Master Thesis

August 24, 2016

Who Provides the Most Bang for the Buck?

An application-benchmark based performance analysis of two IaaS providers

Christian Davatz

of Samedan, Switzerland (10-719-243)

supervised by

Prof. Dr. Harald C. Gall

Dr. Philipp Leitner

University of Zurich[UZH]

s.e.a.l.

software evolution & architecture lab

Master Thesis

Who Provides the Most Bang for the Buck?

An application-benchmark based performance analysis of two IaaS providers

Christian Davatz

University of Zurich[UZH]

s.e.a.l.
software evolution & architecture lab

Master Thesis

Author: Christian Davatz, davatzc@gmail.com

Project period: February - August, 2016

Software Evolution & Architecture Lab
Department of Informatics, University of Zurich

Acknowledgements

I would like to thank Prof. Dr. Harald Gall for giving me the opportunity to write this Master Thesis at the Software Evolution and Architecture Lab at the University of Zurich and for providing funding to realize this thesis.

My deepest gratitude goes to my advisor, Dr. Philipp Leitner. Thank you for all the valuable feedback during the meetings and also for the additional reviews in the end-phase of this thesis. Furthermore, I want to thank Dr. Christian Inzinger for introducing me to the topic of cloud benchmarking and the insightful inputs during the meetings.

Moreover, I would like to thank Joel Scheuner who helped me a lot with technical issues related to Cloud Workbench.

I am really grateful for the productive work-atmosphere and the excellent workplace in the ICU-Lounge, which would not have been the same without the helpful inputs of David Gallati and Michael Weiss. Also, many thanks go to all other ICU fellows for the table-soccer games and the coffee and lunch breaks we had during the past six months.

Further, I would like to thank Romeo Arpagaus and Rade Kolbas, my team-leaders at Haufe-umantis AG in St. Gallen, which always support any education-related initiative to their best.

Last but not least, I would like to thank my parents, Andreas and Katharina and my brother Mathias for always supporting and encouraging me.

Abstract

Cloud computing offers compared dedicated in-house infrastructure unprecedented advantages in terms of performance, reliability and cost. Not surprisingly, is cloud computing in 2016 still an on-going trend in internet computing. Today, the most important service model is infrastructure-as-a-service and an ever-growing number of commercial vendors edge into the market. While all vendors provide similar functionality, differences in the non-functional properties such as performance, reliability, and cost are significant.

Although all IaaS providers provide performance indicators for these non-functional properties, studies show that the effectively delivered service performance varies. In consequence, is the selection of the most relevant cloud service for a given application not straight forward. Cloud benchmarking (i.e., the process of thoroughly evaluating the performance of these services) is therefore a common contemporary research topic in the cloud domain.

In this work, we propose a generic application-benchmark to support practitioners in collecting performance data across different cloud providers. In a second step, we use the benchmark to collect performance data on the performance of several instance types at Amazon EC2 and Google Compute Engine. Our results show, that compute-specialized instance types deliver a better performance-cost ratio than general purpose instance types and larger instances are less cost efficient than smaller ones. Additionally, while instances from Amazon EC2 provide both, better predictability and stability, instances from Google Compute Engine provide the better performance.

Finally, we outline a performance-cost index, which allows to compare several instance types across several benchmarks.

Zusammenfassung

Cloud-Computing bietet im Vergleich zur traditionellen in-house Bereitstellung von IT Infrastruktur noch nie dagewesene Vorteile in Bezug auf Leistung, Zuverlässigkeit und Kosten. Es überrascht deshalb nicht, dass Cloud Computing auch im Jahr 2016 nach wie vor an Attraktivität gewinnt.

Das heute wichtigste Geschäftsmodell ist IaaS. Dieses beinhaltet die Bereitstellung grundlegender Recheninfrastruktur wie beispielsweise Rechenkapazität, Speicher- und Netzwerkresourcen als Dienstleistung. Wie Cloud Computing, erfreut sich auch IaaS grosser Beliebtheit und die Zahl der IaaS Dienstleister steigt stetig.

Obwohl alle IaaS-Dienstleister funktional äquivalente Produkte anbieten, unterscheiden sich die erbrachten Dienstleistungen in ihren nicht-funktionalen Eigenschaften, wie Leistung, Zuverlässigkeit und Kosten, teils signifikant.

Zwar bieten IaaS-Dienstleister Leistungsindikatoren für diese nicht-funktionalen Eigenschaften an - Studien zeigen allerdings, dass die tatsächlich gelieferte Serviceleistung variiert. Aufgrund dessen ist die Auswahl des relevantesten Cloud-Dienstleisters für eine bestimmte Anwendung nicht trivial.

In dieser Arbeit schlagen wir einen generischen, anwendungsbasierten Benchmark vor, der IaaS Nutzer beim Sammeln der nötigen Daten unterstützt und bei einer Vielzahl von Cloud Anbietern angewendet werden kann. In einem zweiten Schritt nutzen wir den Benchmark um leistungsbezogene Daten von Instanztypen bei Amazon EC2 und Google Compute Engine zu sammeln. Unsere Ergebnisse zeigen, dass rechenspezialisierte Instanztypen ein besseres Leistung-Kosten-Verhältnis aufweisen als Allzweck-Instanztypen. Zudem zeigen wir, dass grössere Instanzgrössen weniger kosteneffizient sind als kleinere. Des Weiteren zeigt sich, dass die Leistung von Instanzen bei Amazon EC2 zwar stabiler und besser vorhersehbar ist, Google Compute Engine aber mehr Leistung für das gleiche Geld bietet.

Um die gesammelten Daten mit anderen Benchmarks vergleichbar zu machen, wird ein neues Normalisierungsverfahren umrissen. Anhand des „Comparable Benchmark Scores“, kann die Leistung verschiedener Instanztypen über verschiedene Benchmarks hinweg verglichen werden, und so ein Leistungs-Kosten Index erstellt werden.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Cloud computing [AFG+10, BYV+09, MG11] is still an on-going trend in internet computing [Gar15, Gar16]. In cloud computing, resources, such as CPU processing time, disk space, and networking capabilities are offered as a service. In consequence, cloud computing offers compared to the traditional computing model that uses dedicated in-house infrastructure, unprecedented advantages in terms of performance, reliability and cost [AFG+09, HSS+10]. Today, the most important service model is infrastructure-as-a-service (IaaS) [Hil09]. In the IaaS model of cloud computing, computing resources such as *"processing, storage, networks, and other fundamental computing resources"* [MG11] are acquired on a pay-per-use basis, typically in form of virtual machines (VMs) [BYV+09].

Since IaaS is receiving a significant hype in industry, a large number of commercial vendors have started to offer IaaS services (e.g., Amazon's EC2[1], Google's Compute Engine[2], Microsoft's Azure[3], or Rackspace's Public Cloud Hosting[4]). While all these vendors provide similar functionality, differences in the non-functional properties such as performance, reliability, and cost are significant.

Although all cloud providers provide performance indicators for their services, Lenk et al. [LML+11] report that these are not sufficient to compare different offerings. Studies listed in [FJV+12,LC16] show, that the effectively delivered service performance varies between providers. Additionally, and what is even more interesting, service performance also varies for services that are provided by the same provider and comply to the same specification [DPC10, LML+11, FJV+12,GLOT13]. Hardware heterogeneity, contention, and other phenomena can result in tremendously differing performance across supposedly equivalent instances [FJV+12].

The selection of the most relevant cloud service for a given application is consequently an elaborate endeavour and hence not straight forward. In order to support practitioners in selecting the most relevant cloud service for their applications and to mitigate the effects of performance variations, such as higher costs due to longer running tasks (e.g., [OG14]) and difficulties in resource planning (e.g., [CLN12]), many researchers have started initiatives to evaluate the performance of these services [LZO+13]. Cloud benchmarking (i.e., the process of thoroughly evaluating the performance of these services) is a common contemporary research topic in the cloud domain [LC16].

[1] https://aws.amazon.com/de/ec2/
[2] https://www.cloud.google.com/products/compute-engine/
[3] https://www.windowsazure.com/en-us/
[4] http://www.rackspace.com/cloud/

1.1 Problem Statement

In general, there exist two distinct approaches to assessing the performance of cloud services, namely the *predictive* and the *empirical* approach [GCMS15]. The predictive approach tries to predict the performance of the actual application either by simulating the application's behaviour in the cloud (e.g., [LZK+11, FFH12]) or by comparing the expected usage profile with data from existing cloud benchmarks (e.g., [Clo15,LYZK10,SDQR10,MH12,SASA+11,IOY+11]). Results obtained from predictive solutions, especially from cloud simulators, can be quite inaccurate. This is due the missing consideration of some cloud characteristics, such as physical resource sharing and multi-tenancy, in current performance prediction models [GCMS15]. In the empirical approach, solutions collect data by running either simple programs (i.e., a microbenchmark) or real-world like applications (i.e., an application benchmark) in the cloud (e.g., [KLIZ12, BLL+14, CMS16]). While results gathered by microbenchmarks are restricted to a specific component of the service and thus are not generalizable for actual applications, the benchmarking of real world applications is practically restricted by the time required and the financial resources available. In consequence, application benchmarking is only feasible for small applications (e.g., [LYZK10, GCMS15]). Representative application benchmarks are typically much easier to deploy and execute than full-blown real-world applications, but testing several instance types of different service providers and with tailored configurations makes application benchmarking still labour intensive.

However, no matter which approach is chosen, both are challenged by the perpetual change originated in the nature of cloud environments, making continuous re-evaluation of the results inevitable [Sch14]. A recent large-scale literature review by Leitner and Cito [LC16] on studies evaluating the performance of different cloud services yields 15 hypothesis covering often documented patterns. In their work, Leitner and Cito [LC16] also conduct experiments and reveal some issues with the current body of knowledge. Consequently, they state the facts that *"All in all, despite the plethora of existing data points, it remains surprisingly difficult to extract meaningful and portable knowledge from existing research."* [LC16]. Further Leitner and Cito [LC16] argue that a thorough survey across a large number of IaaS providers is necessary in order to assess which of the assumptions prevalent in the cloud computing community stood the test of time and remain valid today [LC16].

Hence, we set out to answer the following research question:

> *RQ 1: How can cloud users application-benchmark different IaaS instance types with regards to the performance they provide for hosting an arbitrary cloud application, in a repeatable manner and for instance types of different cloud providers?*

In order to support practitioners in selecting a relevant instance type, we also investigate the following questions based on the data produced in RQ1:

> *RQ 2.1: Are there diseconomies of scale for larger instance sizes?*
>
> *RQ 2.2: Is it economical to choose specialized instances for special tasks?*

With the data gathered in RQ 1, we can also validate the remaining hypothesis Leitner and Cito [LC16] formulated in their work. Hence, further research questions are:

> *RQ 3.1: Is the performance of an instance of a certain size predictable?*
>
> *RQ 3.2: Is the performance of a certain instance stable?*
>
> *RQ 3.3: Are larger instances more stable than smaller ones?*

Additionally, we also research the following, more general question:

> *RQ 4: How can the performance of different instance types and different cloud providers be objectively compared across multiple benchmarks?*

1.2 Contributions

In this work, we lay the foundation for cross-provider and application-aware benchmarking of IaaS cloud services and work towards an comparable-benchmark-score based performance-cost-index.

