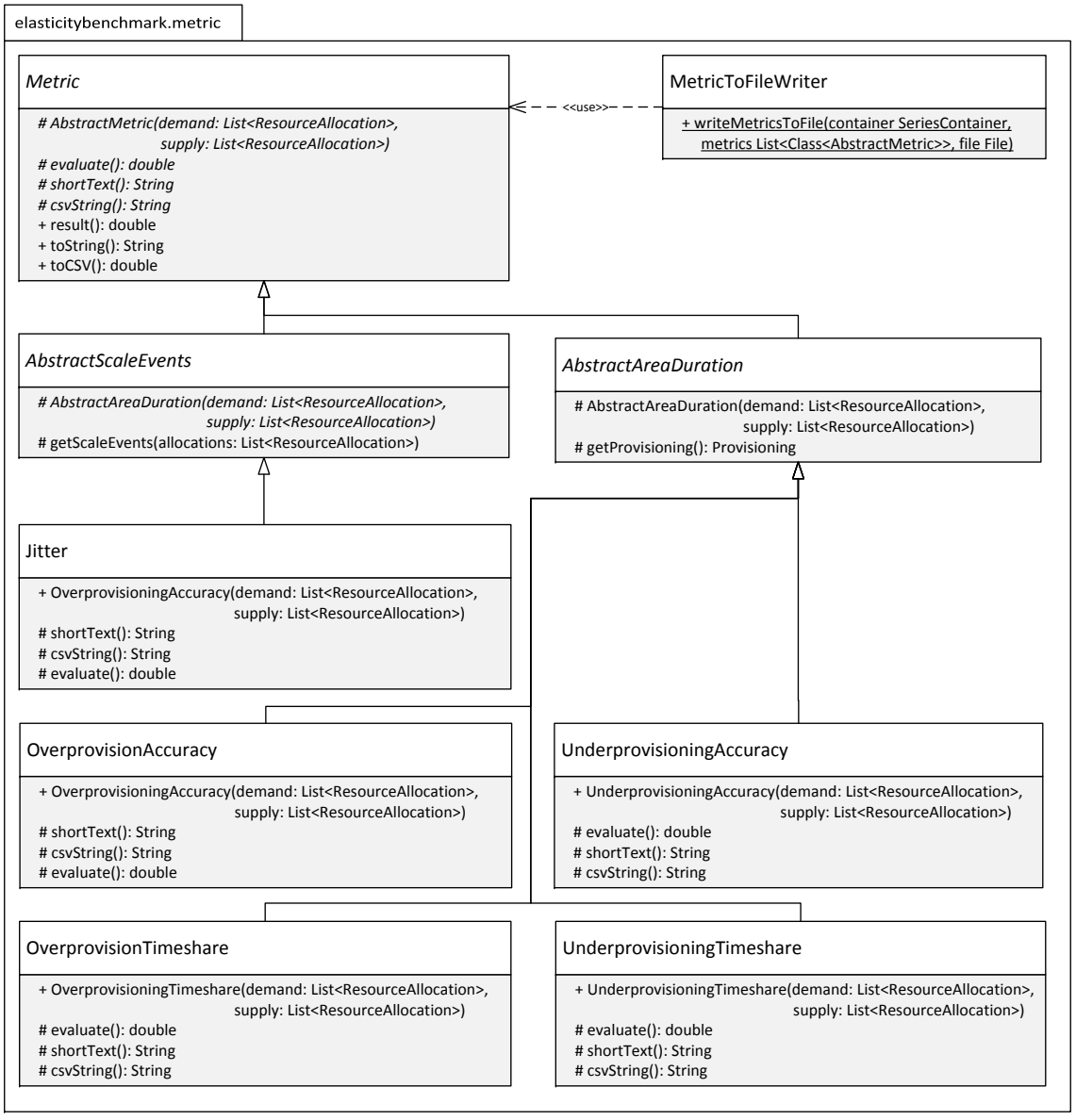


Figure 6.11: Class diagram for the `metric` package

Chapter 5 explained different elasticity metrics. These metrics are implemented in the `metric` package, which is illustrated in Figure 6.11.

All metrics inherit the abstract class `Metric`. It declares the protected abstract methods `evaluate()`, which returns the metric result, `shortText()`, which returns a string describing the numbers used to calculate the metric, and `csvString()`, which returns the information as `shortText()` formatted as CSV output. The public concrete methods `result()`, `toString()`, and `toCSV()` return the result of the corresponding abstract meth-

ods. The results are cached and thus a repeated invocation of the methods reuses the cached results without re-executing the abstract methods again.

The abstract subclasses `AbstractScaleEvents` and `AbstractAreaDuration` provide protected helper methods (not shown in the class diagram) which can be reused within several metric implementations.

The `MetricToFileWriter` class allows to evaluate the demand and the supply curve contained in a `DemandSeriesContainer` with the help of a list of metrics and write the results into a file.

6.1.9 Visualization

The benchmark harness offers different options for visualizing benchmarking input and output data. The corresponding generator classes can be found in the `chart` package. For the chart generation the chart library JFreeChart is used.

Load Profile and Induced Resource Demand

Load profiles are an important input for the benchmark. The LIMBO toolkit already allows to visualize load profiles easily within in the eclipse development environment. For benchmarking purposes, it is useful to visualize the resource demand that is induced by a load according to a load profile. The `ChartGenerator` class is able to create a chart that visualizes both, a load profile and the induced resource demand. As input, a `LoadProfile` instance and a `IntensityDemandMapping` are required. Figure 6.12 illustrates an exemplary chart.

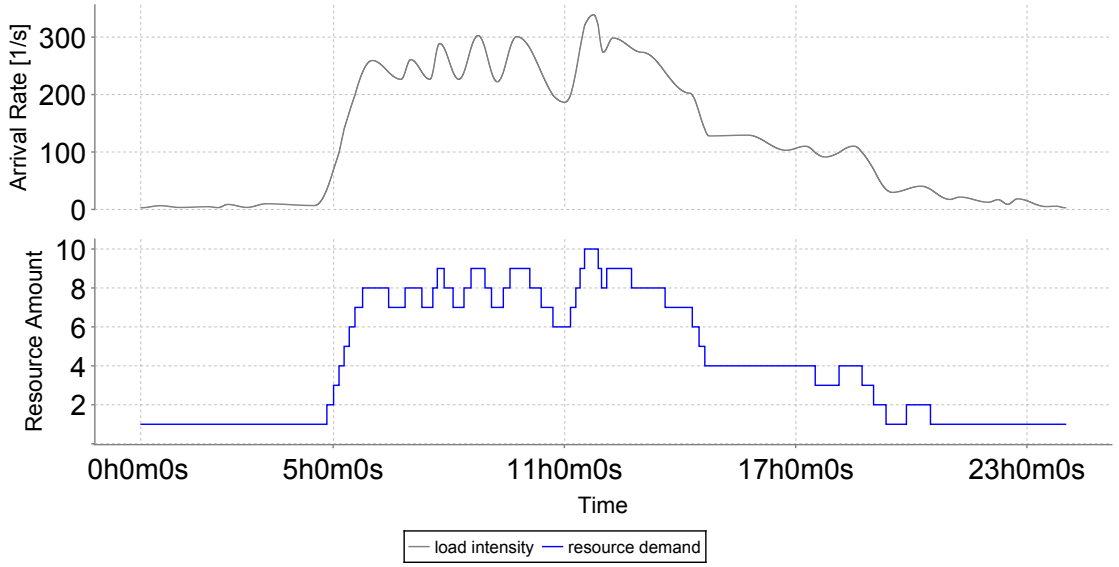
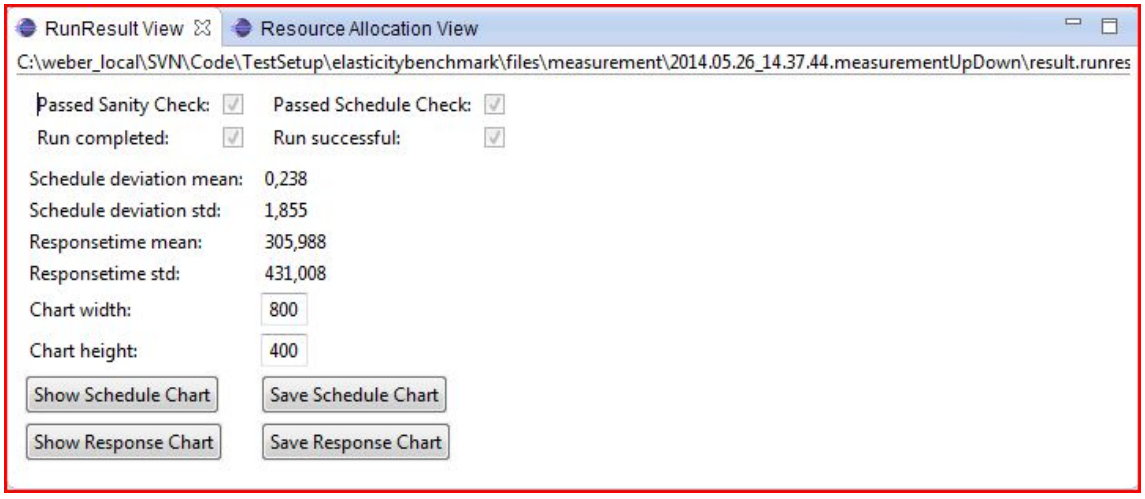


Figure 6.12: Load profile for a whole day and the corresponding induced resource demand

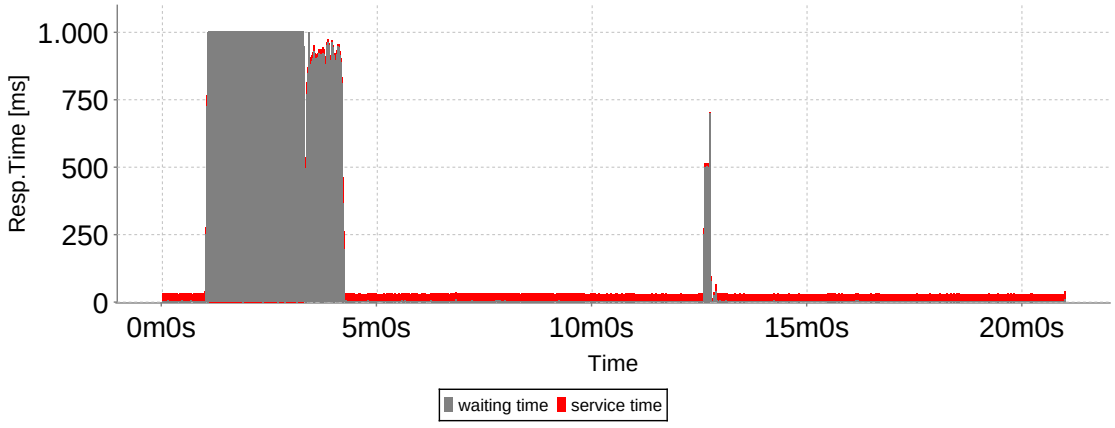
Run Result: Accuracy of Request Submission and Response Times

Within the benchmark harness, results of load executions are stored within `RunResult` objects. These objects can be persisted in a *.runresult file. The `ChartGenerator` class allows to generate charts that illustrate measurement run results. As shown in Figure 6.13, it is possible to visualize the request submission accuracy or the response times over time.

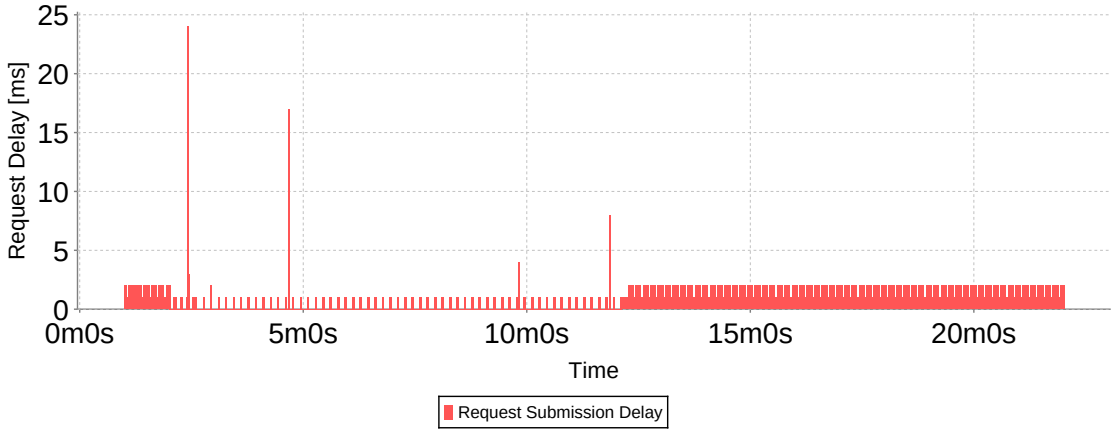
The evaluation of `RunResults` is further supported by an eclipse plugin. The plugin offers a view that shows different statistics about the run and allows to generate a schedule or a response chart.



(a) Eclipse view for viewing RunResults



(b) Response Chart



(c) Schedule Chart (Request Submission Accuracy)

Figure 6.13: An eclipse view shows the statistics for a measurement run (a) and allows to generate a schedule chart (b) or a response chart (c).

Resource Allocations

Measuring elasticity means comparing resource demand and supply curves with the help of metrics. While metrics provide numbers that describe a system, visualizations of demand and supply curves facilitate the understanding of these numbers. Visualizations can even help to get an intuitive understanding of the elasticity of a system. Therefore, the benchmark allows to generate visualizations of resource allocation curves, easily. Many examples for such visualizations can be found in Section (7.3).

6.2 Cloud-Side Load Generation

The CSUT must handle requests that are sent to it. Within the CSUT, typically a load balancer receives requests and forwards them to VM instances that process and generate responses for them. A load generation application accepts requests and generates load on the CSUT's resources. To allow elasticity benchmarking with BUNGEE, the load generation application must be deployed on every VM instance. This section describes the used load generation application.

6.2.1 Requirements

To ensure that the load generation application is suitable for elasticity benchmarking purposes, it was developed according to the following functional and non-functional requirements.

Functional Requirements:

- Processing of a request induces a demand on the CPU.
- Request parameters can be used to modify the demand size.
- The response contains information about the request service time.

Non-functional Requirements:

- The application should be platform independent to allow benchmarking in different system environments.
- Processing of a request should utilize as few resources as possible for other resource types.
- The application should be easily extensible to allow inducing demands on other resource types.
- The application should be stateless. Thus, the induced resource demand should be equal for fixed request parameters.

6.2.2 Implementation

The cloud-side load generation application is a java based HTTP server. This allows to easily deploy it on any platform with a support for java. The application uses Simple[8] (version 5.1.6) as a lightweight HTTP server framework.

Figure 6.14 shows a class diagram for the cloud-side load generation application. Its main class is the `WebServer` class. Whenever a request is received, the Simple framework calls the method `handle()` of the `WebServer` class. The parameters `request` and `response` give access to the request and its parameters and allow to specify and send a response. In order

[8] http://www.simpleframework.org

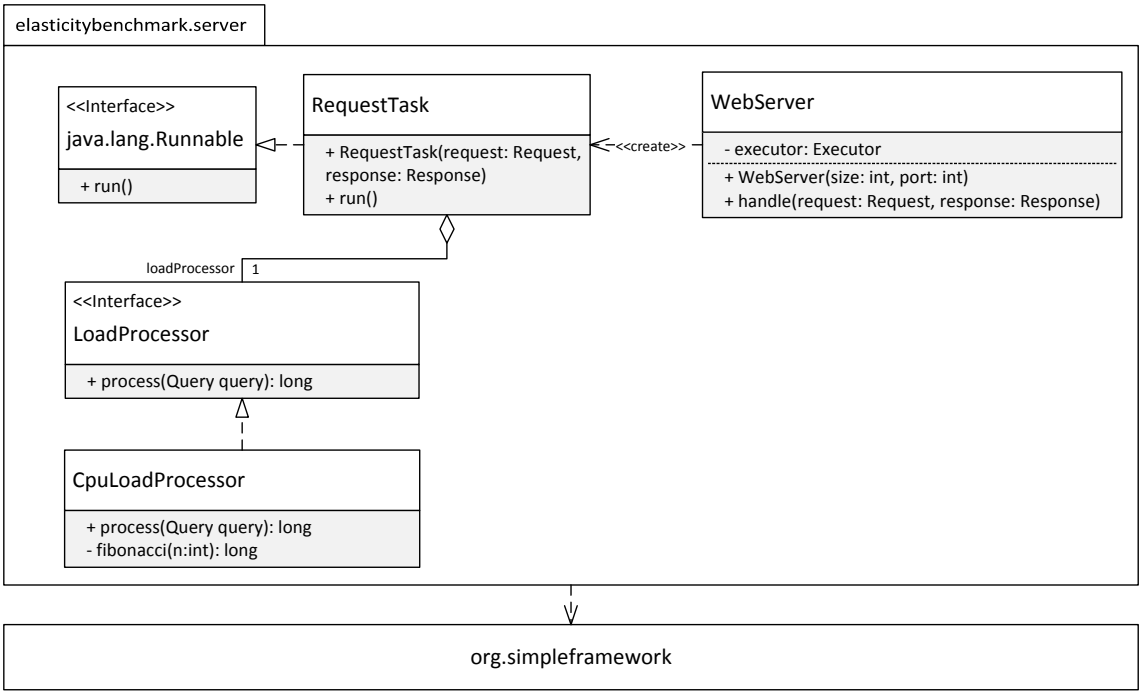


Figure 6.14: Class diagram for the cloud-side load generation application

to be able to process requests concurrently, the actual processing is done by instances of the `RequestTask` class.

For every request the `WebServer` creates a `RequestTask` object and assigns it to a thread pool (`executor`), which then calls the `run()` method of the `RequestTask` with an own thread. `RequestTask` calls the `process()` method of an `LoadProcessor` instance to induce a resource demand. After the processing is done, `RequestTask` creates a response which contains the following information:

- processing start time
- processing end time
- request service time (processing end time - processing start time)
- result returned by the `process()` method
- ip of the virtual machine

The response is sent using the `response` object that was passed by the Simple framework in the `handle()` call.

A sequence diagram that illustrates the handling of a request within the load generation application as described above is shown in Figure 6.15.

For this thesis, `CPULoadProcessor` is a sample class, which induces a demand on the CPU by computing the n-th element of the fibonacci series. n can be specified by a size parameter in the HTTP request query. As discussed in Section 4.3.1, the computation is done in iterative manner not recursive manner in order to minimize memory consumption. Furthermore, result caching and java compiler optimizations are avoided by adding random numbers within each calculation step. This randomization of the computation was previously used in elasticity benchmarking experiments for thread pools [Her11].

The load processing application can be extended easily for other resource types, such as memory resources, by implementing the `LoadProcessor` interface.

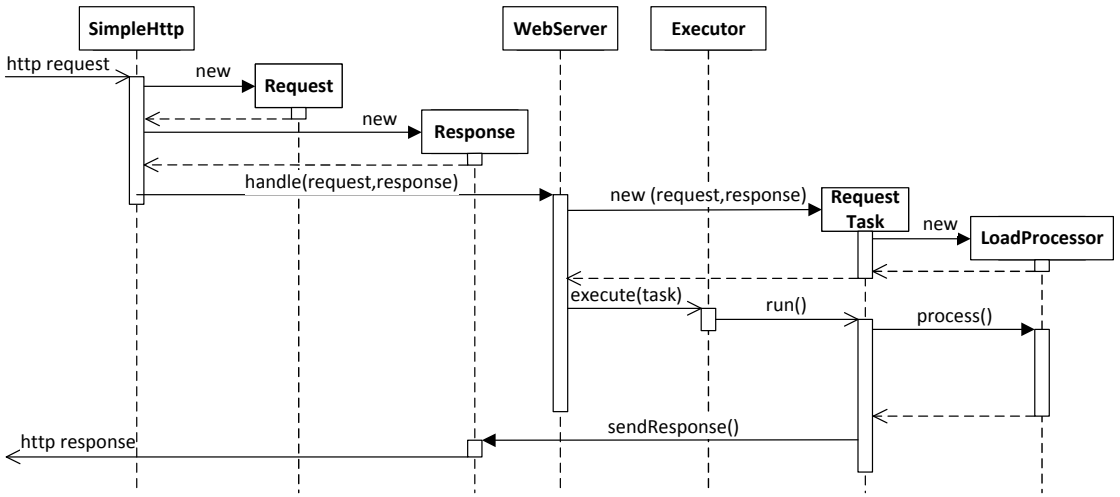


Figure 6.15: Request processing within the cloud-side load generation application

6.3 Conclusion

This chapter explained the benchmarking framework implementation. The architecture and functionality of the harness as well as of the cloud-side load generation application was discussed. Extensibility allows to support other resource types, further cloud management software, or addition of new metrics as part of future work.

7. Evaluation

This chapter evaluates the *System Analysis* and the used metrics and thus addresses the sixth goal mentioned in Section 1.1.

The general experiment setup used for the evaluation of the benchmark on a private cloud is explained in Section 7.1. The *System Analysis* is then evaluated in Section 7.2. It is followed by the evaluation of the metrics in Section 7.3. Section 7.4 demonstrates the benchmarking capabilities for a realistic load profile and a realistic number of resources on the private cloud as well as on a public cloud.

7.1 Experiment Setup

This section describes the experiment setup that is used for the evaluation. The test cloud system is a private cloud that is capable of scaling virtual machines horizontally. Thus, the resource type for the test system is virtual machines i.e., container resources. As discussed in 4.1, this thesis focuses on CPU-bound resources. Therefore, the rather complex behavior of a container resource is abstracted by using a CPU-bound load. The following subsections explain the cloud setup in detail.

7.1.1 Private Cloud Deployment

Figure 7.1 illustrates the experiment setup that is used for the evaluation. It consists of three nodes: The infrastructure node, the management node, and the load driver and benchmark node. The first two nodes form the CSUT that is benchmarked by the third node.

The infrastructure node provides fully virtualized resources via a hypervisor. In this experiment setup, the infrastructure node has a Quad CPU AMD Opteron 6174 with 48 cores at 2.2 GHz and 256 GB RAM. On this hardware, XenServer 6.2 is running as a hypervisor. The management of cloud infrastructure services on top of the hypervisor is managed by a cloud management software running on the management node.

The cloud management software and the load balancer are deployed on the management node. Both run in separate VMs on top of another XenServer 6.2 as hypervisor. Since the management node only runs two VMs, less powerful hardware is used: An Intel Core i7 860 with 8 cores at 2.8 GHz and 8 GB RAM. For this experiment setup, CloudStack 4.2 is used as cloud management software. CloudStack is installed on a CentOS 6.5 guest

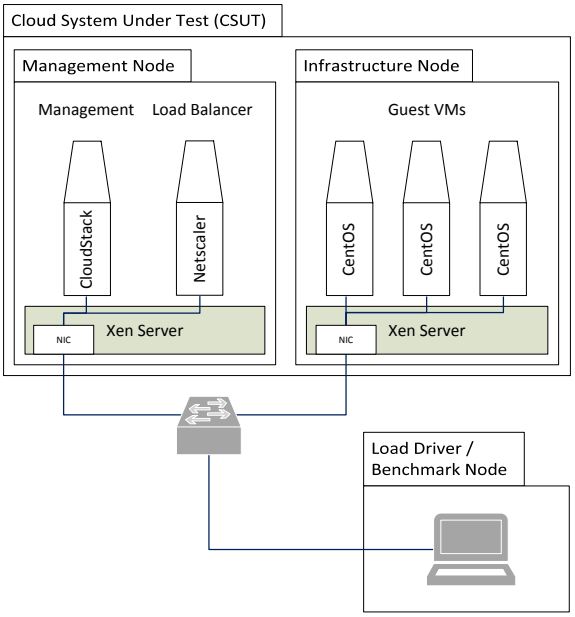


Figure 7.1: Experiment Setup

virtual machine. The load balancer virtual machine runs Citrix Netscaler 10.1 VPX 1000 as software load balancer. Since CloudStack supports Netscaler as an external load balancer, load balancing can be configured within CloudStack.

The load driver and benchmark node runs the benchmark harness and the load driver. Both are java based and can therefore run on any system with java support. For this experiment setup the load driver and benchmark node runs on a Windows Vista Desktop PC with a Dell Precision T3400 (4 x 2.5 GHz) and 8 GB RAM.

The three nodes are physically connected over a 1 GBit ethernet network.

Clock synchronization is ensured by using the Network Time Protocol (NTP). The NTP daemons of all nodes are configured to sync with a Stratum 3 NTP server located in the same network.

7.1.2 Elastic Cloud Service Configuration

This subsection describes the basic configuration of the test system for the evaluation. The Sections 7.2 and 7.3, which evaluate the calibration and the metrics, use the parameters that are described here and mention parameters explicitly only when they are changed for a particular evaluation experiment.

Virtual Machine Setup

The basic virtual machine template for virtual machines that are deployed on the infrastructure node by CloudStack bases on Cent OS 5.6 as operating system with java runtime environment and Simple Network Management Protocol (SNMP) installed. Java is necessary to run the cloud-side load generation application described in Section 6.2, SNMP provides the elasticity mechanism access to resource utilization information. The VM template is configured to start the load generation application on every start up. Thus, it is available without any interaction as soon as the template is deployed and booted.

CloudStack Configuration

CloudStack offers two different options for the network configuration: *Basic* and *Advanced Network Setup*. While the second option offers features like separating different traffic types such as management traffic, guest traffic, and public traffic physically or via VLANs, only the *Basic Network Setup* allows to configure auto-scaling. Thus, CloudStack is configured to use the *Basic Network Setup* which means that all traffic types share the same network.

For the experiments, four different service offerings, which define how much virtualized resources should be assigned to a virtual machine, are used. All offerings provide 1 GB RAM and just local storage, since the load is CPU-bound. The assigned virtual CPUs are set to run at 2.2 GHz for three of the offerings. The smallest service offering is configured to use just one CPU (Offering A), a second offering uses two CPUs (Offering B), and a third offering uses four CPUs (Offering C). A fourth service offering uses one virtual CPU that is set to run at 1024 MHz (Offering D). If not stated otherwise, Offering A is used.

Elasticity Parameters

CloudStack allows to configure a rule based elasticity mechanism. This section explains the different parameters that influence the behavior of the elasticity mechanism.

Table 7.1 shows an overview about the parameters. The column *Default* shows the standard values for the evaluation experiments. When a parameter is changed for a specific evaluation experiment, this is mentioned in the description of this evaluation.

The parameters *minInstances* and *maxInstances* specify how many resources should be allocated at minimum and at maximum, respectively. The de-/allocation of new VMs is triggered by specific rules. The rules base on conditions that are evaluated frequently. The parameter *evalInterval* specifies this frequency. When the number of VMs has changed, it may take some time until the new configuration takes effect. Therefore, the *quietTime* allows to specify a period of time for which the scale up/down rules are not evaluated after a change. If a VM is deallocated, it will not handle new connections any more. Still, there may be old connections which have not been processed completely. CloudStack allows the VM to process and close these old connections within in a certain period of time. This period is determined by the *destroyVmGracePer* parameter.

For both scaling directions, four additional parameters define when the de-/allocation of a virtual machine is triggered. The parameter *condTrueDurUp/Down* defines how long a condition has to be true in order to trigger the de-/allocation of a VM. The condition itself is expressed by the parameters *counterUp/Down*, *operatorUp/Down* and *thresholdUp/Down*. *CounterUp/Down* describes the quality measure that is compared with the threshold. Possible measures are User CPU utilization, CPU Idle time, or response time for example. The *operatorUp/Down* parameter defines if the *thresholdUp/Down* parameter is an upper or lower threshold. Finally, the *thresholdUp/Down* defines the threshold itself.

Health Check Parameters

CloudStack allows to configure a health check additionally to basic elasticity parameters mentioned in the last paragraph. A health check determines, if a VM instance is considered as healthy. The load balancer forwards requests only to healthy instances. If an instance is unhealthy, the allocation of a substitute instance can be triggered.

Table 7.2 illustrates the parameters for health checks. As for the elasticity parameter table, the column *Default* shows the standard values for evaluation experiments.

The parameter *pingPath* specifies to which address health check requests are sent. A health check request is considered as successful, if the instance answers the request within time

	Name	Default	Description
General	minInstances	1	minimum number of VM instances
	maxInstances	2	maximum number of VM instances
	evalInterval	5s	frequency at which the conditions for the scale up down rules are evaluated
	quietTime	300s	period for which the policy is not evaluated after the number of instances changed
	destroyVmGracePer	30s	time allowed for existing connections to get closed before a VM is destroyed
Scale up	condTrueDurUp	30s	duration for which the condition has to be true before a scale up is triggered
	counterUp	User CPU	quality measure which is compared with *thresholdUp*
	operatorUp	GT	operator which compares *counterUp* and *thresholdUp*. Values: GT (greater then), LT (less then)
	thresholdUp	90%	threshold for for *counterUp*
Scale down	condTrueDurDown	30s	duration for which the condition has to be true a before scale down is triggered
	counterDown	User CPU	quality measure which is compared with *thresholdDown*
	operatorDown	LT	operator which compares *counterDown* and *thresholdDown*. Values: GT (greater then), LT (less then)
	thresholdDown	50%	threshold for for *counterDown*

Table 7.1: Cloudstack elasticity parameters

Name	Default	Description
pingPath	/?size=1	address which is queried to check the instance health
healthyResponseTimeout	1s	period of time within that a response is expected from healthy instances
healthCheckInterval	5s	time between two consecutive health checks
healthyThreshold	1	number of subsequent health checks request successes before instance is declared healthy
unhealthyThreshold	4	number of subsequent health checks request failures before instance is declared unhealthy

Table 7.2: Cloudstack health check parameters

defined by the *healthyResponseTimeout* parameter. The parameters *healthyThreshold* and *unhealthyThreshold* define the number of subsequent health check request successes and failures respectively before an instance is declared as healthy or unhealthy.

The selected default values tend to declare instances healthy early and unhealthy late. These values were chosen in order to prevent unnecessary allocation of new instance for the evaluation. However, in a real world scenario a more conservative health check (*healthyThreshold* = 2, *unhealthyThreshold* = 1) would be more appropriate in most cases.

7.1.3 Benchmark Harness Configuration

The benchmark harness offers several configuration options that allow to configure it according to the means of the targeted domain. Table 7.3 shows the different parameters and the default values which where used for this evaluation. The following paragraph explains the different parameters.

Name	Default
size	50000
requestTimeout	1000ms
SLO	95% of all requests must be processed successfully within a maximum response time of 500ms.
$warmup_{calibration}$	180s
$warmup_{measurement}$	300s

Table 7.3: Benchmark harness parameters

Amount of Work per Request

The amount of work which is executed within each request is defined by a *size* parameter, which is send with each request. It is set to 50000 for the evaluation. This means each request issues a randomized calculation of the 50000th element of the fibonacci series (compare Section 4.3.1).

Service Level Objective and Request Timeout

During the calibration phase, the benchmark needs a service level objective in order to perform the *System Analysis*. For the evaluation the following service level objective was used: **SLO:**
"95% of all requests must be processed successfully within a maximum response time of 500ms."
Additionally the benchmark has a *requestTimeout* parameter. This parameter defines, how long the benchmark waits for a response before the connection is aborted. For this evaluation *requestTimeout* is set to 1000ms.

Warm up Times and Calibration Duration

When a cloud system is exposed to a load it may behave different at the beginning due to some initialization overhead. Therefore, the benchmark allows to define warm up periods for the calibration and the measurement phase. During warm up periods, the benchmark sends request to the cloud system, but no measurements are taken. The load intensity for the warm up requests is always the first intensity that occurs in the used load profile. The warm up period $warmup_{calibration}$ for the calibration precedes every measurement within the *System Analysis*. For this evaluation, it was set to 180 seconds. The warm up period $warmup_{measurement}$ precedes the every benchmark measurement run. The $warmup_{measurement}$ is set to 300 seconds for this evaluation.

7.1.4 Evaluation Automatization

In order to facilitate the evaluation, the different experiments for the calibration and the metric evaluation are designed to run automatically. For every experiment an appropriate cloud setup is created and configured programmatically using the CloudStack API before the actual experiment starts. When CloudStack reports that the setup was created successfully, the evaluation mechanism waits ten more minutes to allow the system to initialize properly. After this time period, the warm up period begins. When the measurements are finished, the cloud setup is removed again. This approach ensures that measurements are taken under equal conditions.

7.2 Analysis Evaluation

Two different activities precede the actual elasticity measurement: The *System Analysis* and the *Benchmark Calibration*. The latter depends on the correctness of the *System Analysis*. The *System Analysis* is therefore evaluated with respect to two different aspects:

- Is the result of the *System Analysis* reproducible on the test system? (RQ 6.1)
- What is the deviation between the results of *Detailed System Analysis* and the results of *Simple System Analysis*, which assumes a linearly increasing resource demand? (RQ 6.2)

7.2.1 Reproducibility

This subsection tests the following hypothesis:

Hypothesis 1 *Under the assumption that the analysis result follows a normal distribution, the error of the System Analysis for the first scaling stage is smaller than 5% on a confidence level of 95%.*

The reproducibility of the *System Analysis* is analyzed for three different system configurations that are different with respect to the processing efficiency of the underlying resources. To obtain different levels of efficiency for resource instances (VMs), CloudStack is configured to use service offerings that either assign one (Offering A), two (Offering B), or four (Offering C) virtual CPUs to the created VMs. The evaluation is conducted for every configuration separately.

Let $X_i \in N(\mu, \sigma)$ be the samples of the result of the analysis and n be the number of samples. The sample mean $\overline{X}$ can be expressed as $\overline{X} = \frac{\sum X_i}{n}$. The sample standard deviation S can be expressed as $S = \frac{\sum (X_i - \overline{X})^2}{n-1}$.

To show:

$$\begin{aligned} &P(c_1 \leq \mu \leq c_2) \leq 1 - \alpha \\ &\text{with} \\ &c_1 = 0.95 * \bar{X},\ c_2 = 1.05 * \bar{X},\ \alpha = 0.05 \end{aligned} \tag{7.1}$$

It can be shown [Man64] that $T = \frac{[\overline{X} - \mu]\sqrt{n}}{S}$ has a t-distribution with $(n-1)$ d.f.

It follows:

$$\begin{aligned} &P(c_{low} \leq \mu \leq c_{high}) \leq 1 - \alpha \\ &\text{with} \\ &c_{low} = \overline{X} - t_{1-\alpha/2;n-1} * S/\sqrt{n} \\ &c_{high} = \overline{X} + t_{1-\alpha/2;n-1} * S/\sqrt{n} \end{aligned}$$

where $t_{1-\alpha/2;n-1}$ is the upper $(1-\frac{\alpha}{2})$ critical point of the **t** distr. with n-1 d.f.

To prove claim 7.1, it will be shown that

$$c_{low} = \overline{X} - t_{1-\alpha/2;n-1} * S/\sqrt{n} \geq 0.95 * \bar{X} = c_1 \tag{7.2}$$
$$c_{high} = \overline{X} - t_{1-\alpha/2;n-1} * S/\sqrt{n} \leq 1.05 * \bar{X} = c_2$$

holds true for a set of n scaling analysis samples.

Off.	Analysis Samples [req./sec.]										$\overline{X}$	S	c_1	c_{low}	c_{high}	c_2
A	35	35	35	35	35	35	35	35	35	35	35.0	0.00	33.25	35.00	35.00	36.75
B	55	57	56	56	56	56	58	55	57	56	56.2	0.92	53.39	55.54	56.86	59.01
C	97	101	100	100	97	99	98	101	99	101	99.3	1.57	94.34	98.18	100.42	104.27

Table 7.4: Results of the reproducibility evaluation for the *System Analysis*

Table 7.4 shows the result of the *System Analysis* for the first scaling stage for $n = 10$ measurement samples and three different system configurations. For all configurations the equations 7.2 are true. Thus, it is not possible to reject Hypothesis 1.

7.2.2 Linearity Assumption

The *Simple System Analysis* only analyzes the load processing capabilities of the first scaling stage. It then assumes, that the resource demand increases linearly with the load intensity and therefore creates a mapping with steps of equal length, like illustrated in Figure 7.2. For the system depicted in Figure 7.2, only the load processing capability - here 25 - for one resource was determined by the *Simple System Analysis*. The load processing capabilities for two, three and four resources - here 50, 75, and 100 - are extrapolated based on the linearity assumption. Due to some overhead, e.g., overhead in the load balancer when

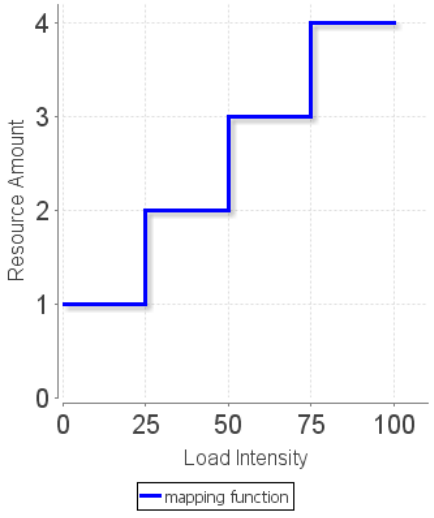


Figure 7.2: System with linear increasing resource demand

four resources are used, it may be that the real load processing capability does not equal the one extrapolated based the linearity assumption. This evaluation illustrates how big the error of not measuring the load processing capabilities for more than one resources is on the used test system.

Hypothesis 2 *The test system's resource demand scales linearly with the load intensity in a scale out scenario.*

It is notable that this hypothesis is not about a property of the benchmark but about a property of the test system.

To test the hypothesis for a given system configuration, two measurements are taken. First, the *Simple System Analysis* is used to get the load processing capability $i_{unscaled}$ for one resource. Then, system is scaled manually to use n resources. Now, the load processing capability i_{scaled} for the scaled system is determined. In the next step, i_{scaled} is compared with the extrapolated intensity $i_{extrapolated}$ for n resources. Hereby, $i_{extrapolated}$ is calculated as $i_{extrapolated} = n * i_{unscaled}$. Using this numbers, the absolute deviation dev_{abs} and the relative deviation dev_{rel} for the test system are:

$$dev_{abs} = i_{extrapolated} - i_{scaled}$$
$$dev_{rel} = \frac{dev_{abs}}{i_{scaled}}$$

This analysis has been conducted for two different system configurations that are different with respect to the levels of efficiency of the underlying resources. To obtain different levels of efficiency for resource instances (VMs), CloudStack has been configured to use service offerings which either assign one (Offering A) or two (Offering B) virtual CPUs to the created VMs.

The result of this evaluation is shown Table 7.5 for Offering A and in Table 7.6 for Offering B respectively.

For Offering A, the maximum intensity i_{scaled} was analyzed for all system configurations from n = 1..18 resources. The result is shown in Table 7.5.

n	$i_{unscaled}$ [req./sec.]	$i_{extrapolated}$ [req./sec.]	i_{scaled} [req./sec.]	dev_{abs} [req./sec.]	dev_{rel} [%]
2	35	70	70	0	0.0
3	35	105	99	6	6.1
4	35	140	140	0	0.0
5	35	175	169	6	3.6
6	35	210	199	11	5.5
7	35	245	239	6	2.5
8	35	280	284	-4	-1.4
9	35	315	320	-5	-1.6
10	35	350	339	11	3.2
11	35	385	359	26	7.2
12	35	420	399	21	5.3
13	35	455	462	-7	-1.5
14	35	490	479	11	2.3
15	35	525	519	6	1.2
16	35	560	564	-4	-0.01
17	35	595	598	-3	-0.01
18	35	630	633	-3	-0.01

Table 7.5: Linearity analysis for Offering A

Within this experiment, the measured deviation from the linearity assumption is always below 10%. Since the accuracy of measurement itself is limited (95% confidence for relative accuracy of ±5%, compare Section 7.2.1), it can be assumed that the measured deviation is mainly due to an inaccurate measurement.

For the first twelve resource scaling stages, the linearity analysis has been conducted three times. It is notable that these measurements have generated the same small deviations from the linearity assumption consistently. For six resources for example, all three measurements have returned 199 as measured maximum intensity i_{scaled}. Since the extrapolated maximum intensity $i_{extrapolated}$ for six resources is 210, this means a relative deviation dev_{rel} of 5.5%. This observation contradicts the assumption that the measured deviations from the linearity assumption are mainly due to an inaccurate measurement. It rather indicates that the deviations from the linearity assumption are mainly system specific.

For Offering B, the maximum intensity i_{scaled} has been analyzed for all system configuration from $n = 1..22$ resources. For all scaling stages, maximum intensity i_{scaled} has been measured three times. Table 7.5 shows the averaged results.

n	$i_{unscaled}$ [req./sec.]	$i_{extrapolated}$ [req./sec.]	i_{scaled} [req./sec.]	dev_{abs} [req./sec.]	dev_{rel} [%]
2	58	116	115.7	0.3	0.3
3	58	174	225.7	4.0	2.4
4	58	232	225.7	6.3	2.8
5	58	290	282.0	8.0	2.8
6	58	348	336.3	11.7	3.5
7	58	406	390.3	15.7	4.0
8	58	464	449.0	15.0	3.3
9	58	522	503.3	18.7	3.7
10	58	580	559.3	20.7	3.7
11	58	638	613.3	24.7	4.0
12	58	696	670.3	25.7	3.8
13	58	754	718.0	36.0	5.0
14	58	812	773.0	39.0	5.0
15	58	870	829.0	41.0	4.9
16	58	928	881.3	46.7	5.3
17	58	986	923.3	53.7	5.8
18	58	1044	1003.3	40.7	4.1
19	58	1102	1044.7	57.3	5.5
20	58	1160	1099.7	60.3	5.5
21	58	1218	1154.0	64.0	5.5
22	58	1276	1202.3	73.7	6.1

Table 7.6: Linearity analysis for Offering B
(i_{scaled} is averaged over three independent analysis runs)

As for Offering A, the deviation from the linearity assumption is below 10% for all analyzed scaling stages. Furthermore, the deviation tends to increase slowly with the number of used resources. A possible explanation for this increasing deviation is overhead within the hypervisor or the load balancer. Note that for the largest scale out scenario 92% of the underlying hardware resources of the test system have been used.

Hypothesis 2 holds for the tested scale out scenarios to a limited extend. Although the deviation from the linearity assumption is always below 10%, small deviations have been observed consistently. If possible, the usage of the *Detailed System Analysis* should therefore be preferred for new systems.

7.2.3 Discussion

The evaluation of the *System Analysis* demonstrated the reproducibility of its results for three system configurations. Within a second evaluation, it was shown that for scale-outs of up to 22 resources the linearity assumption holds for the test system to a limited extend. Constant small deviations from the linearity assumption have been observed for different scaling stages, consistently. The usage of the *Detailed System Analysis* is therefore preferred.

During the evaluation period, several updates were installed on the hypervisor. The complete reproducibility evaluation and the evaluation of the linearity assumption for Offering A were conducted before, the linearity evaluation for Offering B has been conducted after the installation of hypervisor updates. An additional analysis for Offering A after the installation of updates on the hypervisor has not shown the deviations illustrated in Table 7.5 anymore. This change in the observed scaling behavior signifies that it is important to reanalyze the scaling behavior after any direct or indirect change of the tested system.

7.3 Metric Evaluation

This section evaluates, if the accuracy and timing metrics shown in Chapter 5 allow to rank systems of different degrees of resource elasticity on an ordinal scale and thereby answers RQ 6.3. Every metric is evaluated with the help of a simple load profile which induces demand changes that are appropriate for evaluating the elasticity aspect measured by the respective metric. The metrics are then evaluated for systems that exhibit different degrees of elasticity. If the metrics reflect the different degrees of elasticity with a correct ranking according to the evaluated aspect, the ordinal character of the metrics is demonstrated. In order to induce system behaviors that exhibit different degrees of elasticity, the parameters of the elasticity mechanism of the test system are varied.