To this end, we continue prior research on benchmarking IaaS providers based on Infrastructure-as-Code [SLCG14, SCLG15] and propose a generic application-benchmark, which instead of providing a benchmark application, allows cloud users to benchmark their own application in an automated and repeatable manner.

We implement this benchmark in Cloud Workbench[5](CWB) [Sch14] and by deploying the benchmark to Amazon EC2 and to Google Compute Engine, we show how our application-aware benchmark can be used to benchmark an arbitrary cloud service in a fully automated manner. As a reference-application we choose AcmeAir[6]. AcmeAir is a performance benchmark developed and made available by IBM. The AcmeAir web application is a web-service like application exhibiting real business requirements[7] and is therefore considered to be well suited as reference-application.

We benchmark Bursting, General Purpose, and Compute Optimized instance types, and in total 14 instance configurations at Amazon EC2, and 6 at Google Compute Engine (GCE). Our results show, that for practitioners it is advisable to choose small but optimized instance types. In fact, provide compute optimized instance types at Amazon EC2 up to 20% and at GCE up to 30% more performance for the same costs. Additionally, we observe diseconomies of scale at both providers which in combination with the lack of additional performance stability and the lack of additional performance predictability do not make up the higher price level. In general, show all non-bursting instance types a relatively predictable performance, while none of the tested instance sizes can be classified as stable over time.

Further we develop a performance-cost-index which abstracts from individual metrics by reporting on the performance in terms of comparable-benchmark-scores. Additionally, we propose to extend this performance-cost index with performance data from reference-applications of various types and domains. By reporting on the performance of the reference-applications on different cloud providers, we enable an apple to apple comparison. Consequently, the more properties a real application shares with one of the reference-applications in the index, the better the index predicts the performance for the real application. By this means, the index facilitates the comparison of the different cloud services' performance and incurring costs and by doing so supports informed decisions regarding cloud service choice.

[5]https://www.github.com/sealuzh/cloud-workbench
[6]http://www.acmeair.github.io
[7]https://www.github.com/acmeair/acmeair

1.3 Thesis Outline

The remainder of this work is structured as follows: In Chapter 2, the theoretical foundation is laid and important models and concepts are introduced. In Chapter 3, related research is discussed. Chapter 4 treats the development of a generic application benchmark based on Cloud Workbench (CWB), with Apache JMeter as driver and with AcmeAir as first reference-application. The experimental setup is outlined in Chapter 5. The results are presented and discussed in Chapter 6 before in Chapter 7 the performance-cost index is developed. Chapter 8 and 9 round off the work with a discussion of threats to validity and closing remarks, latter including the conclusion and an outlook to future work.

Chapter 2

Background

2.1 Definition of Cloud Computing

The main idea behind cloud computing is not a new one, however, it was only in 2006 when Google's CEO Eric Schmidt coined the term by describing the business model of providing services across the Internet as *cloud computing*. Since then, the term cloud computing has been used to represent many different ideas in a variety of contexts [ZCB10, AFG+09]. Therefore, and to prevent any misconceptions, we adopt the definition of cloud computing provided by The National Institute of Standards and Technology (NIST):

> *Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction. [...] [MG11]*

The essential characteristics, the service models, and the deployment models are in the following subsections further introduced. All information is taken from [MG11].

2.1.1 Essential Characteristics

On-demand self-service. Every customer can acquire and release computing resources on-demand, in an automated manner and without requiring human interaction from the cloud provider.

Broad network access. The services offered by the cloud provider are available over the internet and accessible by arbitrary device platforms.

Resource pooling. The cloud provider offers a pool of computing resources that can be dynamically assigned and reassigned to multiple resource consumers (multi-tenant model) based on the consumer's demand. The customer may be able to specify a location at a higher level of abstraction (datacenter location) but generally has no control or knowledge over the exact location of the provided resources.

Rapid elasticity. Computing resources can be provisioned and released on demand, at any time and in any quantity. Therefore, the resources available for provisioning often appear to the customer to be unlimited.

Measured service. The services provided by the cloud provider are equipped with measuring capabilities and are therefore often charged on a pay-per-use basis.

2.1.2 Service Models

Cloud computing refers to both the hardware located in the datacenters and the software offered as services over the Internet, which is running on that hardware [AFG+10]. Also the NIST definition [MG11] captures this aspect and introduces three service models for cloud computing [MG11]. These models can also be seen as three different levels of abstraction and are therefore also referred to as service levels [FAS+12]. Figure 2.1 illustrates the different levels of abstraction and indicates which entity is in control and hence in charge.

Infrastructure as a Service (IaaS). IaaS refers to on-demand availability of infrastructural resources, such as network, CPU, operating systems and storage, usually in terms of virtual machines (VMs) [BYV+09]. The entity owning the cloud is called IaaS provider. Examples of IaaS providers include Amazon EC2[1], Google Compute Engine[2], Microsoft Azure[3], or Rackspace[4]).

Platform as a Service (PaaS). PaaS refers to providing platform layer resources, including operatingsystem support and software development and runtime frameworks. Examples of PaaS providers include Google App Engine[5], IBM Bluemix[6] and Force.com[7].

Software as a Service (SaaS). SaaS refers to providing on-demand applications over the Internet Examples of SaaS providers include Salesforce.com[8], Prezzi.com[9] or Google Services such as Maps and Gmail.

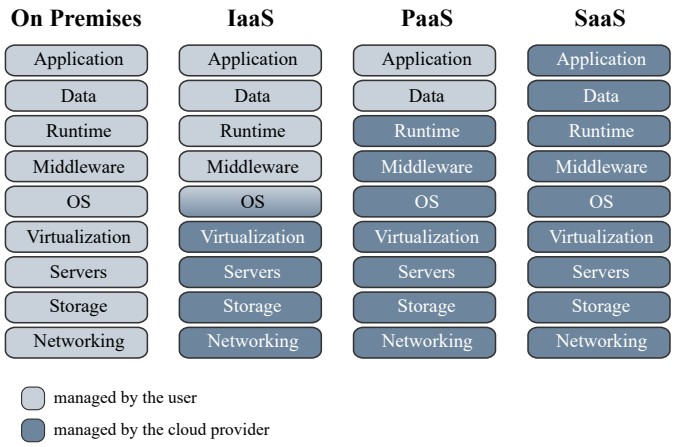


Figure 2.1: Service models and schedule of responsibilities. Taken form [Ger15]

[1]http://aws.amazon.com/de/ec2/
[2]https://cloud.google.com/products/compute-engine/
[3]https://www.windowsazure.com/en-us/
[4]http://www.rackspace.com/cloud/
[5]https://cloud.google.com/appengine
[6]https://console.ng.bluemix.net/
[7]http://force.com/
[8]http://www.salesforce.com
[9]http://taleo.com/

Additionally to the here introduced service models, several specializations and extensions such as Database as a service (DBaaS) or Container as a Service (CaaS) have emerged (e.g., [YBDS08], [DRK14]).

2.1.3 Deployment Models

The NIST [MG11] definition of cloud computing presents four deployment models, namely Private Cloud, Community Cloud, Public Cloud and Hybrid Cloud. While all deployment models exhibit the essential cloud characteristics and service models, they differ in who owns and operates the cloud resources and who is eligible to consume services.

Private Cloud. The private cloud provides services exclusively to a single organization. It is owned by the organization, an external provider, or some combination of them [MG11]. A private cloud provides the highest degree of control over performance, reliability and security. Its similarity to a traditional, proprietary datacenter prevents the realization of typical key benefits such as no up-front capital investment [ZCB10].

Community cloud. The community cloud provides services exclusively to a dedicated community of organizations. It is owned by a single or multiple organizations in the community, a third party, or some combination of them [MG11]. Regarding security, the community cloud could be seen as a trade-off between security and cloud benefits.

Public Cloud. The public cloud provides services to the general public. It is owned by a business, academic, or government organization, or some combination of them [MG11]. While the public cloud offers several key benefits to its consumers, they lack fine-grained control over data, network and security settings, which restricts their effectiveness in many business scenarios [ZCB10].

Hybrid cloud. The hybrid cloud combines two or more of the other deployment models by proprietary or standardized technology that enables application and data portability [MG11] (e.g., to circumvent legal restrictions on data security).

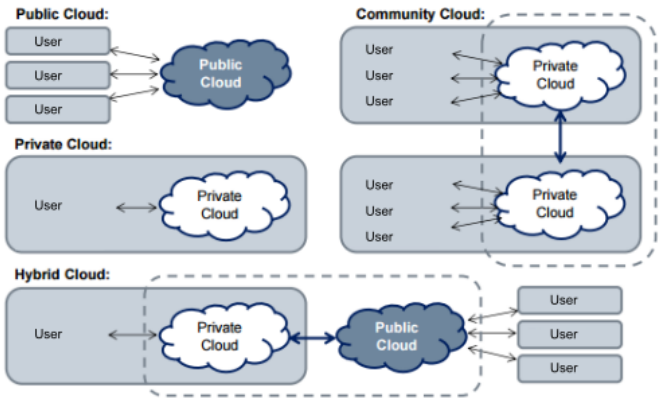


Figure 2.2: Deployment models, adapted from [Lab12, DSP10]

2.2 Resources in IaaS Cloud Computing

To achieve elasticity and the illusion of infinite capacity, cloud providers rely on statistical multiplexing [AFG+09]. That is, physical resources are shared among users. Based on the fact that an average user's demand varies, resources are allocated dynamically to serve single users on an as-needed basis and thus, general resource utilization can be optimized. In order to hide the implementation of how resources are multiplexed and shared, physical resources have to be virtualized [AFG+09]. Armbrust et al. [AFG+09] identify three resource types needed for running a basic application (*computation resources*, *storage resources* and *networking resources*). Cloud providers offer these computing resources as bundles, so called "Instance Types" [Ama16b] and also known as "Machine Types" [Goo16d]. Each instance type offers at least one "instance size", which determines the actual number of CPU cores, the memory size, and the provided storage and networking resources. When requesting a virtual machine from a certain instance type and size, users also have to specify a datacenter within the cloud and a base image. Latter provides the operating system and additional software packages that should initially be installed in the virtual machine when it is launched [LC16]. A concrete virtual machine exhibiting an instance type and instance size, is in the cloud context also called instance.

Amazon EC2 categorizes the available instance types into so called "Instance Families". The instance family describes the general use case such as "General Purpose" or "Computation Optimized". Google follows another naming convention. While Amazon EC2 calls an individual bundle "Instance Type", Google adheres to the notion of a "Machine Type". Howsoever, a concrete virtual machine exhibiting the specification of a "Machine Type" is still called instance.

Both providers offer a wast selection of instance families and types. The ones important to this work are described in further detail in the subsequent sections.

Burstable Performance

Burstable Performance instance types offer a baseline level of CPU performance but are able to burst above the baseline for a short period of time. The duration of the bursting duration is either fixed [Goo16d] or determined by some sort of credit system [Ama16a], where low load periods accumulate credits and high load periods consume credits.

General Purpose

General Purpose instance types are designed to offer a balance of compute, memory, and network resources and are therefore suited for a broad range of applications [Goo16d, Ama16b]. Possible applications are the hosting of small- and mid-sized databases or running backend-servers for SAP or Microsoft SharePoint [Ama16b].