For comprehensibility reasons, the load profiles illustrated in this section are unadjusted load profiles. In the illustrations load intensity 100 is the maximum intensity one resource can withstand. However, in the calibration step the real intensity is adjusted in a way that the resource demand on the test system equals the resource demand in the illustrations.

In the following the evaluation for each metric is explained.

7.3.1 Under-provision Accuracy: $accuracy_U$

- Hypothesis

  Hypothesis 3 *The metric $accuracy_U$ allows to rank systems with different degrees of elasticity on an ordinal scale.*

- Load Profile

  The under-provision accuracy metric is evaluated with the load profile illustrated in Figure 7.3. The load profile starts with an intensity that is just a little bit below the maximum intensity for two instances. After five minutes, the intensity changes stepwise - every five minutes - to lower load intensities. However, the last intensity

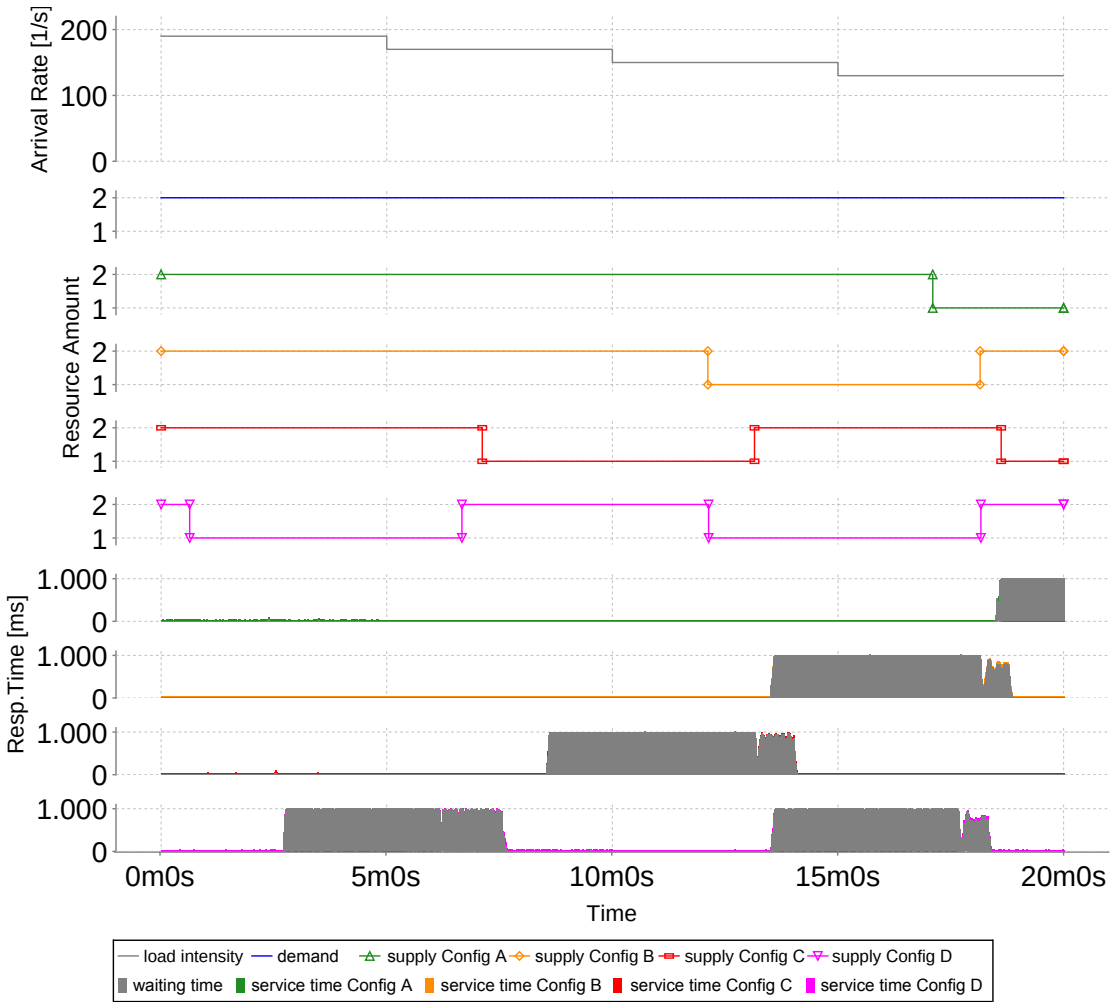


Figure 7.3: Evaluation of the $accuracy_U$ metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing *thresholdDown* values (A: 55%, B: 65%, C: 75%, D: 85%).

is still high enough to require two resources. A system with a low degree of elasticity may deallocate the second resource because of the shrinking demand although it is still needed. Thus, such a system lacks in accuracy.

- Different Degrees of Elasticity
  The degree of elasticity of the test system is varied by changing the *thresholdDown* parameter. For increasing values of this parameter the system tends to deprovision resources to early, which leads to lower degrees of elasticity. The $accuracy_U$ metric is evaluated for the following *thresholdDown* values: 55% (Configuration A), 65% (Configuration B), 75% (Configuration C) and 85% (Configuration D).

- Results
  Figure 7.3 shows the elasticity behaviors for different values of *thresholdDown*. It can be seen that due to lower degrees of elasticity for increasing *thresholdDown* values, the amount of under-provisioned resources increases. This is reflected by the metric results shown in Table 7.7. For decreasing degrees of elasticity, that means for increasing values of *thresholdDown*, the $accuracy_U$ metric increases. Thus, the $accuracy_U$ allows to rank elastic systems on an ordinal scale as stated in Hypothesis 3.

thresholdDown [%]	55	65	75	85
$accuracy_U$ [*resource units*]	0.145	0.302	0.371	0.603

Table 7.7: Measurement results for the $accuracy_U$ metric

- Remarks
  Although the systems deprovision the second resource to early, they reallocate it again after a while. This behavior is due to the fact, that for every arrival rate above 100, the CPU load is very high if just one resource is used. Thus, the scale up rule triggers the allocation of a new resource shortly after each deallocation.

7.3.2 Over-provision Accuracy: $accuracy_O$

- Hypothesis

  Hypothesis 4 *The metric $accuracy_O$ allows to rank systems with different degrees of elasticity on an ordinal scale.*

- Load Profile

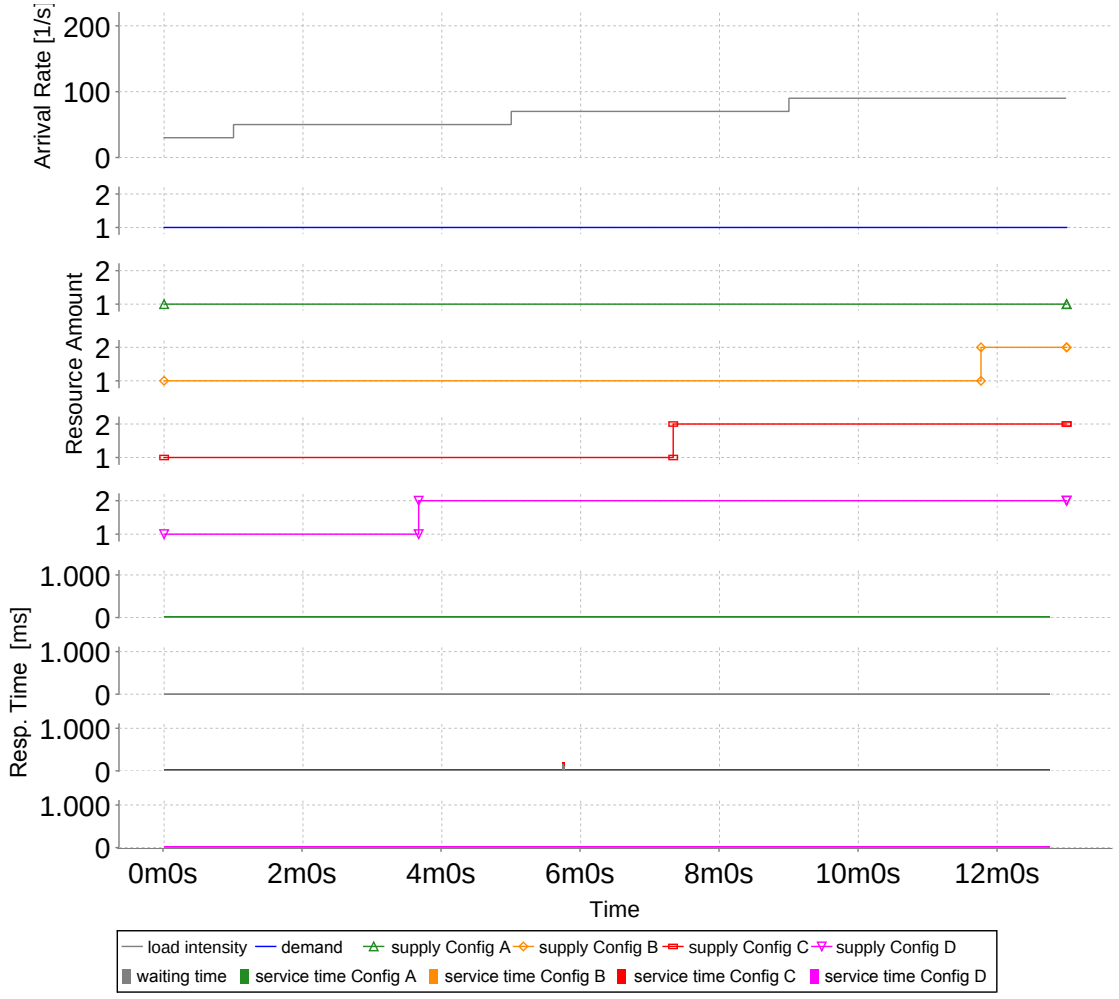


Figure 7.4: Evaluation of the $accuracy_O$ metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for decreasing *thresholdUp* values (A: 80%, B: 60%, C: 40%, D: 30%).

The over-provision accuracy metric is evaluated with the load profile illustrated in Figure 7.4. The load profile starts with 30% of the maximum intensity that one instance can handle. After one minute, the intensity changes stepwise - every four minutes with 20% increments - to 90% load intensity. A system with a low degree of elasticity will allocate a second resource instance when the CPU utilization increases although it is not needed.

- Different Degrees of Elasticity
  The degree of elasticity of the test system is varied by changing the *thresholdUp* for the scale up rule. For decreasing values of this parameter the system tends to provision resources to early, which leads to lower degrees of elasticity. The $accuracy_O$ metric is evaluated for the following *thresholdUp* values: 80% (Configuration A), 60% (Configuration B), 40% (Configuration C) and 30% (Configuration D).

- Results
  Figure 7.4 shows the elasticity behaviors for different values of *thresholdUp*. It can be seen that due to lower degrees of elasticity for decreasing *thresholdUp* values, the amount of over-provisioned resources increases. This is reflected by the metric results shown in Table 7.8. For decreasing degrees of elasticity, that means for decreasing values of *thresholdUp*, the $accuracy_O$ metric increases. Thus, the $accuracy_O$ allows to rank elastic systems on an ordinal scale as stated in Hypothesis 4.

thresholdUp [%]	80	60	40	30
$accuracy_U$ [*resource units*]	0	0.094	0.435	0.717

Table 7.8: Measurement results for the $accuracy_O$ metric

7.3.3 Under-provision Timeshare: $timeshare_U$

- Hypothesis

  Hypothesis 5 *The metric $timeshare_U$ allows to rank systems with different degrees of elasticity on an ordinal scale.*

- Load Profile
  The under-provision timeshare metric is evaluated using a simple step load profile, illustrated in Figure 7.5. The load profile starts with a constant load intensity i_{low} which can easily handled with one resource instance. After one minute, the intensity changes to a higher load intensity i_{high} for which two resource instances are necessary for four minutes. The load profile is calibrated in a way, that intensity i_{low} is half of the maximum intensity the test system can withstand using one resource and i_{low} is 175% of the maximum intensity the test system can withstand using one resource. Thus, with the correct amount of resources the system should be able to handle the load easily, for any point in time.

- Different Degrees of Elasticity
  For the evaluation, the degree of elasticity of the test system is changed by modifying the *condTrueDurUp* parameter. With increasing values for this parameter the system reacts delayed, which leads to lower degrees of elasticity. The $accuracy_U$ metric is evaluated for the following *condTrueDurUp* values: 5s (Configuration A), 10s (Configuration B), 30s (Configuration C) and 60s (Configuration D).

- Results
  Figure 7.5 shows the elasticity behaviors for different values of the *condTrueDurUp*

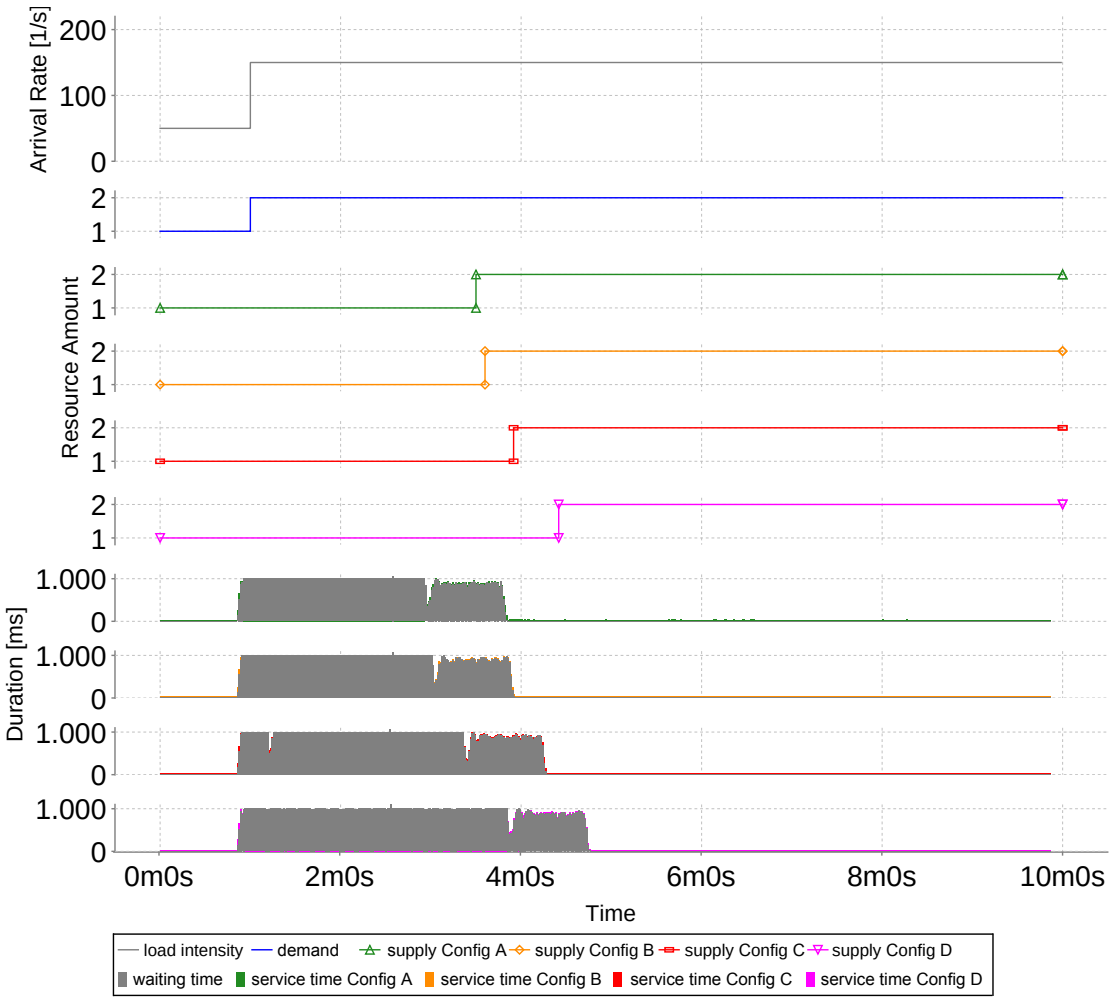


Figure 7.5: Evaluation of the $timeshare_U$ metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing *condTrueDurUp* values (A: 5s, B: 10s, C: 30s, D: 60s).

parameter. It can be seen that due to lower degrees of elasticity for increasing *condTrueDurUp* values, the timeshare where the system under-provisions increases. This is reflected by the metric results shown in Table 7.9. For decreasing degrees of elasticity, that means for increasing *condTrueDurUp* values, the $timeshare_U$ metric increases. Thus, the $timeshare_U$ allows to rank elastic systems on an ordinal scale as stated in Hypothesis 5.

condTrueDurUp [*s*]	5	10	30	60
$timeshare_U$ [%]	25.1	26.0	29.3	34.3

Table 7.9: Measurement results for the $timeshare_U$ metric

7.3.4 Timeshare Ratio: $timeshare_O$

- Hypothesis

Hypothesis 6 *The metric* $timeshare_O$ *allows to rank systems with different degrees of elasticity on an ordinal scale.*

- Load Profile

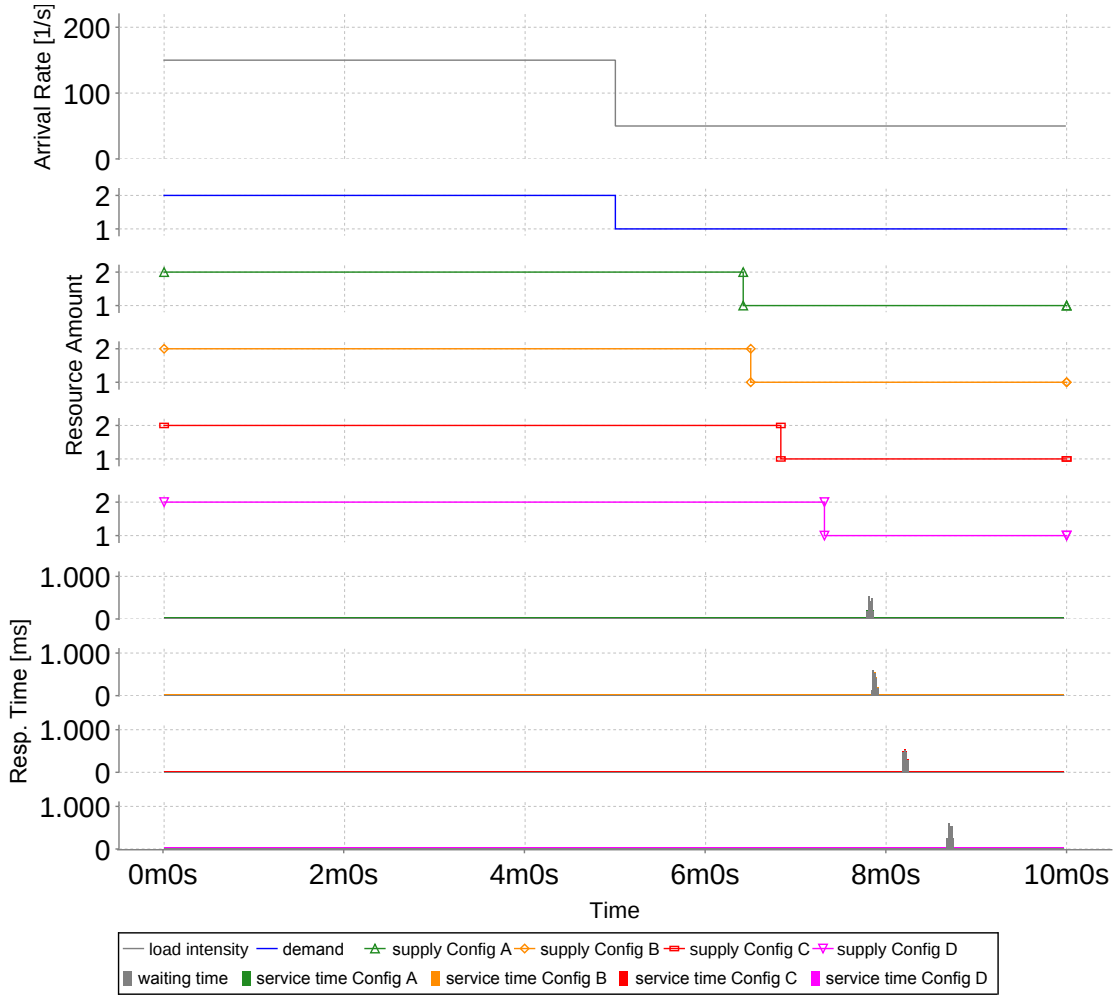


Figure 7.6: Evaluation of the $timeshare_O$ metric. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing *condTrueDurDown* values (A: 5s, B: 10s, C: 30s, D: 60s).

Similar to the evaluation of the under-provision timeshare, the over-provision timeshare metric is evaluated using a simple step load profile, illustrated in Figure 7.6. The load profile starts with a constant load intensity i_{high} which can easily be handled with two resource instances. After one minute, the intensity changes to a lower load intensity i_{low} for which just one resource instance is enough for four minutes. The load profile is calibrated in a way, that intensity i_{high} is 175% of the maximum intensity the test system can withstand using one resource and i_{low} is 50% of the maximum intensity the test system can withstand using one resource. Thus, with the correct amount of resources the system should be able to handle the load easily, for any point in time.

- Different Degrees of Elasticity
  The degree of elasticity of the test system is changed the same way as it was done for the under-provisioning timeshare evaluation.

- Results
  Figure 7.6 shows the elasticity behaviors for different values of the *condTrueDurDown*

parameter. It can be seen that due to lower degrees of elasticity for increasing *condTrueDurDown* values, the timeshare where the system over-provisions increases. This is reflected by the metric results shown in Table 7.10. For decreasing degrees of elasticity, that means for increasing *condTrueDurDown* values, the $timeshare_O$ metric increases. Thus, the $timeshare_O$ allows to rank elastic systems on an ordinal scale as stated in Hypothesis 6.

condTrueDurDown[*s*]	5	10	30	60
$timeshare_O$ [%]	14.3	15.0	18.4	23.3

Table 7.10: Measurement results for the $timeshare_O$ metric

7.3.5 Jitter Metric: *jitter*

The *jitter* metric evaluates whether a system exhibits imperfect elasticity because it makes to many or to few adaptations of the resource supply. In both cases the absolute value of the metric increases for decreasing degrees of elasticity. However, for the first case the sign of the metric should be positive in contrast for the letter case negative. To facilitate the evaluation, both behaviors are evaluated separately.

7.3.5.1 Positive Jitter Evaluation - Superfluous Adaptations

- Hypothesis

  Hypothesis 7a *The metric jitter allows to rank systems with - due to superfluous resource adaptations - different degrees of elasticity on an ordinal scale.*

- Load Profile
  The load profile used for the first evaluation step is a constant load profile illustrated in Figure 7.7. The load profile is calibrated in a way that the intensity is 90% of the maximum intensity for one resource. Since the resource is utilized at a high level, rule based elasticity mechanisms tend allocate another instance, to be able to handle a possibly increasing load. As soon as the resource is provisioned the average resource utilization drops and - depending on the configuration - elasticity mechanisms may deallocate the superfluous resource again. Thus, this load profile tries to provoke unnecessary allocations and deallocations. A system with high degree of elasticity will not do unnecessary de-/allocations or will just do a few of them.

- Different Degrees of Elasticity
  For the evaluation, the thresholds [*thresholdDown*, *thresholdUp*] are set to: [40,50]. With these thresholds the elasticity mechanism tends to allocate and deallocate a second resource. Furthermore, the *quietTime* is set to 30s, to allow faster reactions. The degree of elasticity of the test system is changed by modifying the *condTrueDurUp/Down* parameter for both, the scale up and the scale down rule. For decreasing values of these parameters, the system reacts overly responsive, which leads to lower degrees of elasticity (to many unnecessary allocations). The *jitter* metric is evaluated for the following *condTrueDurUp/Down* values: 120s (Configuration A), 60s (Configuration B), 30s (Configuration Cc), 5s (Configuration D).

- Results
  Figure 7.7 shows the elasticity behaviors for different values of the *condTrueDurUp/Down* parameter. It can be seen that due to decreasing degrees of elasticity

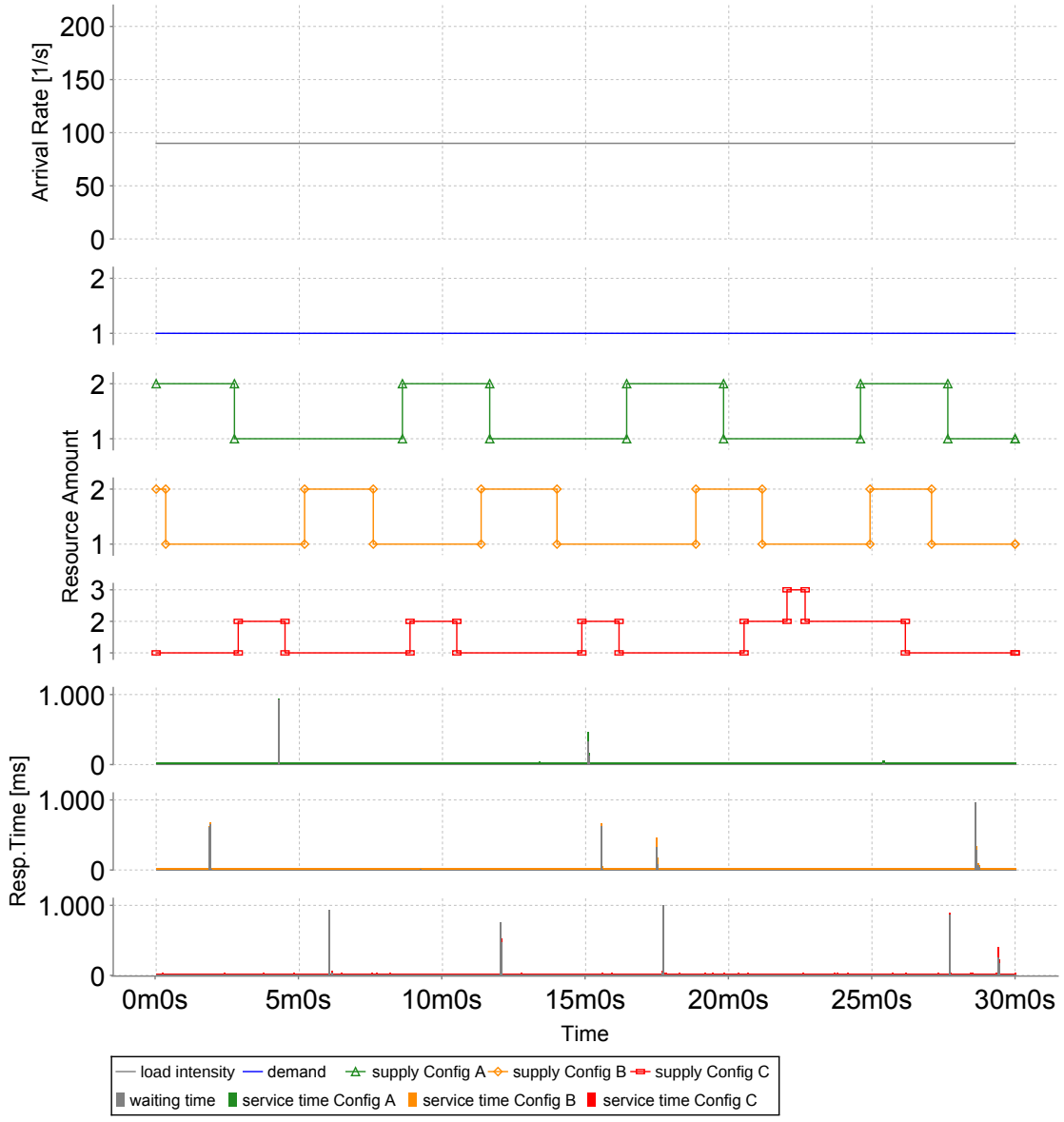


Figure 7.7: Evaluation of the *jitter* metric for superfluous adaptations. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for decreasing *condTrueDurUp/Down* values (A: 120s, B: 60s, C: 30s, D: 5s).

for decreasing *condTrueDurUp/Down* values, the amount of unnecessary resource de-/allocation events increases. This is reflected by the metric results shown in Table 7.11. For decreasing degrees of elasticity elasticity, that means for decreasing *condTrueDur* values, the *jitter* metric increases. Thus, the *jitter* metric allows to rank elastic systems whose imperfect elasticity is due to superfluous adaptations on an ordinal scale as stated in Hypothesis 7a.

- Remarks
  When the smallest *condTrueDurUp/Down* is used (Configuration D), the system provisions a third resource in some cases although *maxInstances* is set to two. One explanation for this behavior is, that although a second instance is started before, the *quiettime* plus the *condTrueDurUp* was not enough time to allow the system to take advantage of the second instance and therefore a substitute instance is allocated.

condTrueDurUp/Down **[s]**	120	60	30	5
jitter $\left[\frac{\#adap.}{min}\right]$	0.233	0.300	0.333	0.433

Table 7.11: Measurement results for the *jitter* metric (positive *jitter*).

Shortly after the creation of the third instance the system takes advantage of the second instance which results in a fast deallocation. This behavior of creating superfluous instances is reflected by the *jitter* metric.

7.3.5.2 Negative Jitter Evaluation - Missing Adaptations

- Hypothesis

  Hypothesis 7b *The metric jitter allows to rank systems with - due to missing adaptations - different elasticity on an ordinal scale.*

- Load Profile
  The load profile used for the second evaluation step is illustrated in Figure 7.8. The load profile changes between two intensities i_{low} and i_{high} in a repeated manner. Hereby, the Δt time for which the intensity is constant is seven minutes at the beginning. After every two intensity changes Δt is decrease by 15%. Like in the load profile for the *timeshare* evaluation, the load profile is calibrated in a way, that intensity i_{low} is half of the maximum intensity the test system can withstand using one resource and i_{high} is 175% of the maximum intensity the test system can withstand using one resource. Thus, with the correct amount of resources the system should be able to handle the load easily, for any point in time. A system with a high degree of elasticity will be able to follow all the demand changes. In contrast, a system with a low degree of elasticity will only be able to follow the first few demand changes.

- Different Degrees of Elasticity
  As in the first *jitter* evaluation, the *quietTime* is set to 30s, to allow faster reactions. The degree of elasticity of the test system is changed by modifying the *condTrueDurUp/Down* parameter for both, the scale up rule and the scale down rule. For increasing values of this parameter, the system reacts delayed leading to decreasing degrees of elasticity. The *jitter* metric is evaluated for the following values: 5s (Configuration A), 30s (Configuration B) 60s (Configuration C) and 120s (Configuration D).

- Results

condTrueDurUp/Down **[s]**	5	30	60	120
jitter $[\frac{\#adap.}{min}]$	-0.093	-0.107	-0.107	-0.133

Table 7.12: Measurement results for the *jitter* metric (negative *jitter*)

Figure 7.8 shows the elasticity behaviors for different values of the *condTrueDurUp/Down* parameter. It can be seen that due to decreasing degrees of elasticity for increasing *condTrueDurUp/Down* values, the number of demand changes which the system is able to follow decreases. This is reflected by the metric results shown in Table 7.12. For decreasing degrees of elasticity, that means for increasing *condTrueDurUp/Down*, the absolute value of the *jitter* metric increases. Thus, the *jitter* metric allows to rank elastic systems whose imperfect elasticity is due to missing adaptations on an ordinal scale as stated in Hypothesis 7b.

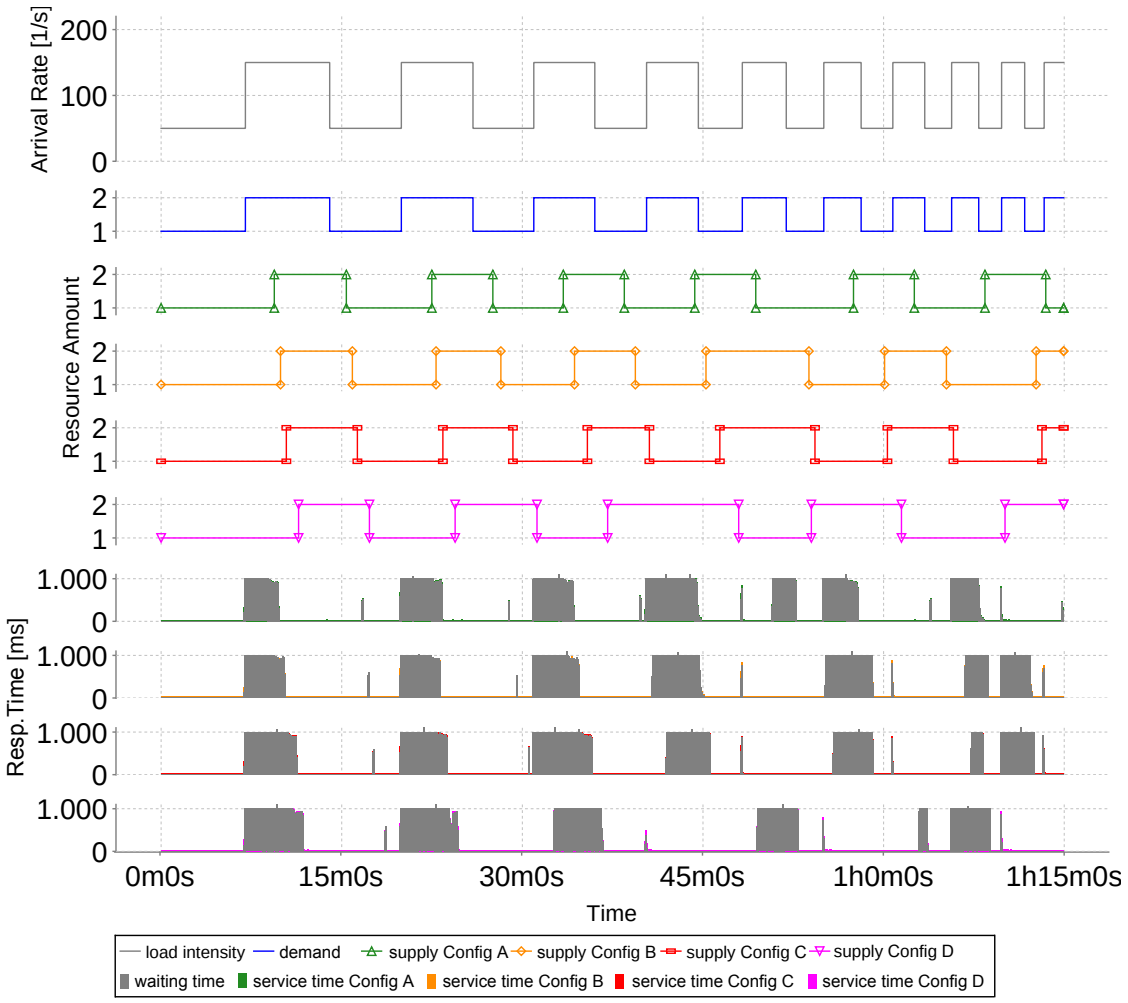


Figure 7.8: Evaluation of the *jitter* metric for missing adaptations. Load profile (top), induced resource demand (second graph) and measured resource supply and response times for increasing *condTrueDurUp/Down* values (A: 5s, B: 30s, C: 60s, D: 120s).

- Remarks
  Although the time between two adaptations is smaller for the Configuration B than for Configuration C, both systems do the same amount of adaptations within the measurement period. Thus, the elasticity with respect to missing adaptations is equal for both measurements. This is reflected by the *jitter* metric.

7.3.6 Discussion

The evaluation of the elasticity metrics $accuracy_U$, $accuracy_O$, $timeshare_U$, $timeshare_O$ and *jitter* showed that these metrics allow to rank resource elastic systems on an ordinal scale. Comparing the evaluation of different metrics, it can be seen that some elasticity parameters have different effects on different elasticity aspects. For example, increasing the values for the *condTrueDurUp/Down* parameter values leads to lower degrees of elasticity in situations where the demand changes over time. Therefore, $timeshare_U$, $timeshare_O$ increase with increasing *condTrueDurUp/Down* parameter values in the evaluation experiments. In contrast, the negative *jitter* evaluation shows that increasing the *condTrueDurUp/Down* parameter values can lead to a higher degree of elasticity in cases where the demand does not change over time, because the amount of superfluous allocations is reduced. Thus,

these experiments demonstrated that a configuration of a rule based elasticity mechanism which is superior to other configurations for a given load profile must not be the best configuration for other load profiles.

7.4 Case Study with a Realistic Load Profile

The preceding section evaluated the elasticity metrics with very simple artificial load profiles and only two scaling stages. This section demonstrates the benchmarking capabilities for a realistic load profile and a realistic number of resources on a private cloud as well as on a public cloud.

7.4.1 Private Cloud - CloudStack

This subsection demonstrates the benchmarking capabilities for different configurations of the private cloud test system described in Section 7.1.1.

7.4.1.1 Load Profile

The load profile used for the case study is illustrated in Figure 7.9(c). It is derived from a real intensity trace that has previously been used in [vK14, HHKA14]. The trace features the amount of CICS, IMS, and OPEN transactions on an IBM z196 Mainframe during February 2011 with a quarter-hourly resolution.

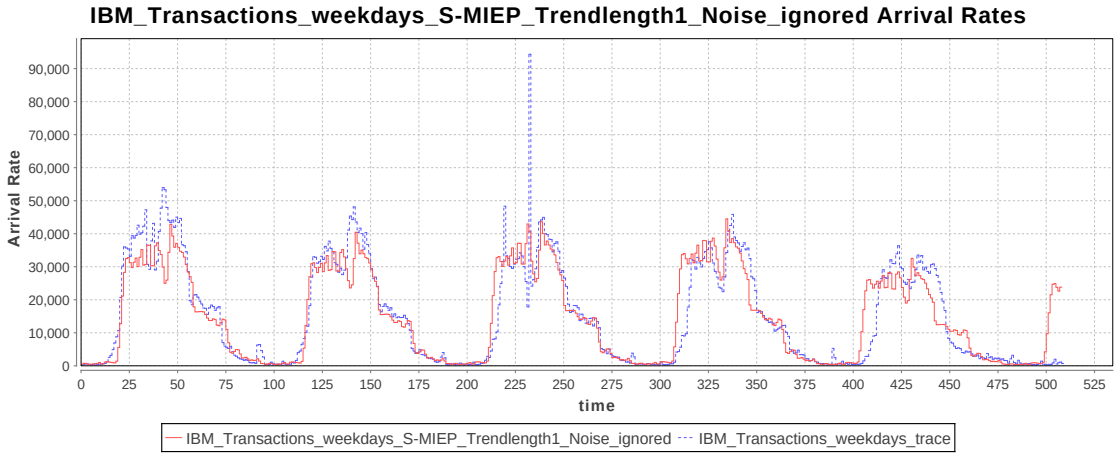
In order to obtain a representative load profile for a work day, the LIMBO toolkit has been used to extract a DLIM model for the first five days of the trace with the *Simple Model Instance Extraction Process* [vK14]. The result is illustrated in Figure 7.9(a). An evaluation of the model extraction process for this trace can be found in [vK14]. In a second step, the number of seasonal periods has been set to one. Thus, the model represents only one day but still contains data from five days. Additionally, the modeled peaks are removed from the DLIM model (Figure 7.9(b)).

To reduce the experimentation time the load profile has been compacted from 24 hours to 6 hours. Thus, the load intensity within the model increases by a factor of four (Figure 7.9(c)).

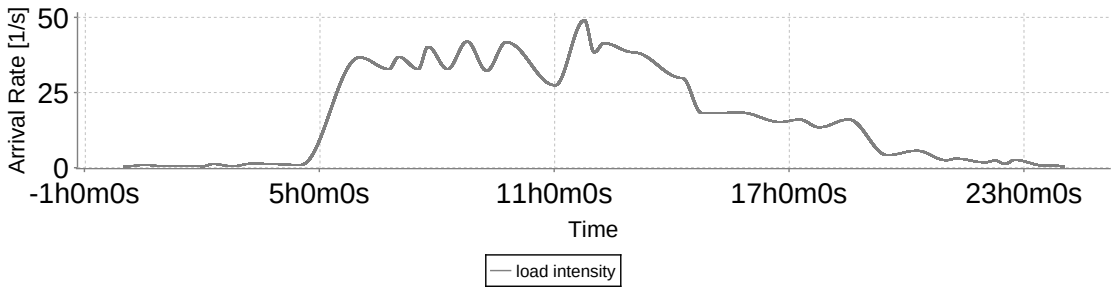
7.4.1.2 Calibration

This case study showcases the elasticity evaluation of the test system (compare Section 7.1) using up to ten VMs as resources. Thus, the *System Analysis* must be conducted for ten resources. The evaluation in Section 7.2.2 showed that the linearity assumption for the test system is roughly true for up to ten resources. However, the length of steps within the intensity demand curve varied to some extend. To be able to account better for this irregular scaling behavior, the *Detailed System Analysis* has been used. Figure 7.10 illustrates the derived mapping function $demand(intensity)$. Apart from some very small deviations, the result is equal to the results derived within the evaluation of the linearity assumption (Section 7.2.2, Table 7.5). Within the adjustment step, the load profile is adjusted to require ten the resources for the profiles maximum intensity. The resulting adjusted load profile is shown in Figure 7.9(d).

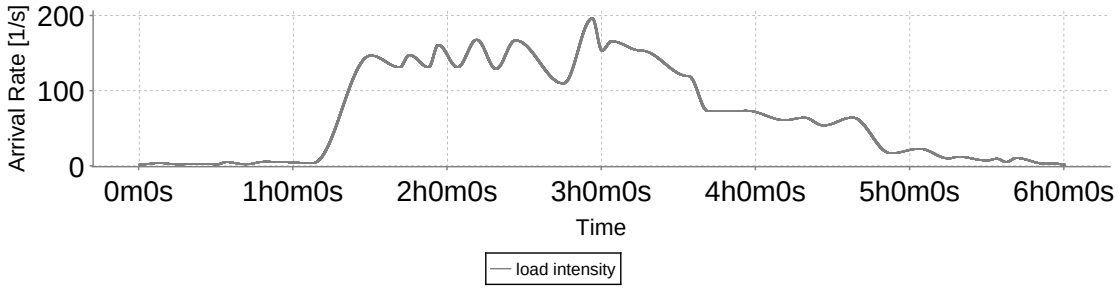
The load intensity within this load profile varies between two and 339 requests per second. The timestamp file created within the measurement process contains about 2.7 million timestamps.