Optimized

In contrast to the general purpose instance types, optimized instance types are designed for a special use-case requiring instances providing enhanced resources of a certain type [Goo16d]. So are compute optimized instance types on EC2 equipped with the highest performing processors and often also featuring better storage connectivity, compared to general purpose instance types, in order to be suited for batch-processing jobs or web-server deployments [Ama16b]. Analogously are storage optimized instance types providing high Random I/O performance, huge SSD drives and high memory, dedicating them to the hosting of NoSQL Databases or cluster file systems [Ama16b].

Custom

If none of the predefined instance types fits the users needs, some cloud providers offer custom instance types (e.g., [Goo16a]). Custom instance types allow the user to specify the number of vCPUs and the amount of memory according to defined rules. Custom instance types are more expensive than predefined instance types and thus only ideal for workloads that require a very unbalanced set of resources [Goo16a].

2.3 Costs for Cloud Resources

There exist several different pricing models for cloud resources. In fact, every cloud provider has its own model (e.g., [Ama16c, Goo16c]). In the following subsections, the pricing models of Amazon EC2 and Google Compute Engine are introduced. Although both providers charge not only for computing resources, but also for storage and networking resources, for this thesis, we will only consider costs for computing resources.

2.3.1 Pricing Models Amazon EC2

Amazon offers three different pricing models for instances of the same type and size. The "On-Demand Instance" pricing model, the "Reserved Instances" pricing model, and the "Spot Instance" pricing model. Additionally to the pricing model, prices also depend on the region in which an instance is acquired.

On-demand instances On-demand instances are billed on an hourly basis with a fixed price and thus represent the "pay-as-you-go" pricing model cloud computing is known for. Each instance size has another price and the utilization of instances is measured in so called "instance-hours". In particular is every instance billed individually by the hour, from the time it is launched until it is terminated or stopped. One instance-hour is the smallest possible billing unit, consequently each partially consumed instance-hour will be billed as a full hour [Ama16c], i.e.,

$$c_i = f_i(max\{1, \lceil t_i \rceil\}) \forall i \in I \tag{2.1}$$

where I is the set of instance sizes of a certain instance type and

$c_i :=$ charges for a certain instance $i \in I$

$t_i :=$ time the instance $i \in I$ was running

$p_i :=$ constant **per-hour** price of instance $i \in I$

$f_i(x) :=$ the linear cost function of instance $i \in I$ for run time x.

While this model allows great flexibility, it does not allow users with recurring needs to qualify for discounts.

Reserved Instances The reserved instance model allows users with recurring needs to reserve an instance on an 'always-on' (24 x 7) basis, or to schedule the use of the instance on a recurring basis (for example every day from 6 p.m. to 8 p.m. Currently it is not possible to reserve instance on single days). The benefits of reserved instances compared to on-demand instances are twofold:

firstly, the user qualifies for a discount based on the duration of the contract and the chosen payment model, and secondly, the user is guaranteed to get the reserved instance on the specified time and for the specified duration. But as a consequence, the user looses its flexibility [Ama16d].

Spot Instances Spot instances allow the user to place a request for idle computing capacity. The price depends on the supply of and demand for spot instances capacity. Every user can place a request indicating the max-price the user is willing to pay. If the current spot-instance price is below the max-price, the request is fulfilled and the instance starts running. It terminates as soon as the spot price exceeds the user's max-price [Ama16d].

In this thesis we focus only on certain instance types and all of these where acquired using the on-demand pricing model. The used instance types and sizes are presented in Section 5.2.

2.4 Pricing Models Google Compute Engine

Unlike Amazon EC2, Google Compute Engine neither offers distinct pricing models nor the possibility to reserve instances. Instead, Google offers a sustained use discount to regular users and allows users to acquire "preemtible" instances [Goo16c]. An overview of the used instance types and related prices relevant to this work is depicted in Table 5.3 in Section 5.7.

General Pricing Google Compute Engine offers two families of instance types: predefined machine types and custom machine types. Predefined instance types have predefined virtualized hardware properties and benefit from a region dependent fixed price, while custom machine types are priced according to region, the number of vCPUs and memory that the virtual machine instance uses [Goo16c]. Apart from the specific pricing, the following billing model applies to all instance types, predefined or custom.

All instance types are charged a minimum of 10 minutes. For example, if an instance for 2 minutes, it be charged for 10 minutes of usage. After 10 minutes, instances are charged in **1 minute** increments, rounded up to the nearest minute. An instance that is running for 11.25 minutes will consequently be charged for 12 minutes of usage [Goo16c], i.e.,

$$c_i = \begin{cases} f_i(t_i) & \text{if } t_i > 10; \\ f_i(10) & \text{otherwise.} \end{cases} \quad \forall i \in I \tag{2.2}$$

where I is the set of instance sizes of a certain instance type and

$c_i =$ charges for a certain instance $i \in I$

$t_i =$ time the instance $i \in I$ was running

$p_i =$ constant **per-minute** price of instance $i \in I$

$f_i(t) = \lceil t \rceil \cdot p_i$, the linear cost function of instance $i \in I$.

Sustained use discount Google Compute Engine offers a sustained use discount to users that utilize an instance for a significant portion of the billing month. The discount increases with usage-duration and users can get up to a 30% net discount for each instance, that runs the entire month. Sustained use discounts are calculated per inferred instance. Consecutive and non-overlapping parts of use-periods of different instances of the same instance size are summed

up and count towards a single "inferred instance". Parallel running instances of the same type contribute only with their non-overlapping periods towards the "inferred instance". For a visualisation of the calculation, please refer to Figure 2.3.

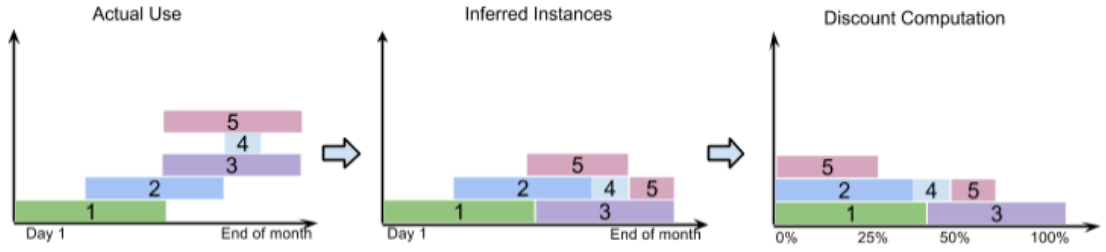


Figure 2.3: Example of GCE's inferred instance calculation for predefined machine types. Taken from [Goo16e]

Preemptible VM Instances Preemptible instances are regular instances, which are provisioned on idle Compute Engine capacity and thus run at much lower price than regular instances. However, they may be terminated (preempted) without notice as soon as the capacity is requested for regular instances. Preemptible instances are idle capacity, consequently their availability varies with the supply and demand for computing resources [Goo16b].

2.5 Benchmarks

In the context of IT, benchmarking usually refers to the process of measuring the performance of a particular computing system, or a subcomponent or feature of it [VV12]. To support the tester, there is a plethora of tools available. As an example, the SPECCpu benchmark[10] can be used to test which CPU can solve the most integer computations in a given time, and the TPC-C benchmark[11] yields which database system for OLTP[12] applications can perform the most transactions per minute [FAS+12]. In practice, benchmarks are used for to gain insights into performance bottlenecks as well as for gathering data required to compare different systems, applications or services to each other [FAS+12]. The system which is benchmarked is often referred to as the system under test (SUT). It consists of the *components of interest* and purely *functional components*. A benchmark can only test the whole SUT, hence it is vital to have complete knowledge of the system in order to derive accurate information. Beside the SUT, a benchmark also includes the driver, which generates the workload and is not part of the SUT. Often benchmarks also define rules on how to setup and run the benchmark, and on how to obtain measurement results [FAS+12]. In IT, there exist two general types of benchmarks which differ in scope and complexity, namely *microbenchmarks* and *macro benchmarks* [Zha01], the latter are also called *application benchmarks*.

[10] http://www.spec.org/cpu2006/
[11] http://www.tpc.org/tpcc/
[12] Online Transaction Processing

2.5.1 Micro Benchmarks

Micro benchmarks aim at investigating the performance of a single component of interest, whether hardware or software, and are therefore designed to put a high artificial workload on this specific component [Gre13]. Micro benchmarks are small in size and sometimes include a short sequence of code (kernel) that solves a small and well-defined problem. As a result, typically the mean of several individual executions is reported. Examples for such basic operations are the time it takes to fetch a data entity from cache/memory, or the time it takes to draw a line on a graphical terminal [Zha01]. While microbenchmarks are well suited to gather and compare information for low-level operations of different systems, information on the performance of an actual application is difficult to derive [Zha01]. Scheuner [Sch14] further categorized microbenchmarks by the specific kind of operation they assess. Computation micro-benchmarks gather fundamental CPU and GPU performance data. I/O micro-benchmarks are used to conduct performance analysis of read and write operations and thus support the selection of the best storage type for a given application. Networking micro-benchmarks help to find bottlenecks on the network layer and scaling micro-benchmarks especially useful for benchmarking applications that acquire resources on demand to cover peak load periods [Sch14].

2.5.2 Application Benchmarks

Macro benchmarks are in current literature often called application benchmarks. They aim at investigating the performance of an actual application as component of interest. This application is either a real life application or a simplified but supposedly representative version of a real life application. Macro benchmarks are usually bundled with a related workload and a set of input data. As macro benchmarks try to reveal the actual performance of an application in a scenario which is representative for a real-life use-case, they place a significant amount of stress on the underlying system. Thus, the accuracy of a result obtained form an application benchmark depends heavily on the representativeness of the workload, the input data and the benchmark application itself [Zha01]. While application benchmarks can be used to reveal an actual application's performance, they fail at investigating performance bottlenecks. Analogously to microbenchmarks, application benchmarks can be categorized according to the type of application they test: Web, Data Intesive, High-performance-computing, etc. [Sch14].

2.5.3 Benchmarks for the Cloud

The advent of the cloud changed the way how benchmarking has to be conducted [BKKL09, FAS+12]. In traditional benchmarking, every component of the SUT was well understood and could be tested in an isolated environment. In cloud computing, the SUT runs on virtualized hardware and usually makes use of other cloud services. That said, if the SUT runs in a cloud environment, it is neither possible to control all components of the SUT nor to prevent resources consumed by the SUT (e.g., storage, network, etc.) from being disturbed by third-party consumers [FAS+12]. Binnig et al. [BKKL09] motivate the need for benchmarks which take the essential cloud characteristics, such as multi-tenancy for instance, into account. In contrast to traditional SUTs, soft- and hardware components should not depend on a static configuration. Moreover, metrics such as the average performance under maximal load are due to the cloud's elasticity obsolete and need to be combined with new metrics to remain useful. New metrics could include the systems ability to adapt to a changed workload in terms of performance and costs or the robustness of the system in the case of a single node failure or even in the case of a complete datacenter outage [BKKL09]. While traditional benchmarks tend to focus on microbenchmarking, Binnig et al. [BKKL09] also introduce the need for application benchmarks that test the whole

application stack and propose benchmarks, that are based on new technologies offering more Web 2.0 like interactions [BKKL09].

2.6 Application Workloads

A typical benchmark consists of a SUT, a driver and a workload representing a real world scenario [FAS+12]. The workload can thereby be described as the consequence of users accessing an application or jobs that need to be processed [FLR+14] and thus characterize the stress a benchmark puts on the SUT. Workloads can either be synthetic or real [LBMAL12] and are defined by the workload pattern and the workload mix they yield. While the workload pattern describes the distribution over time and intensity of the stress, the workload mix describes the composition of the requests used to generate the stress.