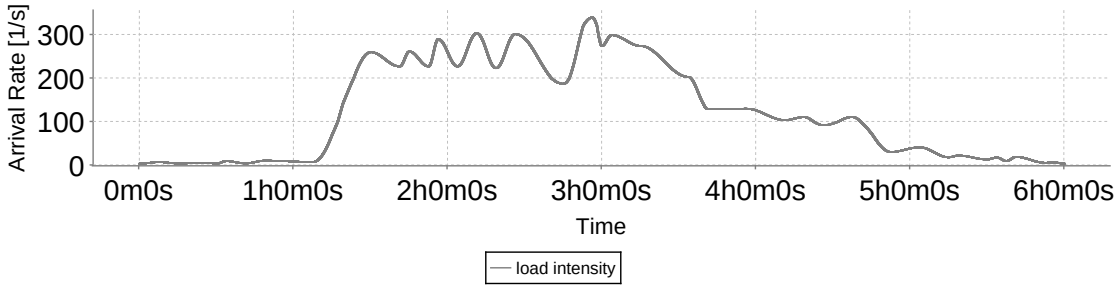
(a) Original and modeled intensity [requests / 15 min] trace [vK14]



(b) Load profile [requests / sec.] for a single day without peaks



(c) Compacted load profile [requests / sec.]: 24h -> 6h



(d) Compacted load profile [requests / sec.] adjusted for the test system

Figure 7.9: Load profile for one day derived from a real five day transaction trace

7.4.1.3 Elasticity Rule Parameter Configurations

The resource elasticity of the test system is evaluated for different elasticity rule parameter settings:

Configuration A serves as a baseline configuration for elasticity comparisons. Due to a big

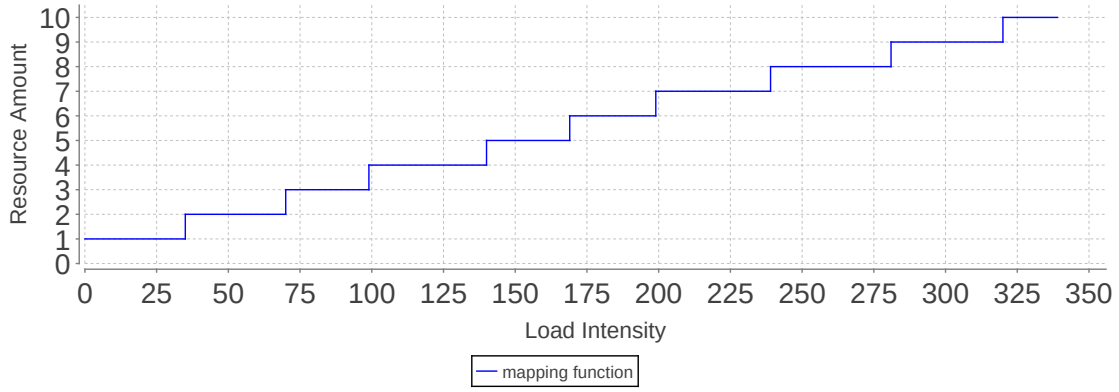


Figure 7.10: Mapping function *demand*(*intensity*) derived with the *Detailed System Analysis*

Config.	quietTime	condTrue-DurUp	condTrue-DurDown	thresh-oldUp	thresh-oldDown
A	240s	120s	120s	90%	10%
B	240s	30s	30s	90%	50%
C	240s	30s	30s	90%	80%
D	120s	30s	30s	65%	50%
E	60s	30s	30s	65%	40%
F	120s	30s	30s	65%	10%

Table 7.13: Different elasticity parameter configurations for CloudStack

quietTime and high *condTrueDurUp/Down* values, it reacts rather sluggish. The high/low *thresholdUp/Down* values provoke bad accuracy, additionally.

Configuration B uses smaller *condTrueDurUp/Down* values and an increased *thresholdDown* in order to induce earlier scale downs.

Configuration C uses a further increased *thresholdDown* in order to induce even earlier scale downs and therefore better accuracy for over-provisioning cases.

Configuration D provokes earlier scale ups by using an decreased *quietTime* and a decreased *thresholdUp*.

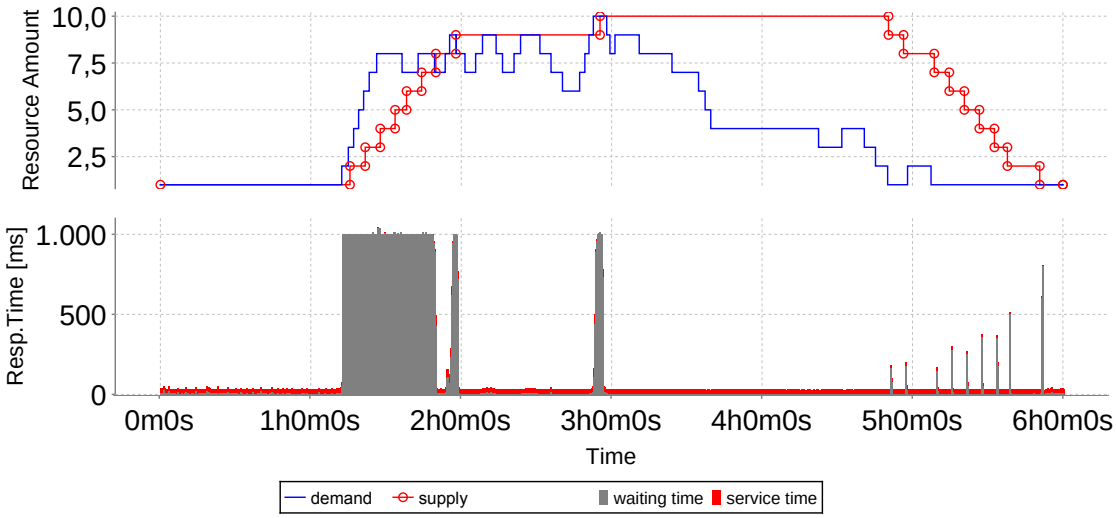
Configuration E tries to provoke even earlier scale ups and at the same time later scale down times by using a further decreased *quietTime* and a decreased *thresholdUp* value.

Configuration F induces the same scale up behavior as Configuration D, but very late scale downs similar to Configuration A.

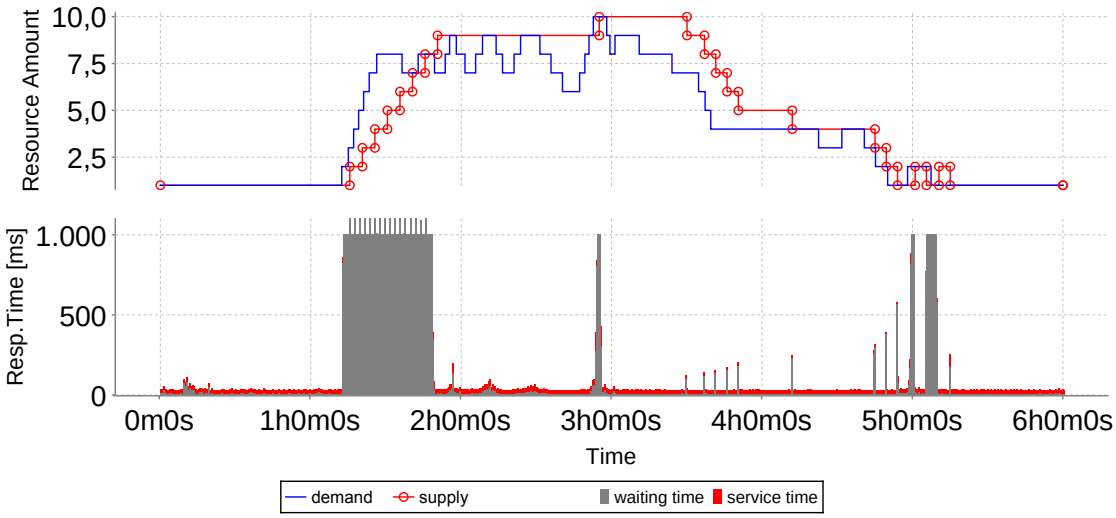
7.4.1.4 Elasticity Measurements

Visualization of Demand, Supply, and Response Time

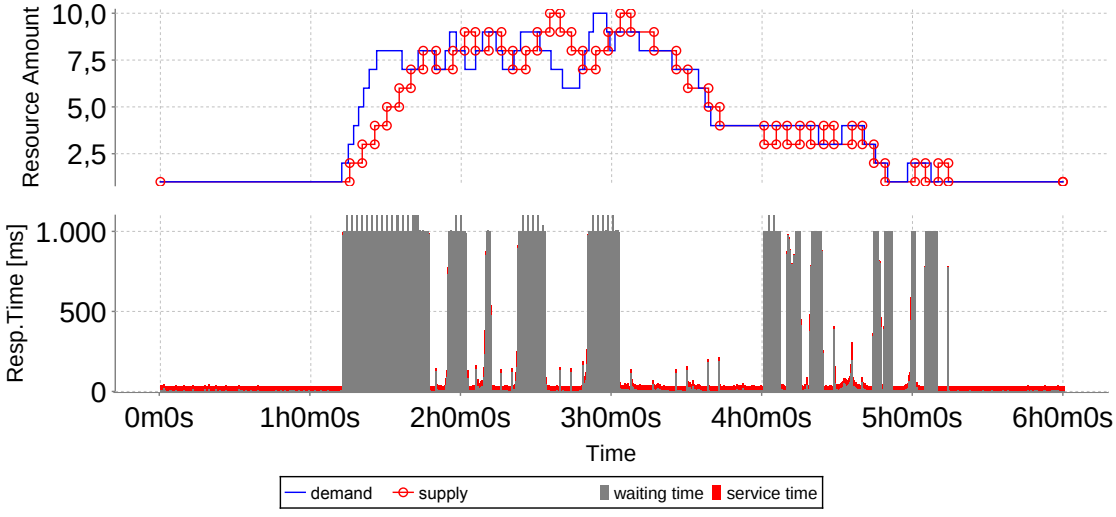
The Figures 7.11 and 7.12 show the resource demand curve and the measured resource supply curve for the six different elasticity rule Configurations A-F. These curves give a visual impression of the different elastic behaviors. Under each resource allocation graph a response time graph shows how the response times vary during the measurement runs. It allows to estimate the amount of SLO violations visually.



(a) Configuration A: quietTime: 240s, condTrueDurUp/Down: 120s, thresholdUp: 90%, thresholdDown: 10%
Baseline Configuration: Slow resource increase, very slow resource decrease

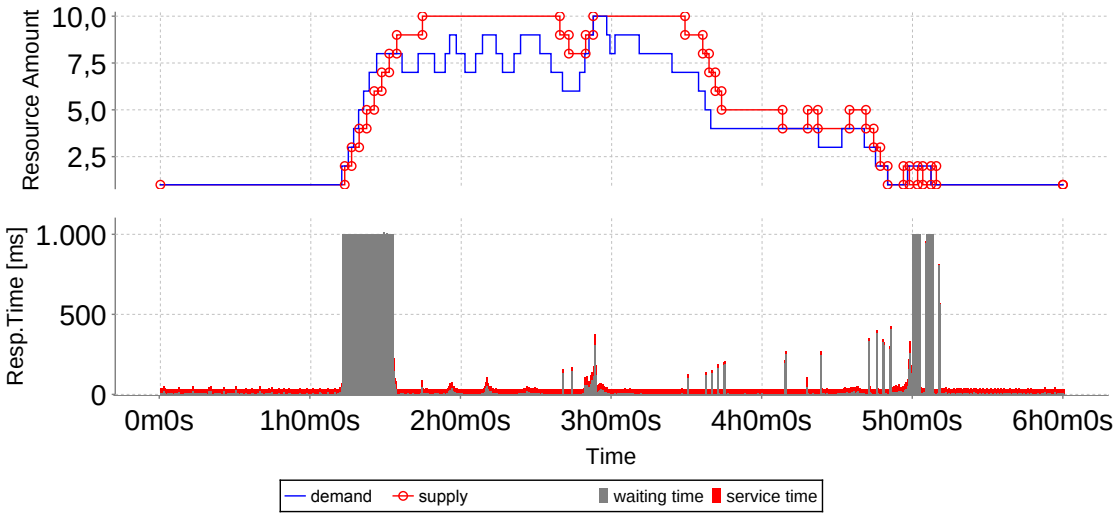


(b) Configuration B: quietTime: 240s, condTrueDurUp/Down: 30s, thresholdUp: 90%, thresholdDown: 50%
Slow resource increase, faster resource decrease than Configuration A

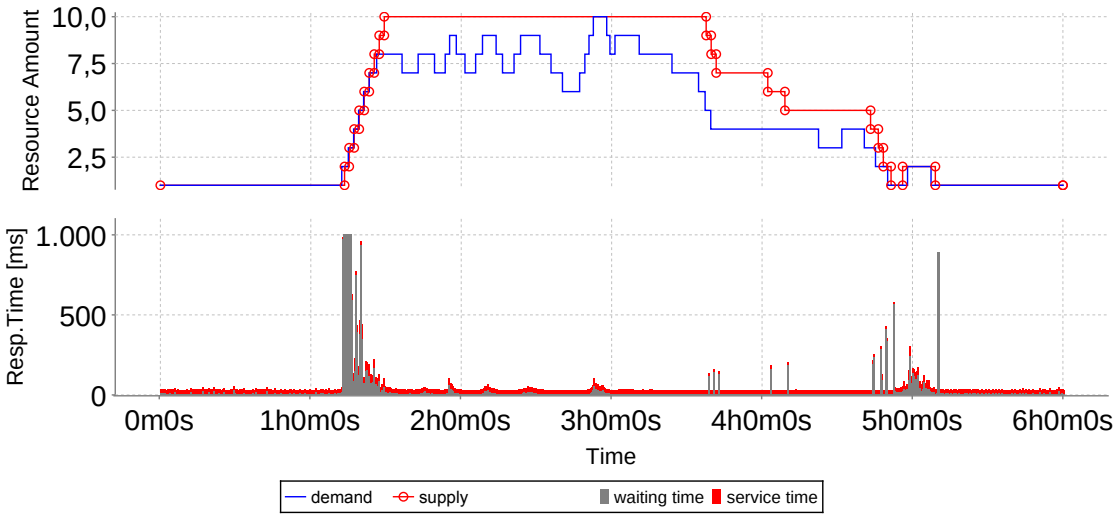


(c) Configuration C: quietTime: 240s, condTrueDurUp/Down: 30s, thresholdUp: 90%, thresholdDown: 80%
Slow resource increase, faster resource decrease than Configuration B

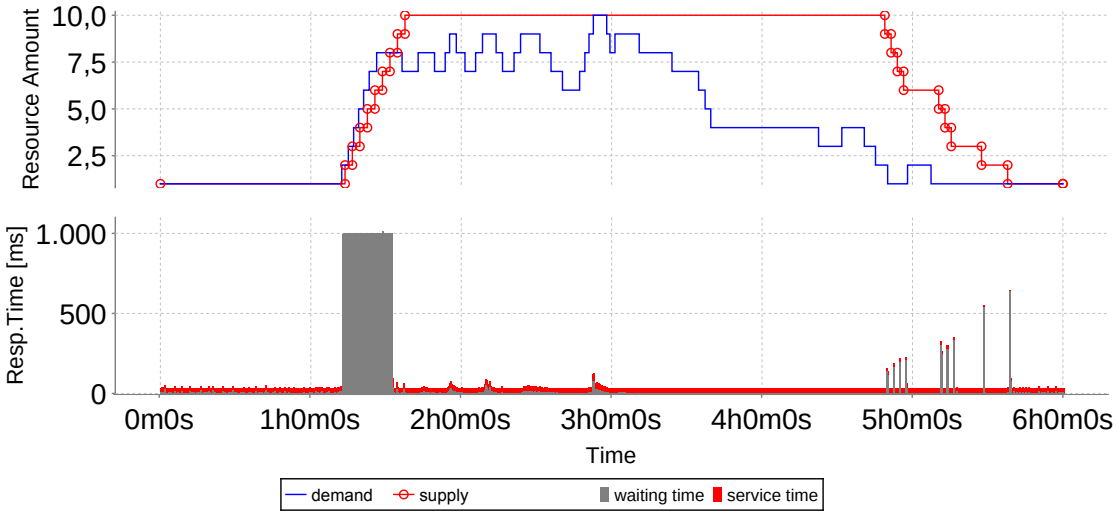
Figure 7.11: Elastic behavior for different elasticity rule parameter settings on CloudStack (1)



(a) Configuration D: quietTime: 120s, condTrueDurUp/Down: 30s, thresholdUp: 65%, thresholdDown: 50% Faster resource increase than Configurations A-C, reasonable resource decrease



(b) Configuration E: quietTime: 60s, condTrueDurUp/Down: 30s, thresholdUp: 65%, thresholdDown: 40% Faster resource increase than Configurations A-D, reasonable resource decrease, few SLO violations



(c) Configuration F: quietTime: 120s, condTrueDurUp/Down: 30s, thresholdUp: 65%, thresholdDown: 10% Resource increase as in Configuration D, slow resource decrease.

Figure 7.12: Elastic behavior for different elasticity rule parameter settings on CloudStack (2)

Metric Result Discussion

Table 7.14 summarizes the metric results for the different elasticity rule parameter configurations. The metric values allow to quantify the different elastic behaviors.

Parameter Configuration	$accuracy_O$ [res. units]	$accuracy_U$ [res. units]	$timeshare_O$ [%]	$timeshare_U$ [%]	$jitter$ $\left[\frac{\#adap.}{min}\right]$
A	2.425	0.264	60.1	11.7	-0.067
B	0.664	0.224	40.6	10.6	-0.056
C	0.219	0.383	15.4	23.4	0.006
D	0.815	0.080	48.7	6.5	-0.028
E	1.184	0.006	56.0	0.6	-0.061
F	2.423	0.067	66.1	4.8	-0.067

Table 7.14: Metric results for the evaluated configurations on CloudStack

Due to a decreased *condTrueDurUp/Down* parameter value for Configuration B compared to Configuration A, the $timeshare_U$ and the $accuracy_U$ metrics decrease (improve) slightly. The increased *thresholdDown* for Configuration B has a more significant impact: The CSUT scales down earlier and as a result the $timeshare_O$ and especially the $accuracy_O$ metric improve considerably.

Configuration C leads, compared to Configurations A and B, to faster scale downs. Thus, the $accuracy_O$ and $timeshare_O$ metrics improve further. Since many of the scale downs occur to early, the $accuracy_U$ and $timeshare_U$ metrics get worse. In contrast to the other configurations, the *jitter* metric is positive for Configuration C. This means the elasticity parameters settings result in unnecessary de-/allocations within the measurement run. The other configurations lead to an elastic behavior with less frequent supply adaptations.

The Configurations D and E make the CSUT scale up earlier than the previous configurations. At the same time scale downs occur later than for Configurations B and C. This behavior with very little under-provisioning is reflected by very low (good) $accuracy_U$ and $timeshare_U$ metric values. Otherwise, the over-provisioning occurs more often and to a greater extend and result in a worse $accuracy_O$ and $timeshare_O$ metric.

Submission Accuracy Evaluation

Parameter Configuration	Percentile for submission delay [ms] 95%	98%
A	1	2
B	1	2
C	1	3
D	1	1
E	1	2
F	1	1

Table 7.15: Percentiles for the delay between scheduled and real request submission

A fair elasticity comparison requires that the resource demand triggered on the test system is the same for every measurement run. This means requests should be send on time as defined by the timestamp file. Table 7.15 illustrates the submission accuracy for the five measurement runs. It can be seen that up to the 98%-percentile the delay between the scheduled and the real request submission is less than or equal to three milliseconds.

7.4.1.5 Aggregated Elasticity Measure

Section 5.4 explained how a group of different systems or configurations can be compared with the help of an elasticity measure that aggregates the elasticity submetrics. This subsection demonstrates the application of an speed up based aggregation as described in Section 5.4.2.

As a baseline, Configuration A is used. With the help of Formula 5.1 the four metrics $accuracy_O, accuracy_U, timeshare_O$ and $timeshare_u$ are aggregated to an aggregated *elastic speedup* measure. The *elastic speedup* metric depends on the weights w_{acc_U}, w_{acc_O}, w_{ts_U}, w_{ts_O}, w_{acc} and w_{ts}, which are all set to 0.5 in a first evaluation step. This results in an equal weighting of the aspects accuracy and timing for both, under-provisioning and over-provisioning scenarios. The resulting *elastic speedup* for Configurations A-F as well as some intermediate results are illustrated in Table 7.16.

Parameter Configuration	*accuracy weighted* [res. units]	*timeshare weighted* [%]	*accuracy speedup*	*timeshare speedup*	**elastic speedup**	SLO violations [%]
C	0.301	19.4	4.467	1.851	2.875	41.2
B	0.444	25.6	3.028	1.402	2.061	17.8
D	0.448	27.6	3.004	1.301	1.977	8.4
E	0.595	28.3	2.260	1.269	1.693	0.7
F	1.245	35.5	1.080	1.013	1.046	7.6
A	1.345	35.9	1.000	1.000	1.000	20.3

Table 7.16: Compute the *elastic speedup* (Formula 5.1) for weights:
$w_{acc_U} = w_{acc_O} = w_{ts_U} = w_{ts_O} = w_{acc} = w_{ts} = 0.5$

According to the aggregated metric, Configuration C offers the best elasticity. It is followed by Configurations B, D, E, F, and A. Additionally to the metric results, Table 7.16 contains a column named *SLO violations*. This column measures the share of requests that could not be processed successfully or have a response time of more than 500ms. With the used SLO defined in Section 7.1.3, for a system with perfect elasticity, the share of those requests should be at most 5%. Although Configuration C offers the best elasticity according to the aggregated metric, it also has highest share of long response times. This fact seems counterintuitive first, but can be explained by the characteristics of the elastic behavior: Although the test system is able to reduce the amount of over-provisioned resources as well as the timeshare of over-provisioning significantly by using Configuration C, it also under-provisions more resources and for a higher timeshare compared to using other configurations. To some extend the under-provisioning is due to the reactive nature of a rule based system. A combination with proactive approaches may reduce the amount of under-provisioning as well as the SLO violations share.

As discussed in Subsection 5.4.2.1, the weights can be used to account for different preferences when comparing the elasticity of different CSUTs. Table 7.17 shows the results for the aggregated elasticity measure *elastic speedup* when using the weights $w_{acc_U} = 0.2, w_{acc_O} = 0.8, w_{ts_U} = 0.2, w_{ts_O} = 0.8, w_{acc} = 0.2, w_{ts} = 0.8$. These weights increase the importance of the $accuracy_O$ metric compared to the $accuracy_U$ metric. At the same time, they increase the importance of the $timeshare_U$ metric compared to the $timeshare_O$ metric. This reflects the fact that additional costs due to over-provisioning mainly depend on the amount of over-provisioned resources. The costs for under-provisioning in contrast, mainly depend on the timeshare within that the number of provided resources is

Parameter Configuration	*accuracy weighted* [res. units]	*timeshare weighted* [%]	*accuracy speedup*	*timeshare speedup*	**elastic speedup**	SLO violations [%]
E	0.948	11.7	2.101	1.830	1.882	0.7
D	0.668	14.9	2.983	1.431	1.658	8.4
B	0.576	16.6	3.460	1.288	1.569	17.8
C	0.252	21.8	7.914	0.981	1.489	41.2
F	1.952	17.1	1.021	1.253	1.203	7.6
A	1.993	21.4	1.000	1.000	1.000	20.3

Table 7.17: Compute the *elastic speedup* (Formula 5.1) for weights:
$w_{acc_U} = 0.2, w_{acc_O} = 0.8, w_{ts_U} = 0.8, w_{ts_O} = 0.2, w_{acc} = 0.2, w_{ts} = 0.8$

not sufficient. Furthermore, the weights reflect that the aggregated timeshare metric (that mainly penalties under-provisioning) is more important than the aggregated accuracy metric (that mainly penalties over-provisioning).

According to these weights, Configuration E provides the best elasticity, followed by Configurations D, B, C, F and A. Now, the configuration with the best elasticity also offers the fewest share of requests with overly long response times (column SLO violations). Still there is no linear correlation between elasticity and the share of requests with response times exceeding 500ms, but this is due to the still wanted influence of over-provisioning.

7.4.1.6 Different Levels of Efficiency of Underlying Resources

The load profile adjustment becomes particularly important when systems are compared that use resources with different levels of efficiency. This subsection demonstrates the impact of using or not using the load profile adjustment. In order to compare resources with different levels of efficiency, different service offerings are used.

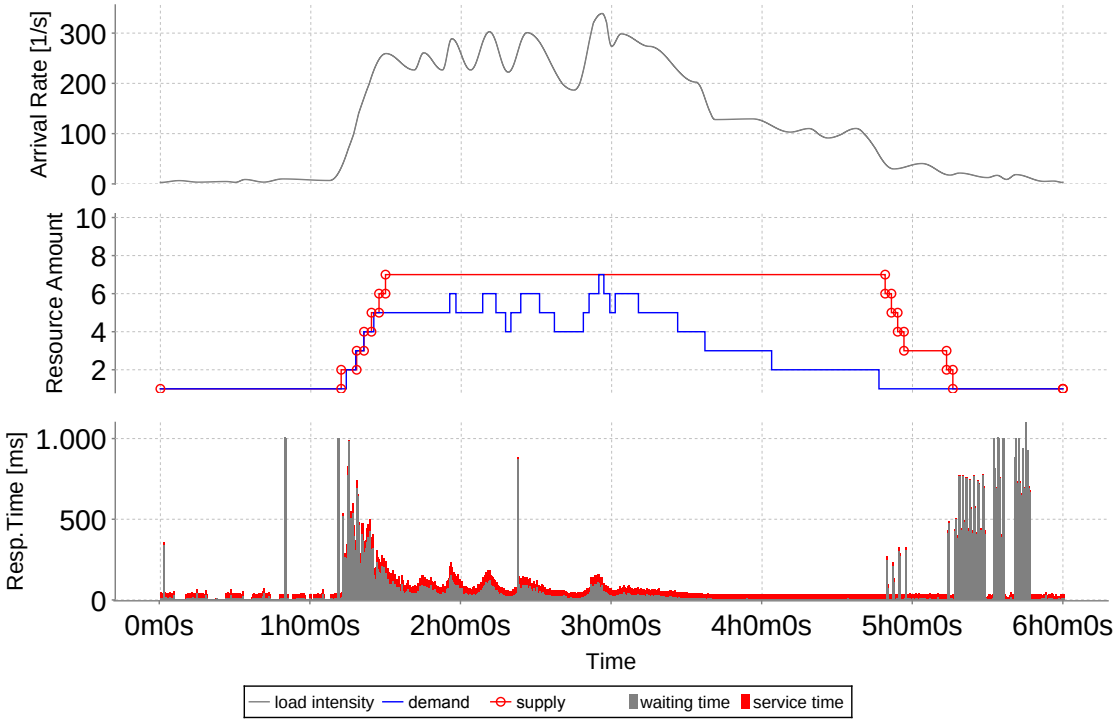
The preceding six hour experiments used a service offering - Offering A - that assigns one virtual CPU running at 2200 MHz to a VM. The following additional experiments use Offering B that assigns one virtual CPU running at 1024 MHz to a VM and Offering D that assigns two virtual CPU running at 2200 MHz to a VM.

Before running measurements, the test system is configured to use service Offering B or Offering D respectively. To allow calibrating the benchmark, the *Detailed System Analysis* is used to retrieve the mapping function *demand*(*intensity*) for Offering B and D.

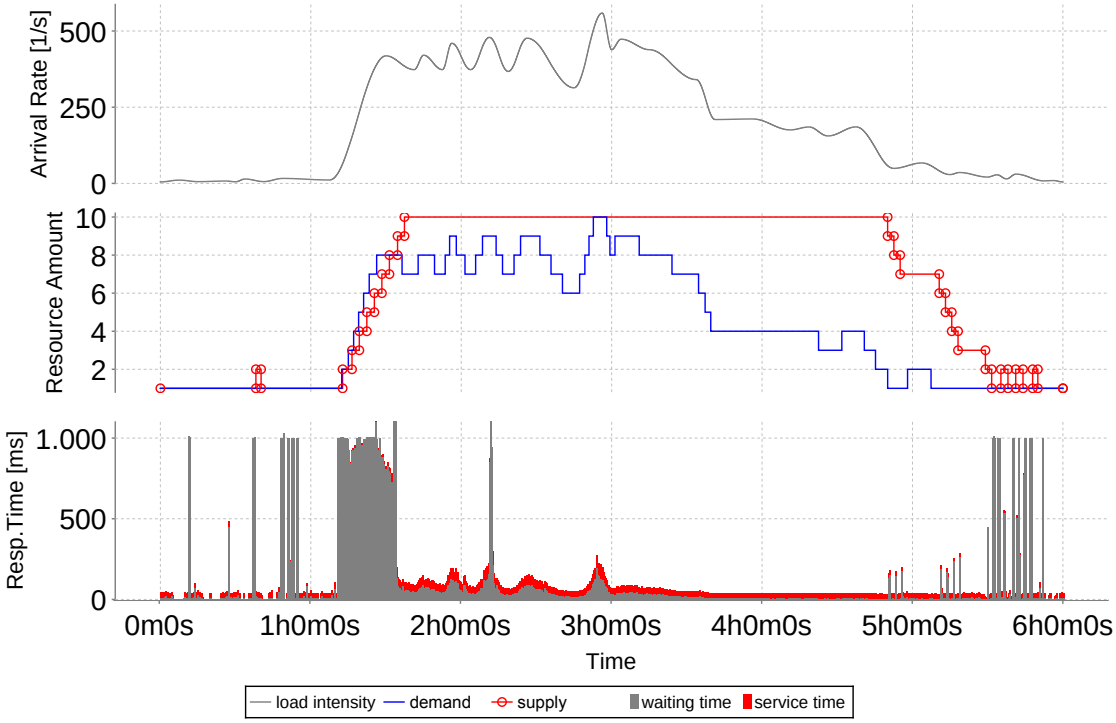
Figure 7.13(a) shows how the test system behaves when it is configured to use Offering B, but the load profile is adjusted for Offering A. Since Offering D uses a more powerful VMs, the system requires less instances and the elastic behavior looks better compared to using Offering A (Figure 7.12(c)).

In Figure 7.13(b), the test system is exposed to the calibrated load profile that was correctly adjusted for service Offering B. Although the used resources are more efficient for Offering B compared to Offering A, the demand changes are now induced at the same points in time. Comparing the elastic behavior when Offering A is used to the behavior when Offering F is used is easier now.

Figure 7.14 demonstrates a similar effect for the use of the less powerful Offering D. When the load profile is not adjusted according to the analysis results for Offering D (Figure 7.14(a)), the system requires more instances and the elastic behavior looks worse compared to using Offering A (Figure 7.12(c)).

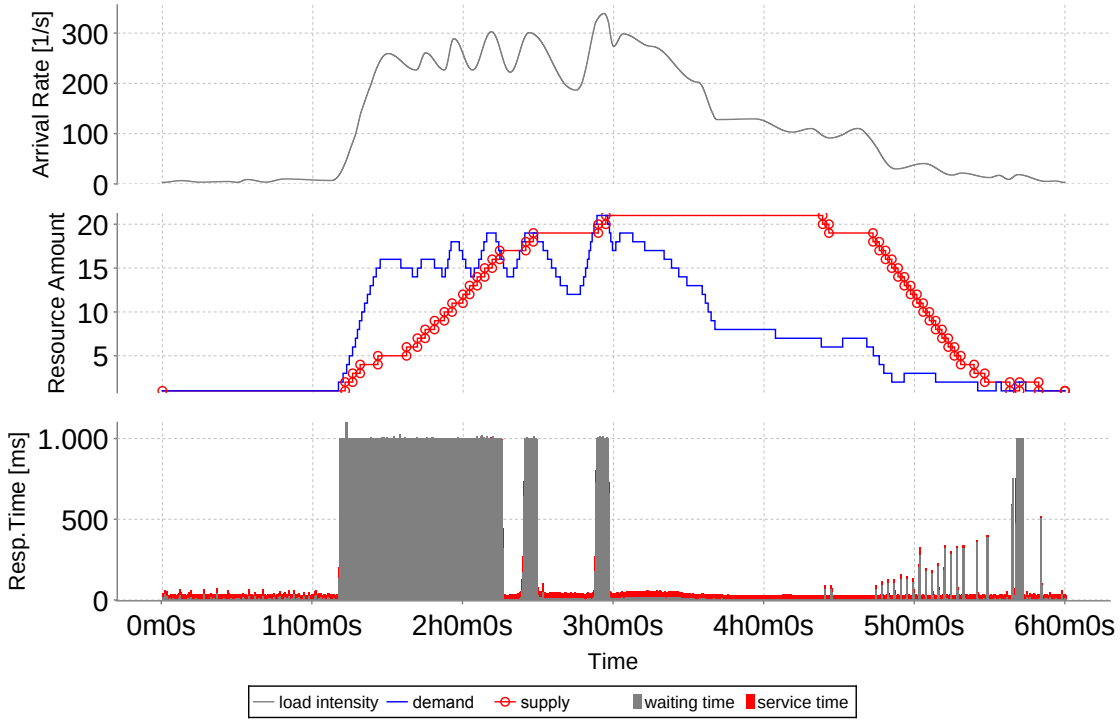


(a) System using Offering B exposed to load intensity profile adjusted for Offering A. Demand curve is not equal to the demand curve for Offering A (Figures 7.11,7.12)

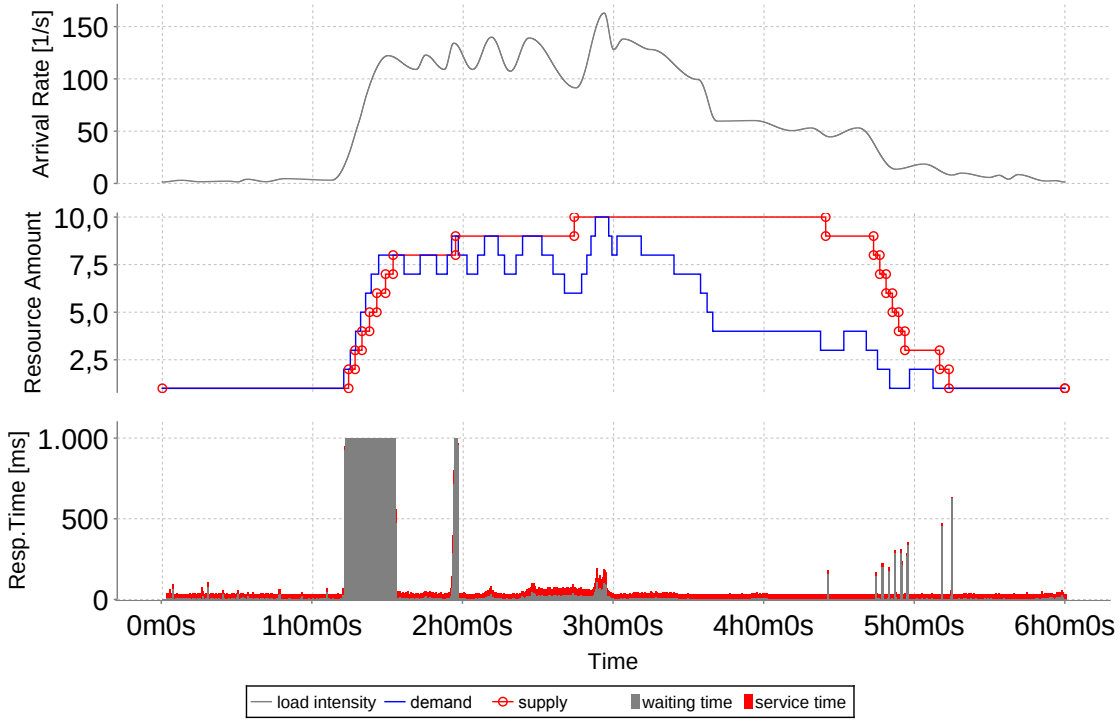


(b) System using Offering B exposed to load intensity profile adjusted for Offering D. Demand curve is equal to the demand curve for Offering A (Figures 7.11,7.12)

Figure 7.13: Not adjusting the load profile system specific leads to a elasticity behavior that cannot be compared easily with other elasticity measurements (1).



(a) System using Offering D exposed to load intensity profile adjusted for Offering A. Demand curve is not equal to the demand curve for Offering A (Figures 7.11,7.12)



(b) System using Offering D exposed to load intensity profile adjusted for Offering D. Demand curve is equal to the demand curve for Offering A (Figures 7.11,7.12)

Figure 7.14: Not adjusting the load profile system specific leads to a elasticity behavior that cannot be compared easily with other elasticity measurements (2).

When the load profile is correctly adjusted in contrast (Figure 7.13(b)), the same resource demand changes are induced at the same point in time as for Offering A (Figure 7.12(c)) and for Offering B (Figure 7.14(b)).

Table 7.18 shows the different elasticity submetrics as well as the overall *elastic speedup* metric for elasticity parameter Configuration F and different combinations of unadjusted and correctly adjusted load profiles for the Offerings A, B and D. As a reference, the first row shows the metric values for Offering A with an correctly adjusted load profile.

Parameter Config. / Offering / Adjusted for Off.	$accuracy_O$ [res. units]	$accuracy_U$ [res. units]	$timeshare_O$ [%]	$timeshare_U$ [%]	$jitter$ $\left[\frac{\#adap.}{min}\right]$	**elastic speedup**
F / A / A	2,423	0.067	66.1	4.8	-0.067	1.046
F / B / A	1.811	0.001	63.8	0.1	-0.033	1.291
F / B / B	2.508	0.061	67.1	4.5	-0.044	1.025
F / D / A	4.190	1.154	53.5	21.0	-0.139	0.696
F / D / D	1.770	0.076	52.2	5.7	-0.067	1.344

Table 7.18: Metric results for offerings A, B & D with (white background) and without (lightgrey background) adapted load profile. Not adapting the load profile to the service offering shows significantly changed elasticity metrics.
Elastic speedup (Formula 5.1) for weights:
$w_{acc_U} = w_{acc_O} = w_{ts_U} = w_{ts_O} = w_{acc} = w_{ts} = 0.5$

The second/fourth row shows the metric values for using Offering B/D, but keeping the load profile adjusted for Offering A. The third/fifth row in contrast shows the metric values for an correctly adjusted load profile.

When the same load profile is used for comparing the behavior for Offering A to that for Offering B, all metrics exhibit better results for Offering B. In contrast, if the properly adjusted load profile is used, the metrics for Offering B are almost equal to those for Offering A.

A similar effect can be observed when comparing the results for Offering D using a load profile adjusted for Offering A with using a load profile correctly adjusted for Offering D. However, the deviation of the metric results between Offering A and D for correctly adjusted load profiles are bigger than those between Offering A and B with correctly adjusted load profiles. This indicates that CloudStack seems to be able to match the resource demand better for the smallest service Offering D compared to the bigger service Offerings A and B. Without adjusting the load profile the metric values would indicate the opposite effect.

7.4.2 Public Cloud - Amazon Web Services

This subsection demonstrates how the benchmark can be used to evaluate different elasticity parameter configurations of the Amazon Elastic Compute Cloud (EC2) IaaS service. EC2 is one of the central products of Amazon's cloud computing platform Amazon Web Services (AWS).

7.4.2.1 Setup

The configuration of the benchmark harness stays identical to the one described in Section 7.1.3. The next paragraphs describe the configuration of the public cloud, which is similar to the private cloud, but has some additional configuration options.

Elasticity Mechanism

The AWS platform offers customers the option to build clouds consisting of a group (*Auto Scaling Group*) of VMs and scale them horizontally according to a rule based elasticity strategy. The elasticity mechanism can be configured by the customer and offers the same parameters as the one offered by CloudStack. In addition, AWS allows to create rules that trigger the concurrent de-/allocation of multiple instances. In contrast to CloudStack, the minimum time for the *condTrueDurUp/Down* parameter is 60s for the AWS rules. Similar to CloudStack, Amazon allows to configure a load balancer that distributes incoming load to the instances in the *Auto Scaling Group*.

Instance Type

AWS offers various instance types that are designed to meet the specific requirements of its customers. Besides general purpose instances, instances optimized for memory, storage, or processing are available at different sizes. For the case study, the instance type *m1.small* has been chosen. This general purpose instance type offers one virtual CPU with a processing power equivalent to an 1.0-1.2 GHz 2007 Opteron processor and 1.7 GB RAM.

Regions

Different *Regions* within which instances and load balancers can be located, allow customers to create clouds geographically close to where they expect their customers. Since, the load driver and benchmark node is located in europe, the region *EU (Ireland)* has been chosen for the case study in order to keep the network latency small.

Health Check Parameters

AWS allows to configure the health check for the load balancer with the same parameters as CloudStack does. AWS restricts the minimum values for the parameters *healthyThreshold* and *unhealthyThreshold* to two and the minimum value for the *healthyResponseTimeout* to two seconds. It is therefore not possible to use the same health check parameters that have been used for the private cloud evaluations. The used health check parameters for the case study on AWS are shown in Table 7.19.

Name	Default	Description
pingPath	/?size=1	address which is queried to check the instance health
healthyResponseTimeout	2s	period of time within that a response is expected from healthy instances
healthCheckInterval	5s	time between two consecutive health checks
healthyThreshold	2	number of subsequent health checks successes before instance is declared healthy
unhealthyThreshold	2	number of subsequent health checks failures before instance is declared unhealthy

Table 7.19: AWS health check parameters

7.4.2.2 Load Profile

In order to allow comparability, the same load profile as for the private cloud case study has been used. Within the calibration, it is adjusted according to the load processing capabilities of the underlying resources.

7.4.2.3 Calibration

As for the private cloud case study, the public AWS cloud is configured to use ten resources at maximum. The cloud is analyzed using the *Detailed System Analysis*. In order to evaluate how reproducible the analysis is for a public cloud, the analysis is done eight times. Table 7.20 shows the analysis results.

used resources	max intensity (averaged) [req./sec.]	max deviation from averaged intensity [%]	deviation of the linearity assumption [%]
1	70.9	4.1	
2	142.8	3.3	-0.7
3	215.1	2.4	-1.16
4	287.6	4.4	-1.43
5	360.4	4.3	-1.66
6	433.8	2.0	-1.96
7	599.6	2.5	-0.70
8	560.8	5.8	1.11
9	595.7	6.8	7.08
10	690.3	6.8	2.68

Table 7.20: *Detailed System Analysis* results for AWS Offering m1.small

The second column shows the maximum intensity that the system could withstand without violating the SLO for one up to ten resources. The presented results are averaged results for a total of eight measurements. The third column contains the maximum deviation from the averaged result that was observed during the measurements. It can be seen that the deviation is below 10% but tends to increase with the number of resources. The third column shows the deviation of the linearity assumption from the averaged intensities for using two resources up to using ten resources. The linearity assumption presumes that the resource demand is increasing linearly with the load intensity. Except for nine resources the deviation from the linearity assumption is always below 3%. Since the analysis results deviate up to 6.8% when nine resources are used, it can be assumed that the increased deviation from the linearity assumption is mainly due to measurement inaccuracy.

The load profile is therefore calibrated with a mapping function *demand*(*intensity*) that assumes a maximum load intensity of 71 requests per second for one resource and a linearly increased intensity for two or more resources. The load intensity within the calibrated load profile varies between five and 710 requests per second. The timestamp file created within the measurement process contains about 5.6 million timestamps.

7.4.2.4 Elasticity Rule Parameter Configurations

The resource elasticity of the AWS cloud is evaluated for elasticity rule parameter settings illustrated in Table 7.21.

The two new parameters *instAdd* and *instRem* define the number of instances that are triggered to be allocated or respectively deallocated concurrently by the corresponding elasticity rule.

Configuration G uses a small *thresholdDown* parameter. This leads to late scaled downs and induces a scaling behavior with a significant amount of over-provisioning.

Config.	quiet-Time	condTrue-DurUp	condTrue-DurDown	thresh-oldUp	thresh-oldDown	instAdd	instRem
G	120s	60s	60s	50%	30%	1	1
H	120s	60s	60s	65%	50%	1	1
I	60s	60s	60s	65%	40%	1	1
J	60s	60s	60s	80%	50%	1	1
K	60s	60s	60s	80%	50%	3	1

Table 7.21: Different elasticity parameter configurations for AWS

Configuration H is very similar to Configuration D used for CloudStack. Compared to Configuration G, it induces earlier resource increases but later resource decreases.

Configuration I is very similar to Configuration E used for CloudStack. A smaller *quietTime* induces faster reactions to demand changes compared to Configurations G and H.

Configuration J tries to provoke late scale ups by using a higher *thresholdUp* compared to Configurations G-I.

Configuration K triggers the allocation of three instances for every scale up and tries to compensate the high *thresholdUp* this way.

7.4.2.5 Elasticity Measurements

Visualization of Demand, Supply and Response Time

The Figures 7.15 and 7.16 show the resource demand curve and the measured resource supply curve for the five different elasticity rule configurations G-K. These curves give a visual impression of the different elastic behaviors. Under each resource allocation graph a response time graph shows how the response times vary during the measurement runs. It allows to estimate the amount of SLO violations visually.