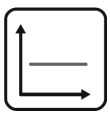
Synthetic Workloads Synthetic workloads model an expected workload and are therefore well suited for controlled experimentation [LBMAL12]. They allow the experimenter to model every possible scenario and to observe the effects of a change in the workload (e.g. higher number of write operations), on the SUT's performance. However, the representativeness of the gathered data depends heavily on the assumptions inherent in the workload definition [LBMAL12].

Real Workloads Instead of defining a supposedly representative workload for a given scenario based on some assumptions, real workloads are recorded during a live run of the SUT and saved as so called trace. The trace can then be replayed by the driver and by doing so the real workload be recreated [LBMAL12]. Real workloads are therefore especially well suited to track the effects of changes to the SUT (or its configuration) on the SUT's performance.

However, real workload data represents a defined set of benchmark parameters and therefore the load cannot be easily adapted (e.g., number of concurrent users, fraction of read and write requests, etc.) to a changed benchmarking scenario [CUWS11].

2.6.1 Workload Patterns

Mao and Humphrey [MH11] present in the context of cloud computing four representative workload patterns: Stable, Growing, Cycle/Bursting and On-and-off [MH11]. More recently, Fehling et al. [FLR+14] extended these. The resulting set of workload patterns is outlined in the following. All icons apart from "on-and-off" take from [FLR+14].



Stable, Static Cloud resources with a more-or-less flat utilization profile over time experience stable [MH11], also called static [FLR+14], workload. A stable workload is characterized by a constant number of requests per time unit [LBMAL12] resulting in an even utilization within certain boundaries [FLR+14].

Growing/Shrinking, Continuously Changing Cloud resources with a continuously increasing or decreasing utilization over time experience growing [MH11], or continuously changing [FLR+14], workload. A steadily increasing workload can for example be caused by a piece of news, a video or other product becoming more and more popular [MH11]. Analogously will the workload on an information platform for a certain product steadily decrease as soon as the product is phased out [FLR+14].

Cyclic/Bursting, Periodic The cyclic/bursting workload [MH11], also called periodic workload [FLR+14], is characterised by stress peaks at reoccurring time intervals. Stress increases over time till a peak is reached and decreases thereafter. A typical example for cloud resources experiencing a cyclic/bursting workload are e-commerce applications [MH11]. During the day the workload is fundamentally different than at night and holiday shopping seasons may cause more traffic than normal." [MH11]

On-and-off The on-and-off workload pattern can be seen as a specialization of the cyclic/bursting pattern. In contrast to the cyclic/bursting workload, the on-and-off workload does increase and decrease in a rather binary than steady way. Mao and Humphrey [MH11] name batch processing and data analysis performed daily or weekly as examples for applications showing this workload pattern [MH11]. These applications have relatively short active periods, after which they remain idle [FLR+14].

Once in a lifetime Another special case of the cyclic/bursting pattern is the once-in-a-life-time pattern. It is characterized a sudden and sharp increase in workload, which, if it is unexpected, in consequence often overloads the servers. An unexpected sharp increase in workload could for example be caused by a breaking-news post on a very popular social media platform or news portal such as Twitter[13] or Slashdot[14]. Latter caused this scenario also to be called "slashdot-effect". An example for a planned scenario causing a once-in-a-lifetime workload is given by the New York Times. Printed documents from their archives had to be digitalized and generated four terabyte of pdf documents [Com07]. Once-in-a-lifetime workloads occur only once in a very long time frame [FLR+14]. In contrast to Fehling et al. [FLR+14], Mao and Humphrey [MH11] did not consider this specialization at all and Lorido et al. [LBMAL12] count it towards the regular cyclic/bursting workload pattern.

Unpredictable While some applications experience a workload exhibiting a clear pattern, other applications' workloads are unpredictable by nature. So might the workload of a pizza-ordering-service of a local pizza-store be influenced by time, weather, prices of products in the supermarket, promotions of competitors etc. and therefore show an arbitrary combination of the other patterns.

2.6.2 Workload Mix

The workload mix describes the ratio of read-only to read-write interactions [CCE+03, Smi00]. In the context of database benchmarking, a read-only interaction would be a pure database lookup such as a SQL select, whereas a read-write interaction could be a create, update or delete operation.

To name an example, TPC-W specifies three different workload mixes [Smi00], each designated to stress a certain aspect of the SUT. The browsing mix contains 95% read-only interactions, the shopping mix 80%, and the ordering mix 50%. While the browsing mix puts a greater load on the webapplication, the ordering mix stresses the database [Smi00].

[13] https://twitter.com
[14] https://slashdot.org/

2.7 Cloud WorkBench

Cloud Workbench (CWB)[15] is a benchmark automation framework levering the idea of "Infrastructure as Code" (e.g., [Hüt12]), where benchmarks can be defined entirely as code and thus be executed with minimal manual interaction [Sch14]. CWB enables the experimenter to define benchmarks that are portable across cloud providers and thus allow the benchmarking of different offerings with one and the same benchmark configuration. The integrated scheduling feature allows the experimenter to conveniently schedule benchmark executions. A benchmark execution is triggered either manually or according to the specified schedule, then acquires computing resources from the cloud provider, provisions the received instances, runs the benchmark, aggregates the benchmark results, and tears-down and releases all resources after the benchmark execution has finished. In order to acquire virtual machines from a wide range of different cloud providers, CWB relies on Vagrant. Vagrant is further introduced in Section 2.7.2. The provisioning of resources is done with Chef, an open-source configuration management tool. Chef is introduced in Section 2.7.2. The Chef community provides readily available configurations for all common tasks and by doing so, speeds up the benchmark development. In consequence, benchmarking cloud services with CWB is both less time-consuming and less error prone than with traditional tools [Sch14].

2.7.1 Benchmark Execution in Detail

A typical benchmark execution is explained in Table 2.1 and depicted in Figure 2.4. The components taking part in the benchmark execution are further explained in Section 2.7.2

In CWB, the Chef platform consists of **Chef Server**, a cookbook repository dedicated to the management and storage of cookbooks. **Chef Client** is used to do the initial examination of the node, to fetch the latest cookbooks from the Chef Server and to provision the node according to the cookbooks.

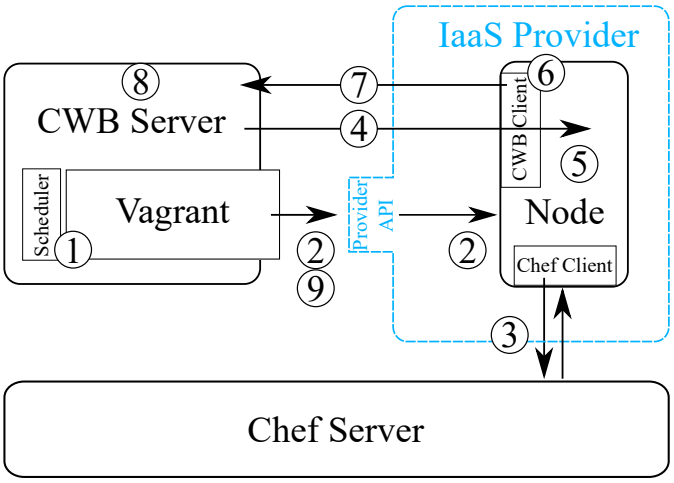


Figure 2.4: CWB architecture. Adapted from [Sch14]

[15] https://github.com/sealuzh/cloud-workbench

Phase	Step	Action
Start:	(1)	A benchmark execution is started either manually or via the integrated scheduling feature.
Prepare:	(2)	Virtual machines are acquired by Vagrant and the IaaS provider initiates the startup of the virtual machines.
	(3)	As soon as the virtual machines are ready, Chef Client is installed and fetches the most recent benchmark cookbook from the Chef Server. Subsequently, all software specified in the cookbook is installed automatically.
Run:	(4)	When Chef Client finished the provisioning, CWB Server starts the benchmark.
	(5)	The benchmark is executed locally on the Node and intermediate results are stored.
Postprocessing:	(6)	Then the CBW Client aggregates the intermediate results.
	(7)	The CWB Client transfers the aggregated results back to the CWB Server.
Finish:	(8)	The results are stored persistently on the CWB Server.
	(9)	Now, the acquired resources are release by Vagrant and the benchmark run is complete.

Table 2.1: Execution steps of a benchmark executed with CWB

2.7.2 Components

CWB consists of several components and subcomponents. The four major components which are essential for the benchmark specification and execution are explained in the following paragraphs.

CWB Server The CWB Server is the main component of the application. It provides the web interface, implements the business logic and stores its data in a relational database. Moreover it provides the scheduling service and orchestrates the different tasks required to setup the SUT and execute the benchmark. At the end of a benchmark execution, the CWB Server collects the benchmark metrics and stores them persistently [Sch14].

Chef The Chef[16] automation platform is a set of tools to automatically provision nodes (i.e., physical or virtual machines) based on a Chef configuration written in Ruby. Chef examines the current state of a node and compares it with the state specified in the Chef configuration. If the states differ, Chef installs missing packages, moves files, alters configurations, and runs scripts in order to converge the actual state to the desired state [Che16a]. A Chef configuration is called "cookbook" and consists of at least one "recipe". The recipe is the essential part of the configuration and tells Chef what actions have to be taken in order to converge a node towards the desired state. Recipes can be made configurable by using attributes. During the examination phase, information about the node is picked up and made available to the recipes through these

[16] https://www.chef.io/

attributes [Che16b]. In CWB, the Chef platform consists of **Chef Server**, a cookbook repository dedicated to the management and storage of cookbooks. **Chef Client** is used to do the initial examination of the node, to fetch the latest cookbooks from the Chef Server and to provision the node according to the cookbooks.

Vagrant Vagrant[17] is an open-source project with the aim to facilitate the management of virtual machine environments. The first version of Vagrant emerged from efforts to automate the creation of local development environments in a reproducible manner and with minimal manual interaction [Sch14] This first version was based on virtual machines provided by VirtualBox[18]. More virtual machine providers were added in subsequent versions and Vagrant soon started to support cloud providers too. Today, all major cloud providers, such as Amazon EC2[19], Google Compute Engine[20], Microsoft Azure[21], and Rackspace[22], are supported [Has16]. CWB integrates Vagrant in order to acquire virtual machines, to trigger the provisioning phase (by delegating to a provisioning framework such as Chef2.7.2 for instance) and to release virtual machines at the end of a benchmark execution. Since CWB uses Vagrant opportunistically and without further customizations, CWB is capable to make full use of all providers and plugins provided by Vagrant and hence supports all common cloud providers supported by Vagrant [Sch14].

CWB Client The CWB Client is a small application which allows the SUT to communicate with the CWB Server. The CWB Client offers functionality to inform the CWB Server about status changes and to submit benchmark metrics as soon as the benchmark run has terminated [Sch14].

2.8 Apache JMeter

Apache JMeter is an open-source load testing tool written in Java and can be used for analyzing and measuring the performance of any kind of webservice. JMeter can be used to evaluate performance on both static files and dynamic resources [Fou16a].