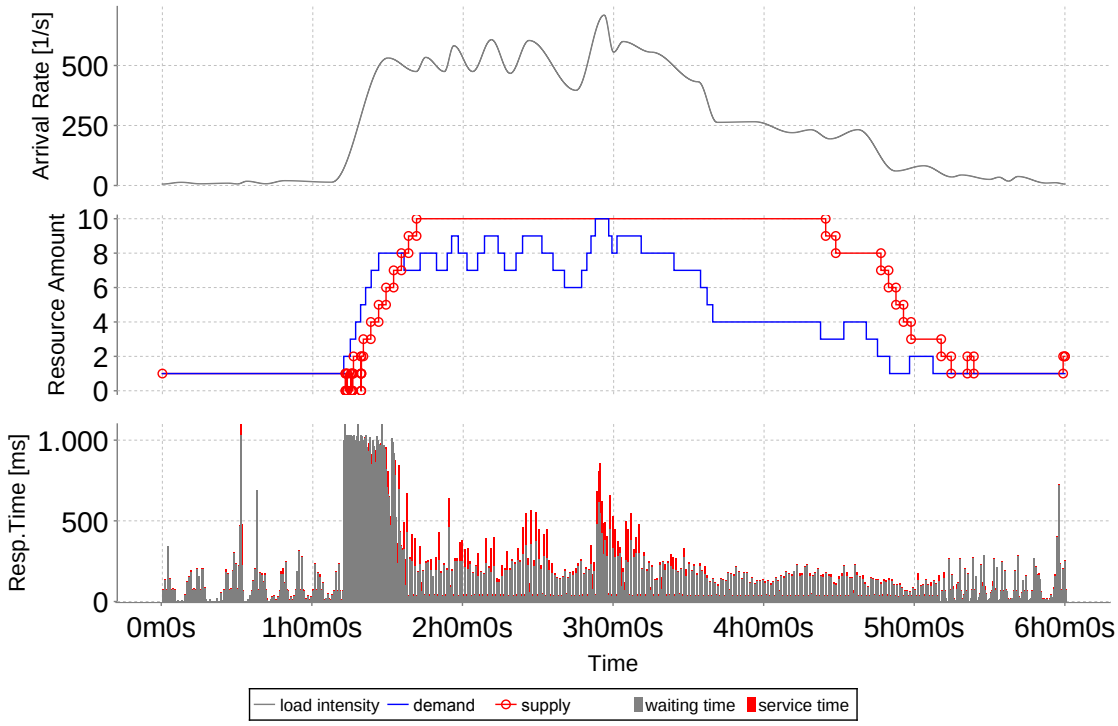
The response time graphs show higher and more variable response times for AWS compared to CloudStack. Possible reasons are a higher performance variability on the public cloud as well as network delays.

While comparing the graphs for the Configurations G-K, the graph for Configuration J stands out immediately. The resource supply jumps between one and zero resources very often instead of scaling up to ten resources. Since the *thresholdUp* parameter was set to 80% and the *condTrueDurUp* parameter was set to 60s for this configuration, the observed behavior means that the average CPU utilization was greater than 80% for more than one minute only in a few times. One explanation for this observation is the health check of the AWS load balancer: Whenever an instance is marked unhealthy because it does not respond to requests fast enough, no requests are forwarded to the instance anymore until the instance is able to answer requests fast enough again. During this period of time, the CPU utilization of the instance is low and thus the average CPU utilization decreases below 80% and as result the elasticity mechanism does not trigger a scale up.

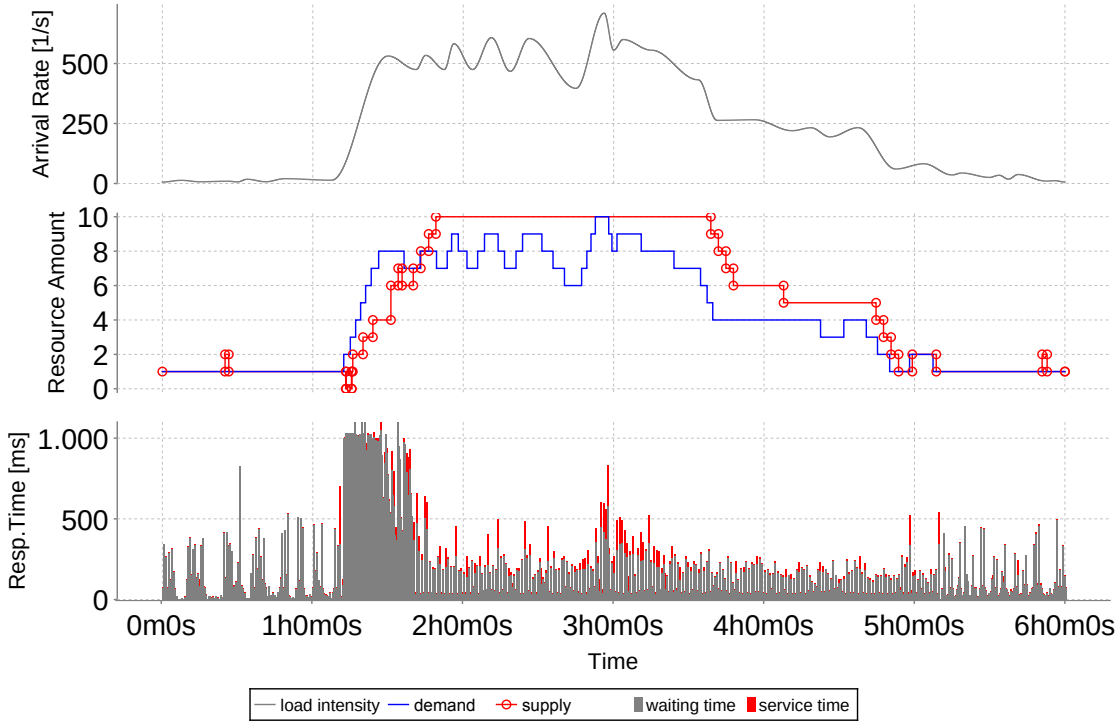
Metric Result Discussion

Table 7.22 extends Table 7.14 and shows the metric results for both, the configurations evaluated on the private CloudStack cloud and those evaluated on the AWS cloud.

Among the AWS Configurations G-K, Configuration G uses the lowest *thresholdDown* value. As a result, Configuration G has the worst $accuracy_O$ metric within this group.

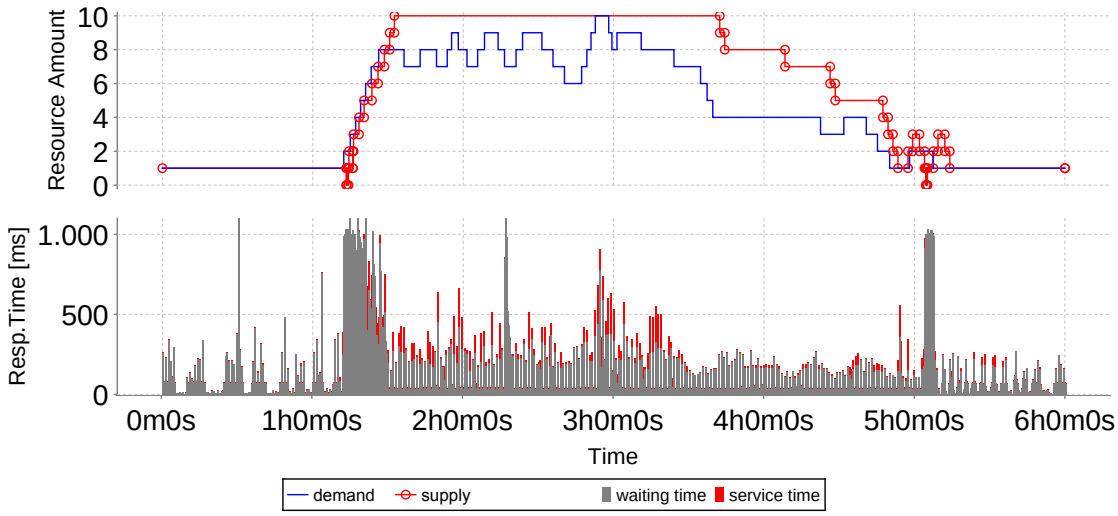


(a) Configuration G: quietTime: 120s, condTrueDurUp/Down: 60s, thresholdUp: 50%, thresholdDown: 30%, instAdd/Rem: 1
Late resource decreases, due to small thresholdDown

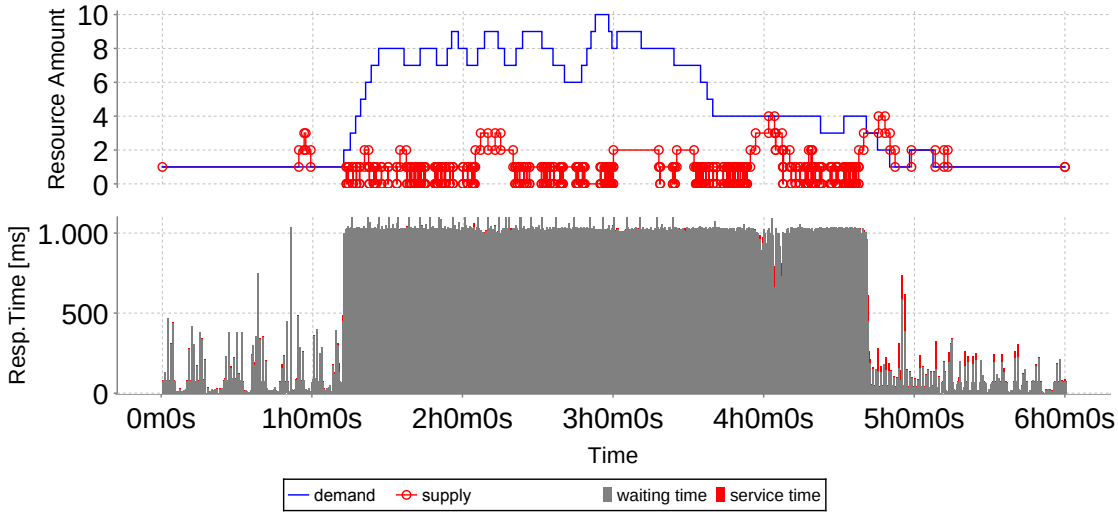


(b) Configuration H: quietTime: 120s, condTrueDurUp/Down: 60s, thresholdUp: 65%, thresholdDown: 50%, instAdd/Rem: 1
Later resource increase, earlier resource decrease compared to Configuration G, similar parameters as in Configuration D

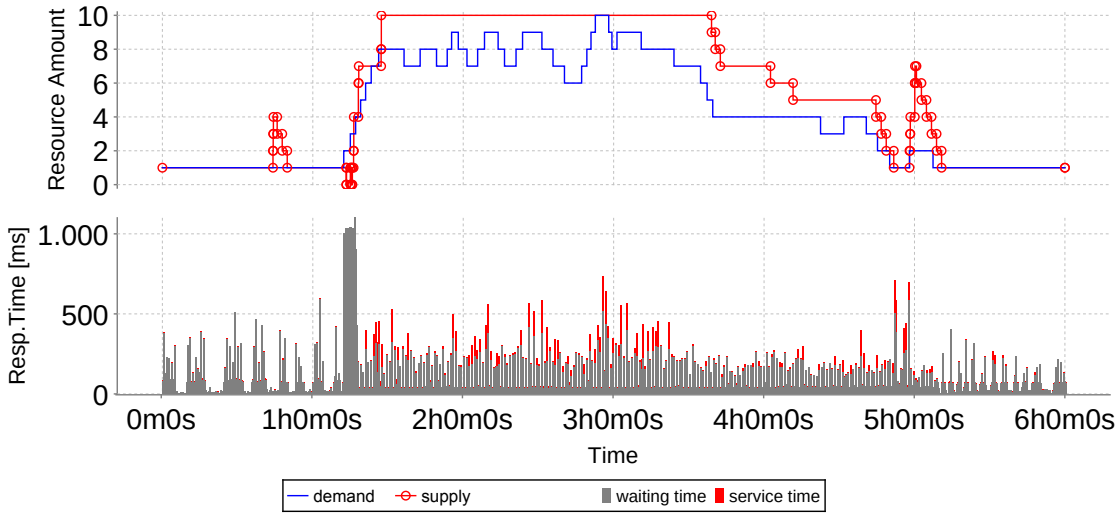
Figure 7.15: Elastic behavior for different elasticity rule parameter settings on AWS (1)



(a) Configuration I: quietTime: 60s, condTrueDurUp/Down: 60s, thresholdUp: 65%, thresholdDown: 40%, instAdd/Rem: 1
Faster resource in- and decrease compared to Configurations G & H, similar parameters as in Configuration E



(b) Configuration J: quietTime: 60s, condTrueDurUp/Down: 60s, thresholdUp: 80%, thresholdDown: 50%, instAdd/Rem: 1
High scale up threshold, scale up rule is not triggered as expected, lots of under-provisioning



(c) Configuration K: quietTime: 120s, condTrueDurUp/Down: 60s, thresholdUp: 80%, thresholdDown: 50%, instAdd: 3, instRem: 1
Scale up adds three instances at a time, almost no SLO violations

Figure 7.16: Elastic behavior for different elasticity rule parameter settings on AWS (2)

Configuration I uses the second lowest *thresholdDown* value which also leads to the second worst $accuracy_O$ metric.

Parameter Configuration	$accuracy_O$ [res. units]	$accuracy_U$ [res. units]	$timeshare_O$ [%]	$timeshare_U$ [%]	$jitter$ $\left[\frac{\#adap.}{min}\right]$
A	2.425	0.264	60.1	11.7	-0.067
B	0.664	0.224	40.6	10.6	-0.056
C	0.219	0.383	15.4	23.4	0.006
D	0.815	0.080	48.7	6.5	-0.028
E	1.184	0.006	56.0	0.6	-0.061
F	2.423	0.067	66.1	4.8	-0.067
G	1.985	0.137	60.1	6.4	-0.025
H	1.053	0.180	51.9	8.1	-0.033
I	1.442	0.049	57.6	4.7	-0.017
J	0.049	2.984	3.9	57.5	0.644
K	1.340	0.019	61.6	1.4	0.000

Table 7.22: Metric results for all evaluated configurations on CloudStack (A-F) and on AWS (G-K)

Configuration H scales up later and down earlier than Configuration G. This behavior leads to slightly worse $accuracy_U$ and $timeshare_U$ metrics and a significantly improved $accuracy_O$ metric for Configuration G. The $timeshare_O$ metric improves slightly, too.

Due to a decreased *quietTime* for Configuration I compared to Configurations G and H, the AWS cloud can scale up resources faster and therefore improve the $accuracy_U$ and the $timeshare_U$ metric. An *thresholdDown* value between those of Configurations G and H leads for Configuration I to $accuracy_O$ and $timeshare_O$ metrics values better than for Configuration G but worse than for Configuration H.

The bad $accuracy_U$ metric for Configuration J is a result of the massive under-provisioning that occurs during the measurement period. The frequent jumps between a resource supply of one and zero resources is reflected in a comparably high *jitter* metric.

The last Configuration K exhibits very good results for the $accuracy_U$ and $timeshare_U$ metrics. These are a result of the large scale ups that can compensate sharp demand increases very well. However, those large scale ups are not always appropriate and lead to unnecessary over-provisioning in some cases. Therefore, the $accuracy_O$ and $timeshare_O$ metrics are worse than for Configuration H, although both have a similar scale down behavior. At the beginning and at the end of the measurement period, the test system shows superfluous adaptations when Configuration K is used. In the middle section in contrast, the system uses ten resources for a long period of time and does not follow the resource demand adaptations. Since the superfluous and the missing adaptations of the resource supply compensate each other, the *jitter* metric evaluates to zero. This compensation of superfluous and missing supply adaptations is unwanted and the *jitter* metric should therefore be improved as part of future work.

7.4.2.6 Aggregated Elasticity Measure

The Tables 7.23 and 7.24 show the parameter configurations for CloudStack and AWS ordered by the *elastic speedup* metric that aggregates the elasticity submetrics according to Formula 5.1.

Parameter Configuration	*accuracy weighted* [res. units]	*timeshare weighted* [%]	*accuracy speedup*	*timeshare speedup*	**elastic speedup**	SLO violations [%]
C	0.301	19.4	4.467	1.851	2.875	41.2
B	0.444	25.6	3.028	1.402	2.061	17.8
D	0.448	27.6	3.004	1.301	1.977	8.4
E	0.595	28.3	2.260	1.269	1.693	0.7
H	0.617	30.0	2.181	1.197	1.615	9.1
K	0.680	31.5	1.979	1.140	1.502	2.5
I	0.746	31.2	1.803	1.152	1.442	5.0
G	1.061	33.3	1.267	1.080	1.170	7.8
F	1.245	35.5	1.080	1.013	1.046	7.6
J	1.517	30.7	0.887	1.169	1.018	93.2
A	1.345	35.9	1.000	1.000	1.000	20.3

Table 7.23: *Elastic speedup* (Formula 5.1) for all configurations and equal weights: $w_{acc_U} = w_{acc_O} = w_{ts_U} = w_{ts_O} = w_{acc} = w_{ts} = 0.5$

Its notable that Configuration J, which does not scale up as expected, has still a slightly better *elastic speedup* metric than the baseline Configuration A when the accuracy and timing metrics are weighted equally for both, over-provisioning and under-provisioning scenarios (Table 7.23). This is due to the fact that for Configuration J the bad $accuracy_U$ metric is compensated by a good $accuracy_O$ metric.

The weights used in Table 7.24 stress the importance of a good $timeshare_U$ metric. Therefore, Configuration J is ranked at the last position in Table 7.24. Its notable that Configuration K, which scales up three resources at a time, is ranked at position three. This indicates that triggering the allocation of several instances at a time can compensate a slow reaction to fast demand changes quite well.

7.4.3 Discussion

This section used a realistic load profile to evaluate different elasticity rule parameter configurations on a private and a public cloud. The different elasticity metrics have been aggregated to a single elasticity measure. Hereby, the influence of different preferences that result in different weights for aggregating the metrics, was demonstrated. Furthermore, the importance of adjusting load profiles according to the load processing capabilities of a system was demonstrated. Additionally, the usage of realistic load profiles showed additionally the difficulty of configuring elasticity mechanisms that use a rule based elasticity strategy and do not or at least seldom violate SLOs without over-provisioning resources most over the time.

Parameter Configuration	*accuracy weighted* [res. units]	*timeshare weighted* [%]	*accuracy speedup*	*timeshare speedup*	**elastic speedup**	SLO violations [%]
E	0.948	11.7	2.101	1.830	1.882	0.7
D	0.668	14.9	2.983	1.431	1.658	8.4
K	1.076	13.4	1.852	1.591	1.640	2.5
B	0.576	16.6	3,460	1.288	1.569	17.8
C	0.252	21.8	7.914	0.981	1.489	41.2
I	1.163	15.3	1.713	1.399	1.457	5.0
H	0.878	16.9	2.269	1.268	1.425	9.1
G	1.615	17.1	1.234	1.247	1.245	7.8
F	1.952	17.1	1.021	1.253	1.203	7.6
A	1.993	21.4	1.000	1.000	1.000	20.3
J	0.636	46.8	3.133	0.457	0.672	93.2

Table 7.24: *Elastic speedup* (Formula 5.1) for all configurations and weights: $w_{acc_U} = 0.2, w_{acc_O} = 0.8, w_{ts_U} = 0.8, w_{ts_O} = 0.2, w_{acc} = 0.2, w_{ts} = 0.8$

8. Future Work

In course of this thesis a concept as well as a basic framework for benchmarking resource elasticity has been developed. The approach allows to measure different aspects of elasticity separately and accounts for different levels of efficiency of underlying resources and different scaling behaviors of systems. Beyond the scope of this thesis, the approach can be further extended and evaluated as described in the following sections.

8.1 Further Evaluations

For the experiments within this thesis, the benchmarking framework has been configured to use equal distance sampling when creating timestamp files for a given load profile. The impact of other sampling methods such as uniform distribution sampling on calibration and/or measurement results can be evaluated as part of future work.

A further step towards load profiles that reflect the load intensity in real cloud applications better bases on adding noise onto the modeled load intensity. The impact of adding/removing noise from a load profile is a starting point for future evaluations.

8.2 Extensions of the Benchmark

The current implementation of the benchmarking framework is capable of inducing resource demands on the CPU. In order to allow resource elasticity benchmarking for other resource types such as memory, disk storage or networking resources, the server-side load processing component has to be extended as described in Section 6.2.2.

Within this approach, requests always trigger constant resource demands on a single resource. In a realistic cloud scenario, different requests can trigger resource demands of different sizes. Furthermore, requests trigger demands on several resources. This can be taken into account by extending the benchmark concept as follows:

- Workloads which consist of different requests can be modeled with a set of load profiles. Every load profile models the load intensity for a single request type. The resource demand induced by the request mix can then be described with a multiparameter demand function: $demand(intensity\_request_1, ..., intensity\_request_n)$.
- A request that triggers demands on several resources can be modeled with separate demand functions for every resource: $demand_{CPU}(intensity)$, $demand_{MEM}(intensity)$, $demand_{IO}(intensity)$.

Additionally, both approaches can be combined.

Another possible extension is to use closed or partially open workloads instead of open workloads. When a closed workload model is used, the load intensity characterizes the number of users instead of the of the arrival rate.

Currently, the benchmark supports horizontal scaling as scaling method. Given that the cloud management software supports vertical scaling, this scaling method can also be evaluated with the presented approach. Evaluating the resource elasticity of systems that use horizontal and vertical scaling at the same time adds additional complexity. The amount of work a system is able to handle can be different depending on how much horizontal or vertical scaling is used, even when the total amount of resources is equal. Two VMs with each two CPUs do not necessarily have the same load processing capabilities as one VM with four CPUs. Thus, it is difficult to derive an unambiguous resource demand.

At the moment, the benchmarking framework allows to compare the resource elasticity of different CSUTs or different resource elasticity parameter configurations of one CSUT provided that CloudStack or AWS is used as cloud management software. Extending the benchmark in order to support other cloud management softwares is part of future work. The benchmark design allows such extensions easily.

For small scale elasticity measurements, using a single load driver is enough for generating the load. However, for large scale elasticity measurements, the capabilities of a single machine may not be sufficient for generating the necessary loads. JMeter supports the distributed generation of loads. Thus, extending the TimestampTimer Plugin to support distributed load generation would allow large scale elasticity measurements.