JMeter supports a wide range of features out-of-the-box, including variable parametrization, assertions (response validation), per-thread cookies, caching, configuration variables and the generation of a variety of reports [Hal08, Eri15]. Additionally, JMeter comes with native support for several different protocol and request types, such as http, ftp, mail and ldap [Fou16b]. Consequently, JMeter can be used to generate load for every imaginable scenario [Eri15].

However, despite supporting all basic features of a web browser, JMeter is not a web browser. In contrast to regular web browsers, JMeter neither renders the HTML pages nor executes JavaScript found in HTML pages [Eri15]. JMeter is indeed capable to display the rendered HTML pages, but by default excludes the time required to render the page or the time required to execute the JavaScript from the results [Eri15].

2.8.1 Workload Generation

JMeter generates the workload according to the JMeter Test Plan. The Test Plan is a XML configuration specifying all details about the amount and type of requests JMeter has to send in order to generate the required workload for the SUT.

[17] https://www.vagrantup.com/
[18] https://www.virtualbox.org/
[19] https://aws.amazon.com/ec2/
[20] https://cloud.google.com
[21] https://azure.microsoft.com/
[22] https://www.rackspace.com

The JMeter Test Plan itself is composed of several elements. The most important elements are outlined in the following.

Controllers Controllers are the basic building blocks for the workload and define what type of request with what data has to be sent to the SUT. Controllers can be either **Samplers** or **Logic Controllers**. While Samplers define the type (GET, POST, etc.) and the data (parameters, headers and body) of the request, Logic Controllers are used to customize the logic JMeter uses to decide when to send a request. As an example could a HTTP Request Sampler be wrapped into a Loop Controller and in consequence be run several times. Or a HTTP Request Sampler could be wrapped into a Once-Only Controller and therefore be run only for the first iteration of the Thread Group [Fou16b].

Thread Group A thread group is a conjunction of a group of Logic Controllers and Samplers and can therefore be seen as a testing-manual which describes how a real user would interact with the SUT. The controls for a thread group allow to specify the number of threads (emulated concurrent users) JMeter should launch, in what period JMeter should create the threads (so called ramp-up period) and how many times each thread has to execute the Thread Group before terminating [Fou16b].

Although the flexibility of the Jmeter Testplan allows to imitate every imaginable scenario, it lacks to support the different workload patterns introduced in 2.6.1. The default Thread Group is due to the few configuration options regarding thread-start and -stop times restricted to imitate stable and growing workload patterns.

JMeter Plugin To support all of the workload patterns introduced in chapter 2.6.1, JMeter provides an additional set of Test Plan elements which are available through a plugin. With the Ultimate Threadgroup[23] or the Free Form Arrivals Thread Group[24] an arbitrary workload pattern can be defined [Pok16].

2.8.2 JMeter Results File

Each test run produces a JMeter results file. In the JMeter configuration, the exact format of the file can be specified, often it is *.jtl*, which is a comma-separated text file format. The results file contains all information for each request. An example may be clarifying:

```
1  timeStamp,elapsed,label,responseCode,responseMessage,threadName,dataType,success,bytes,grpThreads,allThreads,Latency
2  1467376053413,3183,Login,200,OK,ip-172-31-15-5_AcmeAir_API 1-1,text,true,406,17,17,3183
3  1467288194502,15345,Query Flight,200,OK,ip-172-31-15-18_AcmeAir_API 1-158,text,true,342,1070,1070,15345
4  1467376059479,238,Query Flight,200,OK,ip-172-31-15-5_AcmeAir_API 1-11,text,true,342,33,33,238
5  1467376059482,247,Query Flight,200,OK,ip-172-31-15-5_AcmeAir_API 1-12,text,true,342,33,33,247
6  1467288194500,15347,View Profile Information,200,OK,ip-172-31-15-18_AcmeAir_API 1-922,text,true,428,1070,1070,15347
7  1467376056517,196,Update Customer,200,OK,ip-172-31-15-5_AcmeAir_API 1-7,text,true,430,18,18,196
8  1467376059491,245,View Profile Information,200,OK,ip-172-31-15-5_AcmeAir_API 1-1,text,true,430,33,33,245
9  1467376057498,26,logout,200,OK,ip-172-31-15-5_AcmeAir_API 1-6,text,true,264,22,22,26
10 ...
```

Listing 2.1: Jmeter JTL File Example

[23] https://jmeter-plugins.org/wiki/UltimateThreadGroup/
[24] https://jmeter-plugins.org/wiki/FreeFormArrivalsThreadGroup/

2.8.3 Test Architecture

Hardware capabilities as well as the Test Plan design will both impact the number of threads a JMeter instance can effectively run [Fou16a]. While for small experiments a single JMeter instance is sufficient, for more extensive experiments a distributed architecture with multiple JMeter instances for generating the load is vital. Hence, JMeter support two testing modes: a simple, and a distributed mode.

Simple Mode The simple mode is based on a single JMeter instance that is used to generate the whole laod required to stress the SUT. During the test run, the results are aggregated in a single file and after the test execution stored locally on the instance. The resulting test topology is depicted in Figure 2.5.

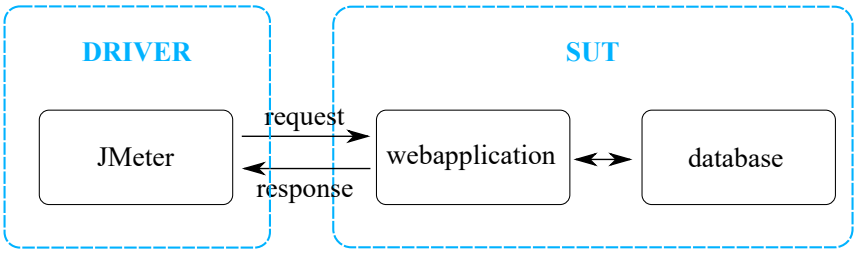


Figure 2.5: JMeter instance topology: simple mode

Distributed Mode The distributed mode is built around a JMeter master instance, which manages and orchestrates several JMeter slave instances. In this mode, the JMeter slave instances generate the load to stress the SUT. The JMeter master sends the JMeter Test Plan (2.8.1) to all slave instances and then starts the test. During the test run, single results are collected decentralized by the Jmeter slave instances and after the test run transmitted to the JMeter master, which aggregates all the results in a single file and stores it locally.

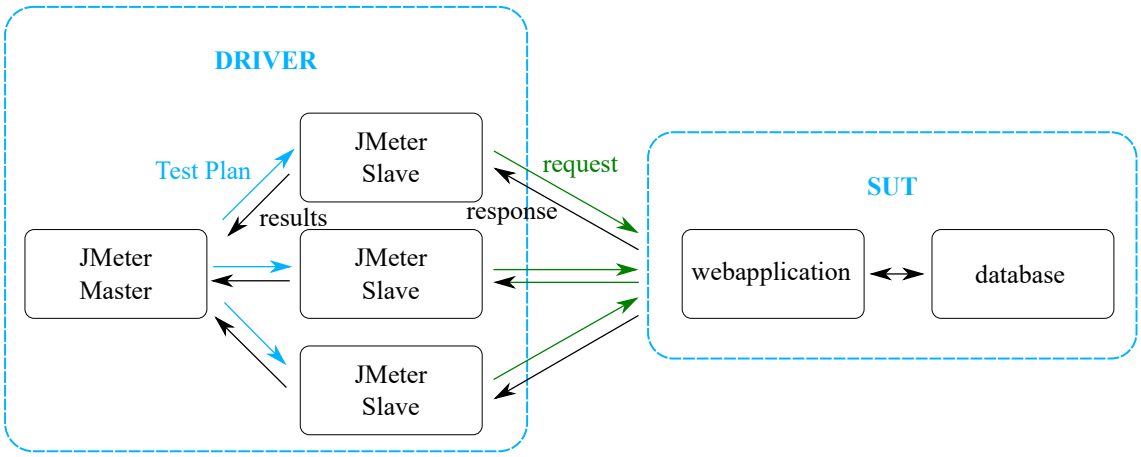


Figure 2.6: JMeter instance topology: distributed mode

Chapter 3

Related Work

There exist already several benchmarking studies researching the performance of cloud services [LZO+13]. Related research highlights the importance of Web 2.0 like workloads and the support for end-to-end support for the whole benchmarking process(e.g., [BKKL09, FAS+12, CUWS11, CCVK13]).

Lenk et al. [LML+11] explicitly proposes to develop for every cloud provider a standardized virtual machine. Not surprisingly, recently developed application benchmarks and benchmark-suits make heavy use of pre-packaged virtual machine images. In order to ease the deployment of the benchmark, they pre-package virtual machine images for SUT, driver including workload, and some kind of management application with all the configurations required to run the benchmark. While this approach allows in theory to support every cloud provider, in practice, users of the benchmark are dependent on the benchmark-developers to provide virtual machine images for new cloud providers edging into the market and to update images regularly. Thus, to our best knowledge, there exists not a single benchmark which is cross-browser native. Along the same lines do only few benchmarks allow to specify a completely custom workload. We identify this as an issue, since permanent change is inherent in the cloud's nature.

We pick up the requirements formulated by [FAS+12] and the generic architecture presented by [ICH+14] as a baseline for the design and the development of our own cross-provider native and generic application-aware benchmark.

We circumvent both, the manual configuration and benchmark set-up as well as the pre-packaging of machine images [LML+11] for individual cloud providers by fully embracing the principles of infrastructure as code [Hüt12]. Consequently, we provide all information required to set up and configure the driver, the workload and the SUT, as code.

Additionally, we implement basic configuration options [FAS+12] allowing the user to specify basic settings or even to use a completely custom SUT and/or workload. By doing so, we allow the cloud user to employ web applications, which make us of bleeding edge technologies, apply Web 2.0 interaction patterns and thus represent the state-of-the-art.

We tackle the need to authentically reproduce user interactions by employing a workload generator, which provides natively the required functionality to address named shortcomings, which are the lack of caching, usage of cookies, parallel downloads and dynamic content loading [CUWS11].

The resulting, generic benchmark is introduced further in Section 4. After this brief introduction, we give in the following sections a comprehensive overview of related work in the field of cloud benchmarking.

3.1 Application Benchmarks

For a long time, the research community has relied on open-source benchmarks such as TPC-W [Smi00,Con05] and RUBiS [ACC+02,Con09] but with the advent of Web 2.0, interactive content and mobile clients, these benchmarks became outdated.

Binning et al. [BKKL09] start a discussion why traditional benchmarks are not sufficient for analyzing cloud services and offer requirements for a new cloud benchmark. They show which of these requirements are satisfied by the popular TPC-W benchmark and conclude with some initial ideas for a new benchmark addressing the shortcomings of the TPC-W benchmark [BKKL09]. While the proposed benchmark addresses the metrics scalability, cost, peak-load handling and fault tolerance, it does not reflect aspects such as hardware heterogeneity, datacenter location or time of execution. In a follow up study to Binning et al. [BKKL09], Kossmann et al. [KKL10] customize the TPC-W benchmark and present the results of benchmarking the end-to-end performance and related cost of running enterprise web applications with OLTP workloads on different cloud services (Amazon, Google, and Microsoft's) [KKL10]. Inspired by the work of Binning et al. [BKKL09], Folkerts et al. [FAS+12] start a discussion on what benchmarking should, can, and cannot be and provide a list of general requirements and challenges for modern cloud benchmarks [FAS+12].

Since TPC-W [Smi00] and RUBiS [ACC+02], a number of new benchmarks have been proposed, such as TPC-E [TPC15] and SPECWeb2009 [Cor09]. However, as there were no open-source or free implementations available to the general public, these benchmarks have only been used by commercial vendors [CUWS11].

In response, Sobel et al. [SSS+08] developed CloudStone, a toolkit consisting of a Web 2.0 application architecture with load injectors relying on a Markov model to model user workloads, a load generator, automation tools, and a methodology and set of parameters for computing the key metric of dollars-per-user-per-month. Besides the key metric, the presented results, which were gathered on Amazon EC2, show the maximum number of users for a specific setup consisting of a VM type and a software configuration [SSS+08].But without capturing or emulating client-side JavaScript or AJAX interactions, an important aspect of common Web 2.0 applications falls short.

To address all these shortcomings, Cecchet et al. [CUW+11] propose BenchLab, an open-source benchmark-suite based on multiple modern Web 2.0 applications. BenchLab provides as SUT several backends, which represent different domains [CUWS11] and are already known from existing benchmarks (RUBiS [Con09], TPC-W [Smi00] and CloudStone [SSS+08]). Moreover, BenchLab [CUW+11,CUWS11] provides an alternative, novel approach to the emulation of complex interactions. In contrast to other benchmarks targeting at web applications, BenchLab makes use of real web-browsers in combination with Selenium[1] to capture and emulate client-side JavaScript or AJAX interactions. To manage and orchestrate all components of the benchmark, and to store the results, BenchLab provides a web based user interface. To ease the configuration and setup, BenchLab images are provided on Amazon [CUW+11]. In their studies, Cecchet et al. show the need to use real web applications as benchmarks and present a tool that authentically reproduces user interactions [CUW+11,CUWS11].

Smart CloudBench [CCVK13,VK14] is a framework, which supports the whole benchmarking process from cloud provider selection to decommissioning of resources which were acquired during the benchmark setup [CCVK13]. The Smart CloudBench application can be used to deploy a Java based implementation of the TPC-W [Smi00] benchmark to all cloud providers supported by Apache jClouds[2]. From the data gathered with the benchmark, Chhetri et al. infer the performance of the tested instance types (i.e. instance sizes) for different scenarios. The whole Smart

[1] http://www.seleniumhq.org/
[2] https://jclouds.apache.org/

CloudBench application stack is available as pre-packaged Amazon EC2 machine images[3] (AMI) and allows the customization of the TPC-W workload [VK14, CCVK13].

Dejun et al. [DPC10] study performance stability and performance homogeneity of VMs provided on Amazon EC2. While performance stability behaves as expected, performance homogeneity emerges as an issue. Dejun et al. observe that the performance of VMs of the same type show very heterogeneous performance profiles, up to a ratio 4 in response time from each other. While this is a issue in terms of performance predictability, they believe that exploiting performance variability could result in an improvement of the overall resource usage. In order to simulate CPU-intensive (processing) and I/O intensive workload patterns, they develop three custom micro-service applications, but withhold details about their concrete implementation [DPC10].

3.2 Micro Benchmarks

Schad et al. [SDQR10] focus on the issue of performance unpredictability in the cloud and exhaustively evaluate the performance of Amazon EC2 instances. To this end, they test single and multiple instances in different datacenters by executing a set of microbenchmarks, which mostly are part of the Unix Benchmark Utility[4]. Their analysis clearly shows that both small and large instances suffer from a large variance in performance. In order to reduce the impact of this issues, they propose to report the used underlying system type (CPU Model, etc.) together with the results [SDQR10].

Lenk et al. [LML+11] investigate if the performance indicators presented by IaaS providers are sufficient to compare the actual VM's performance. Additionally they research the question if standard benchmarks can be used to make different IaaS Cloud offerings more comparable in order to assist the user in his decision which Cloud offering to use. They conclude that performance indicators presented by IaaS providers are not sufficient and formulate the need for a benchmark suite containing all tools required for a cost- and time- efficient performance evaluation. Further, they propose standardized cloud-performance-measurement VMs to compare performance between providers. In their own experiments, they made heavy use of the Phoronix[5] microbenchmark they used three different cloud providers, namely Amazon EC2[6], Flexiscale[7] and Rackspace[8] [LML+11].

Salah et al. [SASA+11] performed numerous microbenchmarks to empirically evaluate and especially compare the performance of Amazon EC2, ElasticHosts and BlueLock [SASA+11].

Leitner and Scheuner [LS15] offer some insights into the bursting and non-bursting instance behaviour of Amazon EC2 bursting instance types [LS15].

3.3 Other Benchmarks

Online Transaction Processing (OLTP) Cooper et al. [CST+10] develop a benchmark framework called Yahoo! Cloud Serving Benchmark (YCSB) for the comparison of systems developed for cloud data serving and report performance results for four distributed database systems [CST+10]. Based on the YCSB-framework, Rabl et al. [RGVS+12] benchmark six distributed

[3] http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/AMIs.html
[4] http://phystech.com/download/ubench.html
[5] http://www.phoronix-test-suite.com/
[6] https://aws.amazon.com
[7] http://www.flexiscale.com/
[8] https://www.rackspace.com

database systems and Kuhlenkamp et al. [KKR14] partially reproduce the experiments from Rabl et al. [RGVS+12] within Amazon EC2.

Difallah and Pavlo [DPCCM13] develop an extensible testbed for benchmarking relational databases. Concretely, they address the need for a comprehensive testbed that supports both a large number of database systems and a wide range of benchmarks that capture the essence of an important set of applications and develop a benchmark suite for OLTP benchmarks called OLTP-Bench [DPCCM13].

VM Startup Time Mao and Humphrey [MH12] conduct a systematic study on the VM startup time across three cloud providers, namely in Amazon EC2, Windows Azure and Rackspace.

Latency Sensitive Applications Barker and Shenoy [BS10] present in their study two different use cases for latency-sensitive cloud applications in the cloud and conduct microbenchmarking to empirically evaluate the efficacy of Amazon EC2 for running latency-sensitive multimedia applications. Their experiments revealed that the CPU and disk jitter, and the throughput can indeed fluctuate due to background load from other virtual machines [BS10].

High Performance Computing (HPC) Furthermore, a number of researchers from the field HPC started evaluating Amazon EC2 for scientific computation tasks (e.g., [Wal08,OIY+08,OIY+09, NB09, XAL10, JRM+10, ETP+13]). While some of these (e.g., [NB09, OIY+08, ETP+13]) use one or more standard benchmarks such as LINPACK[9], HPCC[10], Bonnie[11] or NAS Parallel Benchmarks[12], others use representative applications. For instance Jackson et al [JRM+10] deploy a whole set of applications representing a diverse range of numerical methods and data-structure representations in the fields of climate, material science, fusion, accelerator modeling, astrophysics, and quantum chromodynamics [JRM+10]. In contrast to these studies, Iosup et al. [IOY+11] perform extensive measurements to compare several clouds (Amazon EC2, GoGrid, ElasticHosts, and Mosso) and compare clouds with other HPC-environments based on real long-term scientific-computing traces [IOY+11].

Cloud Simulators Li et al. [LZK+11] propose CloudProphet, a performance prediction tool which employs a trace-and-replay approach to provide accurate application-specific prediction results. In the tracing step, detailed workload information is tracked. During the replaying step, an CloudProphet agent running in the cloud environment under test replays the workload trace and hence emulates the workload. By measuring the performance of the SUT, CloudProphet tries to estimate the real application performance after the cloud deployment [LZK+11].

In order to evaluate the performance of provisioning policie models, application workload models, and resources performance models in a repeatable manner and under varying system and user configurations, Calheiros et al. [CRB+11] developed CloudSim, an extensible simulation toolkit that enables modelling and simulation of cloud applications and their deployment options in a cloud environment [CRB+11]. CloudSim supports for system and behavior modelling of datacenters, VMs and resource provisioning policies. Based on CloudSim, Fattikau et al. [FFH12] developed CDOSim. CDOSim is a simulation tool that allows to simulate cost and performance properties of specified cloud deployment options.

[9] https://www.top500.org/project/linpack/
[10] http://icl.cs.utk.edu/hpcc/
[11] http://www.textuality.com/bonnie/
[12] http://www.nas.nasa.gov/publications/npb.html

3.4 Guidelines and Benchmark Design

There are a number of publications that try to provide guidelines on the subject of benchmark design and implementation [BKKL09, Hup09, Kou06, SKBB09, Sac11, VMSK12]. For our work, we adhere to the guidelines collected by Folkerts et al. [FAS+12]. Folkerts et al. [FAS+12] provide a set of cloud benchmark guidelines which are mostly based on the work of Huppler [Hup09] and Binnig et. al [BKKL09]. The guidelines by Folkert et al. [FAS+12] are formulated as requirements to a new cloud benchmark and cover general requirements, implementation requirements, and workload requirements [FAS+12].

Iosup et al. provide in their work a generic approach to IaaS and PaaS cloud benchmarking, namely a generic benchmark architecture [ICH+14]. While the architecture shows several participating entities in an abstracted way, it leaves concrete implementations to the user.

3.5 Costs and Cost-optimization

Brebner and Liu [BL10] define 3 representative scenarios for hosting cloud applications and calculate the accruing costs. As only paper, they also consider the different limitations in terms of quotas, which restrict infinite scaling and sudden scale-out scenarios [BL10].

Farley [FJV+12] shed in their study light on placement gaming. To this end, they simulate resource scarcity and develop customer-controlled placing strategies for selecting instances in order to exploit performance heterogeneity [FJV+12].

O'Loughlin and Gillam [OG14] show opportunities for a Cloud Service Broker in relation to pricing. A Cloud Service Broker could take advantage of performance variations and solve related and budgeting issues. Firstly, does the time required to complete a task vary with the performance, secondly is the number of instances that is required to complete a job in a certain amount of time dependent on the performance of the individual instances, and thirdly are some computational requirements better addressed by different instance types, thus analyzing the specifics of each instance type implies further costs [OG14].

3.6 Benchmark Automation

In this work, we propose a multi-provider benchmark. To empirically evaluate our efforts, we require means to easily define and execute benchmarks over different cloud providers and in an automated manner. Previous work has proposed multiple approaches to achieve this, including Expertus [JSM+12], Cloud-Bench [SLSR+08] CloudCrawler [CMS13], Smart Cloud-Bench [CCVK13, VK14],Cloud-Gauge [ERR10], C-Meter [LG] and BenchLab [CUW+11, CUWS11]. Although either of these systems could have been used instead as well, we decided to make use of our own framework: Cloud Workbench [Sch14, SCLG15].

Chapter 4

Generic Application Benchmark

Traditional cloud benchmarks are often built around a single application, such as an e-commerce web application as in the case of TPC-W [Smi00] or a bidding system as in the case of RUBiS [Con09]. Benchmarking frameworks often re-use these benchmarks and provide support for a predefined set of these. Consequently, cloud users can choose which benchmark application should be used as the SUT and the workload is adapted accordingly. Examples of such frameworks are BenchLab [CUW+11, FAK+12] and Smart CloudBench [CCVK13].

Summing up, traditional benchmarking frameworks embrace the idea of a benchmark as a defined set consisting of a fixed SUT and a mainly fixed workload. These benchmarks test a specific application with a predefined workload. Although some benchmarks provide the possibility to configure the workload, it is neither possible to use a custom application as SUT nor to define a custom workload to stress the system.