Section 5.4.3 sketched the use of the benchmarking concept for evaluating the financial impacts of resource elasticity. Possible future research directions include a refined analysis of the financial aspects of elasticity as well as a thorough investigation of the links between the elasticity evaluation from a technical perspective and the evaluation from a business oriented perspective.

8.3 Other Considerations

It is not in scope of this thesis to define a set of representative load profiles for an industry standardized benchmark. Nevertheless, a load profile has been selected that exhibits a number of important characteristics and can therefore be suggested to be included in a set of representative load profiles.

In order to make elasticity measurements cheap, the duration of measurement runs should be limited as far as possible. However, when a realistic load profile is compacted, the demand curve is compacted, too. This makes it more difficult to allocate resources on time. At some point, the CSUT is not able to allocate resources on demand at all and elasticity comparisons are not possible anymore. Thus, investigations towards how much experiment compression is sensible are valuable for future work.

9. Conclusion

Today's cloud infrastructure platforms often provide auto-scaling features that enable the dynamic allocation of resources over time according to a varying demand. An elasticity benchmark helps to analyze this elastic behavior and allows to compare different cloud platforms. Furthermore, a benchmark can make the influence of different elasticity mechanism parameter configurations visible and allows researchers to compare newly developed elasticity strategies to existing ones.

As a basis for the development of a new benchmark, this thesis has discussed the meaning of elasticity in the cloud context and has differentiated it from the related terms efficiency and scalability. In a second step, existing approaches for evaluating resource elasticity have been analyzed with respect to their focus, strengths and weaknesses. Many of these approaches focus on measuring the financial impact of different degrees of elasticity for a cloud customer. While this perspective may be valid in some scenarios, it mixes up the evaluation of the technical property elasticity and the business model of the provider. Other approaches analyze the technical aspects of elasticity only to a limited extend or do not account for different levels of efficiency of underlying resources when evaluating elasticity.

The benchmarking concept that has been developed in course of this thesis as a third step, allows together with a set of refined elasticity metrics the evaluation of the technical property resource elasticity and takes explicitly into account the scaling behavior and the efficiency of underlying resources of the benchmarked system. This has been achieved by partitioning the process of benchmarking into *System*, *Calibration*, *Measurement* and *Evaluation* activities. Within a *System Analysis*, the benchmark evaluates the scaling behavior and the efficiency of the underlying resources. The result is used during the *Benchmark Calibration* which adjusts a given load profile in a way that the same resource demand changes are induced on all compared systems independently of their underlying hardware performances. During the *Measurement*, the cloud system under test (CSUT) is exposed to a load varying according to the adjusted load profile. The final *Elasticity Evaluation* compares the monitored variations of the resource allocations with the induced resource demand utilizing a set of metrics specifically designed to capture the elasticity aspects accuracy and timing.

In a further step, the concept has been implemented in an benchmarking framework called *BUNGEE*. The framework allows to use realistic load profiles as input and automates the different activities of the benchmarking process. Load profiles can be modeled

or be parsed from existing traces using the LIMBO toolkit developed by V. Kistowski [vKHK14b]. Currently, *BUNGEE* supports to evaluate the resource elasticity of horizontally scaling clouds based on CloudStack or Amazon Web Services (AWS).

An evaluation of the resource elasticity metrics has demonstrated their ability to rank elastic systems on an ordinal scale. In addition, the reproducibility of the benchmark calibration has been positively evaluated on a private CloudStack based cloud. A case study conducted on the private cloud as well as on an AWS based cloud has demonstrated the applicability of the benchmark for realistic scenarios. Within the case study, eleven elasticity rule configurations and four instance types with different levels of efficiency have been evaluated.

Bibliography

[AFG+10] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721672

[ASLM13] R. F. Almeida, F. R. Sousa, S. Lifschitz, and J. C. Machado, "On Defining Metrics for Elasticity of Cloud Databases," in *Proceedings of the 28th Brazilian Symposium on Databases*, 2013. [Online]. Available: http://sbbd2013.cin.ufpe.br/Proceedings/artigos/sbbd_shp_12.html

[BBD+14] M. M. Bersani, D. Bianculli, S. Dustdar, A. Gambi, C. Ghezzi, and S. Krstić, "Towards the Formalization of Properties of Cloud-based Elastic Systems," in *Proceedings of the 6th International Workshop on Principles of Engineering Service-Oriented and Cloud Systems*, ser. PESOS 2014. New York, NY, USA: ACM, 2014, pp. 38–47. [Online]. Available: http://doi.acm.org/10.1145/2593793.2593798

[BKKL09] C. Binnig, D. Kossmann, T. Kraska, and S. Loesing, "How is the Weather Tomorrow?: Towards a Benchmark for the Cloud," in *Proceedings of the Second International Workshop on Testing Database Systems*, ser. DBTest '09. New York, NY, USA: ACM, 2009, pp. 9:1–9:6. [Online]. Available: http://doi.acm.org/10.1145/1594156.1594168

[BLY+10] A. Beitch, B. Liu, T. Yung, R. Griffith, A. Fox, and D. A. Patterson, "Rain: A Workload Generation Toolkit for Cloud Computing Applications," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-14, Feb 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-14.html

[Bre12] P. C. Brebner, "Is your Cloud Elastic Enough? Performance Modeling the Elasticity of Infrastructure as a Service (IaaS) Cloud Applications," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 263–266. [Online]. Available: http://doi.acm.org/10.1145/2188286.2188334

[CCB+12] D. Chandler, N. Coskun, S. Baset, E. Nahum, S. R. M. Khandker, T. Daly, N. W. I. Paul, L. Barton, M. Wagner, R. Hariharan, and Y. seng Chao, "Report on Cloud Computing to the OSG Steering Committee," Tech. Rep., Apr. 2012. [Online]. Available: http://www.spec.org/osgcloud/docs/osgcloudwgreport20120410.pdf

[CGS13] E. F. Coutinho, D. G. Gomes, and J. N. d. Souza, "An Analysis of Elasticity in Cloud Computing Environments Based on Allocation Time and Resources," in *Cloud Computing and Communications (LatinCloud), 2nd IEEE Latin American Conference on*, Dec 2013, pp. 7–12.

[CW09] A. C. Chiang and K. Wainwright, *Fundamental Methods of Mathematical Economics*, 4th ed. McGraw-Hill, 2009.

[DMRT11] T. Dory, B. Mejías, P. V. Roy, and N.-L. Tran, "Measuring Elasticity for Cloud Databases," in *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*, 2011. [Online]. Available: http://www.info.ucl.ac.be/~pvr/CC2011elasticityCRfinal.pdf

[DRW06] L. Duboc, D. S. Rosenblum, and T. Wicks, "A Framework for Modelling and Analysis of Software Systems Scalability," in *Proceedings of the 28th international conference on Software engineering*, ser. ICSE '06. New York, NY, USA: ACM, 2006, pp. 949–952. [Online]. Available: http://doi.acm.org/10.1145/1134285.1134460

[FAS+12] E. Folkerts, A. Alexandrov, K. Sachs, A. Iosup, V. Markl, and C. Tosun, "Benchmarking in the Cloud: What It Should, Can, and Cannot Be," in *Selected Topics in Performance Evaluation and Benchmarking*, ser. Lecture Notes in Computer Science, R. Nambiar and M. Poess, Eds. Springer Berlin Heidelberg, 2012, vol. 7755, pp. 173–188. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-36727-4_12

[FHA99] E. Freeman, S. Hupfer, and K. Arnold, *JavaSpaces Principles, Patterns, and Practice*, ser. Java series. Addison-Wesley, 1999.

[FW86] P. J. Fleming and J. J. Wallace, "How Not to Lie with Statistics: The Correct Way to Summarize Benchmark Results," *Commun. ACM*, vol. 29, no. 3, pp. 218–221, Mar. 1986. [Online]. Available: http://doi.acm.org/10.1145/5666.5673

[GB12] G. Galante and L. C. E. d. Bona, "A Survey on Cloud Computing Elasticity," in *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*, ser. UCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 263–270. [Online]. Available: http://dx.doi.org/10.1109/UCC.2012.30

[Hal08] E. H. Halili, *Apache JMeter: A Practical Beginner's Guide to Automated Testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[Her11] N. R. Herbst, "Quantifying the Impact of Configuration Space for Elasticity Benchmarking," Study Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2011. [Online]. Available: http://sdqweb.ipd.kit.edu/publications/pdfs/Herbst2011a.pdf

[HHKA14] N. R. Herbst, N. Huber, S. Kounev, and E. Amrehn, "Self-adaptive Workload Classification and Forecasting for Proactive Resource Provisioning," *Concurrency and Computation: Practice and Experience*, pp. n/a–n/a, 2014. [Online]. Available: http://dx.doi.org/10.1002/cpe.3224

[HKR13] N. R. Herbst, S. Kounev, and R. Reussner, "Elasticity in Cloud Computing: What it is, and What it is Not (Short Paper)," in *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 2013)*. USENIX, June 2013. [Online]. Available: https://www.usenix.org/conference/icac13/elasticity-cloud-computing-what-it-and-what-it-not

[Hup09] K. Huppler, "Performance Evaluation and Benchmarking," R. Nambiar and M. Poess, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. The Art of Building a Good Benchmark, pp. 18–30. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-10424-4_3

[Hup12] K. Huppler, "Benchmarking with Your Head in the Cloud," in *Topics in Performance Evaluation, Measurement and Characterization*, ser. Lecture Notes in Computer Science, R. Nambiar and M. Poess, Eds. Springer Berlin Heidelberg, 2012, vol. 7144, pp. 97–110. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-32627-1_7

[ILFL12] S. Islam, K. Lee, A. Fekete, and A. Liu, "How a Consumer Can Measure Elasticity for Cloud Platforms," in *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*, ser. ICPE '12. New York, NY, USA: ACM, 2012, pp. 85–96. [Online]. Available: http://doi.acm.org/10.1145/2188286.2188301

[Ins13] C. A. Insitute, "Cloud Solutions Best Practices: 2013 Benchmark Study: Results and Analysis from the Cloud Accounting Institute," 2013. [Online]. Available: http://learn.amllp.com/2013-cloud-benchmark-study

[JS14] B. Jennings and R. Stadler, "Resource Management in Clouds: Survey and Research Challenges," *Journal of Network and Systems Management*, pp. 1–53, 2014. [Online]. Available: http://dx.doi.org/10.1007/s10922-014-9307-7

[KHvKR11] M. Kuperberg, N. R. Herbst, J. G. von Kistowski, and R. Reussner, "Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms," Karlsruhe Institute of Technology (KIT), Tech. Rep., 2011. [Online]. Available: http://digbib.ubka.uni-karlsruhe.de/volltexte/1000023476

[LOZC12] Z. Li, L. O'Brien, H. Zhang, and R. Cai, "On a Catalogue of Metrics for Evaluating Commercial Cloud Services," in *Grid Computing (GRID), 2012 ACM/IEEE 13th International Conference on*, Sept 2012, pp. 164–173. [Online]. Available: http://dx.doi.org/10.1109/Grid.2012.15

[LYKZ10] A. Li, X. Yang, S. Kandula, and M. Zhang, "CloudCmp: Comparing Public Cloud Providers," in *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement*, ser. IMC '10. New York, NY, USA: ACM, 2010, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/1879141.1879143

[Mah36] P. C. Mahalanobis, "On the Generalized Distance in Statistics," in *Proceedings National Institute of Science, India*, vol. 2, no. 1, Apr. 1936, pp. 49–55. [Online]. Available: http://www.new.dli.ernet.in/rawdataupload/upload/insa/INSA_1/20006193_49.pdf

[Man64] J. Mandel, *The Statistical Analysis of Experimental Data*. Dover, 1964.

[MCTD13] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "MELA: Monitoring and Analyzing Elasticity of Cloud Services," in *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, vol. 1, Dec 2013, pp. 80–87. [Online]. Available: http://dx.doi.org/10.1109/CloudCom.2013.18

[MG11] P. M. Mell and T. Grance, "The NIST Definition of Cloud Computing," Gaithersburg, MD, United States, Tech. Rep., 2011. [Online]. Available: http://csrc.nist.gov/publications/PubsSPs.html#800-145

[MJ98] D. Mosberger and T. Jin, "Httperf - a Tool for Measuring Web Server Performance," *SIGMETRICS Perform. Eval. Rev.*, vol. 26, no. 3, pp. 31–37, Dec. 1998. [Online]. Available: http://doi.acm.org/10.1145/306225.306235

[OED14a] "Efficient — Oxford English Dictionary Online," 2014, (accessed July 9, 2014). [Online]. Available: http://www.oxforddictionaries.com/definition/english/efficient

[OED14b] "Elasticity — Oxford English Dictionary Online," 2014, (accessed July 9, 2014). [Online]. Available: http://www.oxforddictionaries.com/definition/english/elasticity

[RR10] T. Rauber and G. Rünger, *Parallel Programming - for Multicore and Cluster Systems.* Springer, 2010.

[SA12] D. Shawky and A. Ali, "Defining a Measure of Cloud Computing Elasticity," in *Systems and Computer Science (ICSCS), 2012 1st International Conference on*, Aug 2012, pp. 1–5. [Online]. Available: http://dx.doi.org/10.1109/IConSCS.2012.6502449

[SC07] S. Salvador and P. Chan, "Toward Accurate Dynamic Time Warping in Linear Time and Space," *Intell. Data Anal.*, vol. 11, no. 5, pp. 561–580, Oct. 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1367985.1367993

[SMC+08] P. Shivam, V. Marupadi, J. Chase, T. Subramaniam, and S. Babu, "Cutting Corners: Workbench Automation for Server Benchmarking," in *USENIX 2008 Annual Technical Conference on Annual Technical Conference*, ser. ATC'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 241–254. [Online]. Available: http://dl.acm.org/citation.cfm?id=1404014.1404032

[SSSS11] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, "A Cost-Aware Elasticity Provisioning System for the Cloud," in *Distributed Computing Systems (ICDCS), 2011 31st International Conference on*, June 2011, pp. 559–570. [Online]. Available: http://dx.doi.org/10.1109/ICDCS.2011.59

[ST09] M. Salehie and L. Tahvildari, "Self-adaptive Software: Landscape and Research Challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, no. 2, pp. 14:1–14:42, May 2009. [Online]. Available: http://doi.acm.org/10.1145/1516533.1516538

[Sul12] B. Suleiman, "Elasticity Economics of Cloud-Based Applications," in *Proceedings of the 2012 IEEE Ninth International Conference on Services Computing*, ser. SCC '12. Washington, DC, USA: IEEE Computer Society, 2012, pp. 694–695. [Online]. Available: http://dx.doi.org/10.1109/SCC.2012.65

[SV13] B. Suleiman and S. Venugopal, "Modeling Performance of Elasticity Rules for Cloud-Based Applications," in *Enterprise Distributed Object Computing Conference (EDOC), 2013 17th IEEE International*, Sept 2013, pp. 201–206. [Online]. Available: http://dx.doi.org/10.1109/EDOC.2013.31

[SWHB06] B. Schroeder, A. Wierman, and M. Harchol-Balter, "Open Versus Closed: A Cautionary Tale," in *Proceedings of the 3rd Conference on Networked Systems Design & Implementation - Volume 3*, ser. NSDI'06. Berkeley, CA, USA: USENIX Association, 2006, pp. 18–18. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267680.1267698

[TTP14] C. Tinnefeld, D. Taschik, and H. Plattner, "Quantifying the Elasticity of a Database Management System," in *DBKDA 2014, The Sixth International Conference on Advances in Databases, Knowledge, and Data Applications*, 2014, pp. 125–131. [Online]. Available: http://www.thinkmind.org/index.php?view=article&articleid=dbkda_2014_5_30_50076

[vK14] J. G. von Kistowski, "Modeling Variations in Load Intensity Profiles," Master Thesis, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2014. [Online]. Available: http://sdqweb.ipd.kit.edu/publications/pdfs/Kistowski2014.pdf

[vKHK14a] J. G. von Kistowski, N. R. Herbst, and S. Kounev, "Modeling Variations in Load Intensity over Time," in *Proceedings of the 3rd International Workshop on Large-Scale Testing (LT 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, March 2014. [Online]. Available: http://dx.doi.org/10.1145/2577036.2577037

[vKHK14b] J. G. von Kistowski, N. R. Herbst, and S. Kounev, "LIMBO: A Tool For Modeling Variable Load Intensities (Demo Paper)," in *Proceedings of the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, March 2014. [Online]. Available: http://dx.doi.org/10.1145/2568088.2576092

[vSVdZS98] M. van Steen, S. Van der Zijden, and H. J. Sips, "Software Engineering for Scalable Distributed Applications," in *Computer Software and Applications Conference, 1998. COMPSAC '98. Proceedings.*, 1998, pp. 285–292. [Online]. Available: http://dx.doi.org/10.1109/CMPSAC.1998.716669

[Wei11] J. Weinman, "Time is Money: The Value of "On-Demand"," 2011, (accessed July 9, 2014). [Online]. Available: http://www.joeweinman.com/resources/Joe_Weinman_Time_Is_Money.pdf

[WHGK14] A. Weber, N. R. Herbst, H. Groenda, and S. Kounev, "Towards a Resource Elasticity Benchmark for Cloud Environments," in *Proceedings of the 2nd International Workshop on Hot Topics in Cloud Service Scalability (HotTopiCS 2014), co-located with the 5th ACM/SPEC International Conference on Performance Engineering (ICPE 2014)*. ACM, March 2014. [Online]. Available: http://sdq.ipd.kit.edu/research/publications/#WeHeGrKo2014-HotTopicsWS-ElaBench

List of Figures

List of Tables

Glossary

Cloud Management Server Manages the elastic infrastructure of a cloud. It typically consists of a monitoring system, a reconfiguration management and an elasticity mechanism. The functionality is implemented in within a cloud management software. 6, 121, 123

Cloud Management Software Software deployed on a cloud management server in order to control the elastic infrastructure of a cloud. 6, 53, 61, 71, 110, 121, 123

Cloud System Under Test Cloud system that is analyzed by an elasticity benchmark. It consists of the cloud management server, the load balancer and the scalable infrastructure. 3, 6, 111, 121, 123

CPU Central Processing Unit. 8, 9, 12, 21, 26, 28, 38, 68, 69, 71, 73, 76, 78, 82, 83, 97, 101, 109, 110, 121

CSUT cloud system under test. 3, 6, 25–28, 32, 34, 37, 39, 41, 45, 49, 51, 53, 56–65, 68, 71, 95, 96, 110, 111, 119, 121

Descartes Load Intensity Model Meta-Model allowing the definition of varying load intensities through combination of piece-wise mathematical functions. 53, 121, 123

DLIM Descartes Load Intensity Model. 53, 54, 90, 121, 124

DTW dynamic time warping. 20, 44, 45, 121

Dynamic Time Warping Algorithm for calculating the distance between multi-dimensional series. See [SC07]. 20, 45, 121, 123

Elasticity Mechanism Component of an elastic cloud that triggers the de-/allocation of resources according to an elasticity strategy. 2, 6, 10, 11, 14, 19, 48, 72, 101, 103, 107, 111, 121

Elasticity Strategy Defines the process of de-/allocating resources in order to match a varying demand as close as possible. 6, 11, 14, 16, 39, 44, 101, 107, 111, 121, 123

GPU Graphics Processing Unit. 8, 121

Health Check Defines when a VM instance is considered as *healthy*. Typically, the instance has to answer a given number of subsequent requests successfully to be considered as *healthy*. Load balancers usually forward requests only to *healthy* instances. With an auto-scaling mechanism in place, *unhealthy* instances are often replace by substitute instances. 64, 65, 73, 75, 101, 103, 121

IaaS Infrastructure as a Service. 2, 3, 5, 14, 25, 26, 100, 121

Infrastructure As A Service Cloud providers offer basic resources, such as processing, storage or network to their customers on demand. The customer has control over the operating system but not over underlying resources [MG11]. 2, 25, 121, 123

LIMBO Eclipse-based tooling platform for editing of DLIM instances. 28, 53, 54, 66, 90, 112, 121

Load Balancer Distributes incoming load to available instances. Different scheduling strategies such as round robin, least connections, and least response time can be applied for distributing the load. 6, 32, 34, 63–65, 68, 71–73, 77, 101, 103, 121, 123

Load Profile Describes the variation of load intensity over time. 3, 22, 26, 28, 31, 36, 51, 53, 66, 75, 90, 107, 111, 119, 121

NTP Network Time Protocol. 72, 121

PaaS Platform as a Service. 14, 25, 121

Platform As A Service Cloud providers offer a computing platform that allows the customer to deploy applications created using programming languages, libraries services and tools supported by the provider. The customer has no control over the operating system or over underlying resources [MG11]. 25, 121, 124

Resource Elasticity Degree to which a system is able to adapt to load changes by provisioning and deprovisioning in an autonomic manner, such that in each point in time the available resources match the current demand as closely as possible [HKR13]. 2, 11, 12, 20, 25, 27, 80, 91, 102, 109, 110, 112, 121

SaaS Software as a Service. 25, 121

Service Level Agreement Agreement between provider and customers that specifies which service is provided at what quality. Contains several service level objectives. 8, 121, 124

Service Level Objective Specifies a measurable quality criteria of a service. Typical quality criteria for cloud services are response time, availability or throughput. SLOs are normally specified with the help of probabilities or probability distributions. Example: “95% of all response times are smaller than one second”. 8, 121, 124

SLA service level agreement. 8, 121

SLO service level objective. 8, 10, 12, 13, 20, 32, 34, 48, 51, 55, 58, 92, 96, 102, 103, 107, 121

SNMP Simple Network Management Protocol. 72, 121

Software As A Service Cloud providers offer applications to their customers. The customer has no control over the applications, apart from limited application specific settings, the operating system or over underlying resources [MG11]. 25, 121, 124

TimestampTimer JMeter plugin that allows to send requests according to a timestamp file. 55, 56, 121

VM virtual machine. 3, 5, 6, 8, 9, 12, 20, 26, 38, 57, 63, 64, 68, 71–74, 76, 78, 90, 97, 101, 110, 121, 123