In our approach to application benchmarking, we treat the SUT as a black-box. This allows the cloud user to use any application as SUT, as long as a related workload is provided too. Our approach offers several advantages. Firstly, our approach allows to re-use any traditional benchmark and related workload. Secondly, it allows the customization of traditional benchmark applications to better represent a certain scenario or use-case while using the original workload, and most importantly, our benchmark even allows to use a real-world application as SUT and a related real-life web-trace as workload. Hence, *generic* application benchmark.

4.1 Automation and Portability

Folkerts et al. [FAS+12] name among other things "portability" and "repeatability" as key requirements for a new cloud benchmark [FAS+12] and some of the traditional frameworks support the fully automated and repeated execution of defined benchmarks already [CCVK13, CUWS11, Sch14]. To cover these requirements, we make use of CWB, which is purpose-designed to support the acquisition, provisioning, execution and decommissioning of benchmarks and all related cloud resources. CWB is introduced in more detail in Section 2.7. CWB heavily relies on the notion of Infrastructure-as-Code, that is, all configurations and setup instructions are defined as code. This is a fundamentally different approach compared to the more often applied packaging of VM images and virtual appliances[1]. Although this seems to be an overhead, the induced benefits clearly outweigh the additional up-front effort required to define all configurations in code. Firstly, it enables the benchmark to be repeated without manual interaction and more importantly, it allows the benchmark and all its components to be ported to other cloud providers with minimal effort. In order to port the benchmark to another cloud provider, only the provider

[1] Virtual appliance := *"A service delivered as a complete software stack installed on one or more virtual machines. [...]"* [Dis10]

related parts of the CWB benchmark definition have to be updated. As an example, Listings 4.1 and 4.2 showcase the changes required to migrate the datastorage-part of the CWB benchmark definition from Amazon EC2 to GCE.

```
1 mongodb.vm.provider :aws do |aws, override|
2   aws.region = 'eu-central-1'
3   aws.availability_zone = 'eu-central-1a'
4   aws.ami = 'ami-e05ab38f'
5   aws.instance_type = 'c4.large'
6   aws.security_groups = ['cwb-web']
7   aws.private_ip_address = "172.31.2.1"
8   aws.tags = {'Name' => 'mongodb'}
9 end
```

Listing 4.1: Benchmark def. Amazon EC2

```
1 mongodb.vm.provider :google do |google, override|
2   google.zone = "europe-west1-b"
3
4   google.image = 'debian-8-jessie-java'
5   google.machine_type = 'n1-highcpu-4'
6   google.scopes = ["cloud-platform"]
7   google.name = "mongodb"
8 end
```

Listing 4.2: Benchmark def. GCE

4.2 System under Test

Our generic application benchmark requires all potential SUTs to exist in form of a Chef cookbook. That is, all software required for the application to be run and all related configurations have to be defined in code. Since Chef is a well established automation solution, a lot of software, ranging from small utilities to full-blown applications, is already available as Chef cookbook on Chef Supermarket[2].

As a first application for the benchmark, we implemented AcmeAir as Chef cookbooks. Acmeair is introduced in the following sections. The developed cookbooks are briefly introduced in Section 4.2.2.

4.2.1 AcmeAir Performance Sample/Benchmark

The Acme Air Performance Sample/Benchmark was developed by IBM and uses as benchmark application an implementation of a fictitious airline called "Acme Air". The application was designed to fulfil key requirements of the airline business, such as the ability to scale to billions of web API calls per day, the need to deploy the application to public clouds, and the need to support multiple channels for user interaction and thus represents a compelling industry examples of a system of engagement [Moo11, Spy13].

In-air connectivity, paperless boarding, and mobile applications that alert customers of changing flight plans on the one hand, and always-on mobile connection with customers, employees, and partners, cloud hosted services, internet of things, and big data analytics on the other hand, profile the airline industry as a relevant case [Spy13].

The transformation of the industry from clients-facing applications to applications that have to offer more and more mobile and business to business services, is also what motivated the AcmeAir performance benchmark. The main aim of AcmeAir is to showcase an application, which is able to process over a billion web API requests per day and thus would belong to the

[2] https://supermarket.chef.io/

programmable web's "Billionaire's Club"[3]. The Acme Air Performance Sample/Benchmark is open source and available on Github[4].

AcmeAir is composed of two systems, a web application and a database. In the subsequent sections, these are further described.

Web application

The AcmeAir web application is composed of three distinct components: the user interface, a restful API and a data service integrating the database. Hence, AcmeAir can be qualified as a three-tier online-transaction processing (OLTP) application [Fow03].

Presentation Layer To fulfil the key requirement of supporting multiple channels for use interaction, AcmeAir provides a GUI dedicated to classic desktop browsers as well as a mobile app. The desktop browser GUI is based on the current web standards (HTML5, CSS3, JavaScript) and makes use of the Dojo JavaScript framework[5].

The mobile app is designed as hybrid app to provide a consistent design across Android and iOS devices and also makes use of technologies such as Apache Cordova[6] to get access to mobile unique features (such as location and camera etc.), allowing the mobile application to fully take advantage of device specifics.

Domain Layer AcmeAir has explicitly been designed to support multi-channel and therefore offers the same core services to desktop, mobile, and business partner services. This allows AcmeAir to provide a consistent user experience regardless of the device and business partner. This is realized by a well defined REST API which is based on IBM's Worklight[7] server technology [Spy13].

Data Source Layer The Data Service implements basic functionality for storing and retrieving data form the data source. It provides the functionality as a service and hides the implementation details for the data management by doing so. Moreover it does not implement any business logic. For every data source that should be supported, a dedicated data service has to be implemented.

Database

The data source is the actual database and is managed by the data service. It stores bookings, customers, sessions, flights, flight segments, and airports. While bookings and sessions are generated during the test run, 394 flight segments which connect 31 airports are predefined.

Implementation Alternatives

The actual AcmeAir application can be implemented in several ways. Since the web application is written in Java, it is not restricted to a specific web server technology. The data service is decoupled from the web application and registers itself with the Java Naming and Directory Service (JNDI[8]). The actual connection is established at runtime over a JNDI lookup. At this point, the following implementations are available in the official AcmeAir Github repository[9]:

[3] http://www.programmableweb.com/news/which-apis-are-handling-billions-requests-day/2012/05/23
[4] https://github.com/acmeair/acmeair
[5] https://dojotoolkit.org/
[6] https://cordova.apache.org/
[7] http://www-03.ibm.com/software/products/en/ibm-mobilefirst-foundation
[8] https://docs.oracle.com/javase/tutorial/jndi/ops/index.html
[9] https://github.com/acmeair/acmeair

Webserver:

- IBM WebSphere Liberty[10]

Database:

- MongoDB 2.6.4[11]
- WebSphere Extreme Scale 8.6.0.8[12]

4.2.2 AcmeAir Chef Cookbooks

To implement an AcmeAir Chef Cookbook, we chose the original version of AcmeAir, which is described in section 4.2.1. AcmeAir was downloaded from Github[13] and built from source as described in the AcmeAir Github Wiki. To this end, we had to setup a dedicated virtual machine providing all required dependencies such as Java 7 and several IBM dependencies such as WebSphere Liberty 8.5.5.6 and the WebSphere Extreme Scale libraries.

In the following, we briefly describe the *acmeair-morphia-wlp-distributed* and the *acmeair_mongodb* Chef cookbooks, which we used later for the data collection. These cookbooks are required to set up a distributed SUT, which consist of a web application instance running WebSphere Liberty and a database instance running MongoDB. Prior to these cookbooks, to start out with we also had developed a cookbook implementing WebSphere Liberty and MongoDB on a single instance and another cookbook implementing WebSphere Liberty and WebSphere Extreme Scale also on a single instance.

Web Application Cookbook

In order to setup the AcmeAir sample application in a fully automated manner, we developed *acmeair-morphia-wlp-distributed*. The cookbook was tested with Ubuntu 14.04 and Debian 8.3 (Jessie) operating systems, although for the experiments only Debian 8.3 (Jessie) was used. Moreover we implemented also several configuration options which can be seen in the attributes file *acmeair-morphia-wlp-distributed/attributes/default.rb*

The *acmeair-morphia-wlp-distributed* cookbook installs first all required software to fetch (apt git package) and unzip (apt unzip package) the AcmeAir build files from Github. Then, IBM WebSphere Liberty is installed by using the Chef *wlp* cookbook available on Chef Supermarket. In order to be able to connect to the MongoDB, the mongo-to-java driver is installed. Additionally, to make the newly installed driver available to Websphere Liberty, the MongoDB feature has to be installed. After that, the webapplication is deployed, all configuration-file templates are extended with the actual configuration information provided by the benchmark definition and then applied to Websphere Liberty. After a possible heap space tuning, the Websphere Liberty server is started eventually.

Database Cookbook

To set up and configure the database on a dedicated instance, we implemented an independent cookbook for it, namely *acmeair_mongodb*. The cookbook makes heavy us of the already existing mongodb cookbook available on Chef Supermarket.

[10] https://developer.ibm.com/wasdev/websphere-liberty/
[11] https://www.mongodb.com/download-center#enterprise
[12] http://www-03.ibm.com/software/products/de/websphere-extreme-scale
[13] https://github.com/acmeair/acmeair/commit/f16122729873ef0449ea276dfb2d2a1d45bebb40

To shorten the provisioning phase and to ease the deployment of the SUT in general, we decided to make a database dump and to pre-load that data instead of generating it during the set-up phase. Hence, the database is preloaded with 1 million customers, 47280 flights, 394 flight segments and 31 airports from the dump. Note: the flights loaded from the dump are only available between May and October 2016.

The data in the dump generates a database of size 668MB, from which 520MB are data. This fits entirely in the main memory of the database server. Moreover, the database contains 6 collections, with 1047718 objects in total. The images stored in the Web server file system use 792KB of disk space.

Additionally to the database, also a database user with privileges on the AcmeAir database is loaded into the database.

An detailed listing of the MongoDB statistics is in the appendix: 9.3. All available configuration options can be seen in the attributes file *acmeair_mongodb/attributes/default.rb*.

4.3 Driver

Cecchet et al. [CUWS11] outline the importance of realistic workload generation as an important part of a cloud benchmark [CUWS11] and report that existing web client emulators do not authentically generate requests and therefore may not place a realistic load on the server [CUW+11]. Most existing client emulators do neither download embedded resources (e.g., images, CSS, JS, etc.) nor trigger AJAX interactions as real web browsers would. Additionally, to optimize page loading speeds, real web browsers also apply strategies to download embedded resources in parallel, while modern web applications send out different versions of the same resource based on the browser type (e.g., mobile, tablet or desktop). As a result, the load experienced by the SUT is not authentic [CUWS11, Sac12].

Instead of implementing a custom client emulator (e.g., [CUWS11, CCE+03, Cor16]), we propose JMeter[14], which is briefly introduced in Section 2.8. We chose JMeter over Faban[15] (e.g., used in [SSS+08, CMS13, CMS16]) due to the abilities to (1) support caching, (2) support cookies, (3) define custom headers (user agent specification), (4) retrieve embedded resources, (5) to use a thread/connection pool to simulate parallel fetching [Fou16a] and (6) create workloads based on real web application traces [Hal08, Eri15].

With JMeter as driver, our generic application benchmark supports a broad range of SUTs. Namely all systems supporting common protocols such as http, ftp, soap/xml and ldap [Hal08, Eri15].

4.3.1 JMeter Cookbooks

In order to be use with CWB, also JMeter had to be implemented as Chef cookbook. Since Folkerts et al. [FAS+12] also name configuration as key requirement to new cloud benchmarks, we provide an extended set of configuration options. To this end, we split up the responsibilities of installing JMeter and setting up the JMeter Test Plan between different cookbooks. We implemented a general cookbook *cwb-jmeter*, which is used to set up Jmeter. This cookbook downloads and installs JMeter and files required. Moreover, it modifies file permission in the file system. Further we identified and implemented the most common needs for configuration (e.g., installation directory path, user for permissions, etc.). All configuration options available can be seen in the attributes file: *cwb-jmeter/attributes/default.rb*. The JMeter Test Plan and all Files related to the test plan itself

[14] http://jmeter.apache.org/
[15] http://faban.org/

are managed by the workload cookbook, which in most cases will be provided by the user. The specifics of the workload cookbook are outlined in Section 4.4.4

4.4 Workload

In the era of Web 2.0, modern Web applications make extensive use of JavaScript, CSS and AJAX to enable rich interactivity (e.g., [BKKL09, CUW+11]). An authentic workload should therefore include all interactions a real user would have with the SUT [CUWS11]. Depending on the SUT's domain, the workload is different. Facing this fact, a modern application benchmark neither can nor should capture the idiosyncrasies of every domain. Thus, instead of providing predefined workloads for every domain, we provide convenient ways to define authentic workloads.

In general, there exist two types of workloads, namely real workloads and synthetic workloads. Both are introduced in Section 2.6. In JMeter, both workload types are defined as a JMeter Test Plan. JMeter Test Plans are briefly introduced in Section 2.8.1. To use a specific workload with our generic benchmark, it has also to be defined in code, namely as Chef cookbook.

For this work, we use the workload which is delivered with the AcmeAir performance benchmark. The AcmeAir workload and its specifics are introduced in Section 4.4.3

In the next sections, we look closer at how real and synthetic workloads can be defined in a JMeter Test Plan.

4.4.1 Definition of Real Workloads

Among others, there exist the following three ways to derive a workload definition from a real workload:

JMeter Script Recorder In order to define real workloads, JMeter provides a built-in test script recorder, also referred to as a proxy server [Eri15], which can be used for recording a test plan while the user browses the SUT [Hal08]. Once configured, the test script recorder tracks all interaction between the browser and the SUT, creates test sample objects for them and finally generates a JMeter Test Plan [Eri15].

Modern Web Browsers As an alternative to the built-in test script recorder, some web browsers[16] and browser plugins[17] can be used to record and save web traces as HAR files. Flood.io[18] provides an online conversion tool[19] to convert these HAR files to JMeter Test Plans. For Chrome users, there exists a plugin[20], which can be used to generate a JMeter Test Plan from a browser trace.

Access Logs In contrast to the just mentioned approaches which generate a test plan on a client system, there also exists the possibility to generate a trace from log files, such as the Apache Access Log[21]. Workloads generated based on log files approximate the real workloads best, conversely they are the most complex.

[16] Chrome 52, Firefox 48, Internet Explorer 9
[17] Fiddler, Firebug, Paw
[18] https://blog.flood.io/convert-har-files-to-jmeter-test-plans/
[19] https://flood.io/har2jmx
[20] https://guide.blazemeter.com/hc/en-us/articles/206732579-Chrome-Extension
[21] For more sources please refer to Section 9.6 in the Appendix.

Folkert et al. [FAS+12] mention in their work the requirement "scalability". The Jmeter Test Script recorder as well as modern web browsers generate traces, which are recorded on a per-user basis. Therefore, workloads generated with these approaches scale linearly with the number of emulated users. If the definition is based on a log file and the log file is taken from a production system, the resulting workload is often too complex to be easily scalable to a certain amount of users.

4.4.2 Definition of Synthetic Workloads

To create test plans manually, Jmeter is equipped with a GUI and also commandline tools. For a detailed explanation of how to define synthetic workloads, we refer to Halili [Hal08]. However, while this is especially handy to capture rather trivial scenarios or to setup test workloads, this is mostly impractical for defining non-trivial testing scenarios. [Eri15].

4.4.3 AcmeAir Benchmark Workload

Together with the AcmeAir performance benchmark, Spyker [Spy13] developed a related synthetic workload. This workload emulates the browsing session of a typical user and tests all implemented features. Since the workload is synthetic and is defined on a per-user basis, the load can be scaled with the number of emulated users. Hence, the key requirements "Representativeness" and "Scalability" proposed by Folkerts et al. [FAS+12] are fulfilled. As "Metric", the third key requirement according to Folkerts et al. [FAS+12], Spyker propose "requests per day", in order to be able to assess if AcmeAir really belongs to programmable web's "Billionaire's Club"[22].

HTTP Requests and Workload Mix

Spyker [Spy13] and his fellows used the following workload to benchmark their setup of AcmeAir. The listing takes the format: requests per iteration $\times$ request (info)[*HTTP request type, approx. percentage of all successful requests sent to the webapplication*].

$1 \times$ Login $[POST, 12\%]$
¼$\times$ Update Customer:

- View Profile Information $[GET, 3\%]$
- Update Customer $[POST, 3\%]$

$5 \times$ Query Flight $[GET, 50\%]$
$1 \times$ Book flight (only if last query returned a valid result) $[POST, 7\%]$
$1 \times$ List all Bookings $[GET, 8\%]$
¼$\times$ Cancel Bookings (in 25% of the cases cancel all bookings apart from 2) $[POST, 10\%]$
$1 \times$ Logout $[GET, 7\%]$

Total GET $\simeq 68\%$
Total POST $\simeq 32\%$
¼= triggered every 4^{th} iteration

This workload tests all implemented features of the web application and puts the stress clearly on the web application. In total, all GET requests count for 68% of all requests sent to the web application.

[22] http://www.programmableweb.com/news/which-apis-are-handling-billions-requests-day/2012/05/23

4.4.4 Workload Cookbooks

In order to allow a separation of concerns, we have split the JMeter related provisioning steps into two cookbooks, namely, the *cwb-jmeter* which manages the set up of JMeter, and user provided workload cookbooks, which have only a single responsibility, namely to provide the JMeter Test Plan. To further ease the development of workload cookbooks, we provide several workload cookbooks showcasing different use-cases, for exmaple *jm-acmeair-default*, *jm-acmeair-api* or *jm-acmeair-default-double-peak* to just mention a few. The *jm-acmeair-default* represents the exact workload as provided by Spyker [Spy13]. *jm-acmeair-api* is an improved version of the original AcmeAir workload (*jm-acmeair-default*) and *jm-acmeair-default-double-peak* is an extended version, which makes use of JMeter Plugins to generate a cyclic/bursting workload pattern with two intensive peaks.

More complex JMeter Test Plans often depend on additional files. In our case, these files are the airport-definitions in the form of CSV files and some request pre- and post-processors. To be able to reuse these files among several workloads, we implemented their installation in separate cookbooks, namely the cookbooks with the *-assets* postfix (e.g., *jm-acmeair-default-assets*).

For all workload cookbooks, the available configuration options can be derived from the respective attributes files *<cookbook-name>/attributes/default.rb*

4.5 Benchmark Architecture

Iosup et al. [ICH+14] propose in their work a generic benchmark architecture. We use their architecture as a template and duly appropriate it to match our needs. The benchmark architecture slightly varies depending on the JMeter mode (2.8.3). If JMeter is executed in the simple mode, only one JMeter instance is acquired. If the more complex, distributed JMeter mode is applied, JMeter slave instances are added to the driver system in order to generate the workload in a distributed manner. The resulting benchmark architecture is depicted in Figure 4.1. Whatsoever JMeter mode is chosen, the CWB Server is always the initial starting point for every benchmark execution. Vagrant acquires instances at the cloud provider of choice and the Chef Client provisions them with the Chef cookbooks provided by the Chef Server. In most of the cases, a distributed JMeter setup is used to generate load to the SUT. In Figure 4.1, the SUT shows AcmeAir, consisting of a web application instance and a database instance. This setup includes one JMeter instance acting as master and several JMeter instances acting as slaves. The JMeter master sends the JMeter Test Plan to the JMeter slave instances and starts the test execution. In specified intervals the JMeter-Slaves send the collected results back to the JMeter master which collects and aggregates the results. After the test is terminated, the JMeter-master processes the results and send the individual metrics back to the CWB server. A more detailed description of the benchmark execution is provided in the background section, namely in Table 2.1.

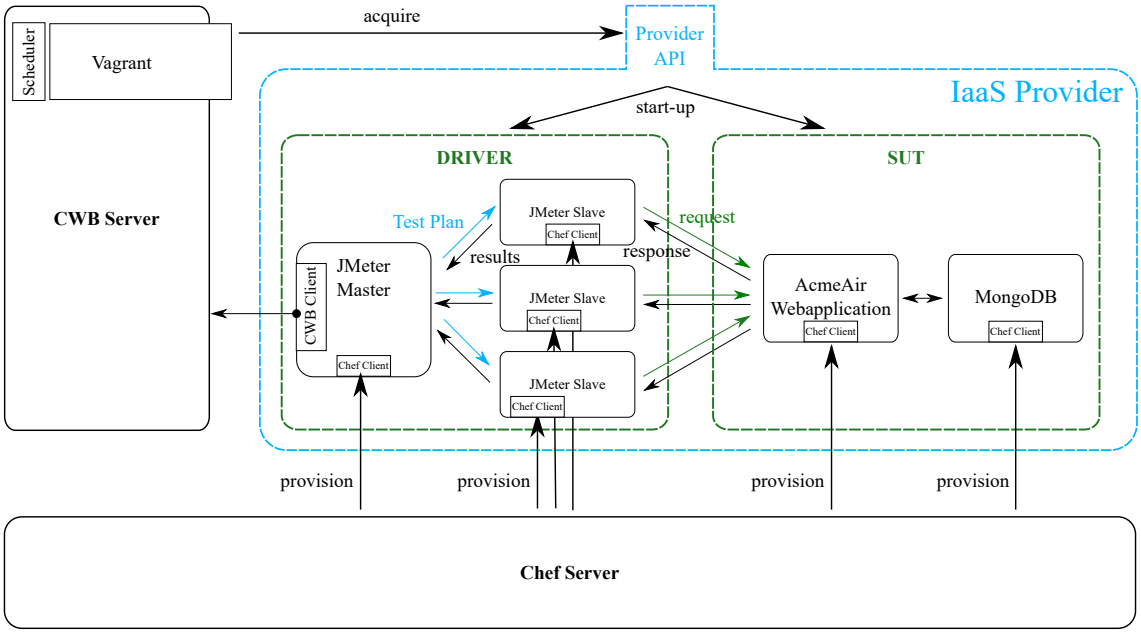


Figure 4.1: Benchmark Architecture (SUT: AcmeAir)

Chapter 5

Experimental Setup

To benchmark different IaaS providers, we propose a generic application benchmark which is designed to be cross-provider native. To showcase this ability, we have chosen two representative IaaS providers for our study, namely Amazon EC2 and GCE. Amazon EC2 has been selected because of its presence in related benchmarking literature (e.g., [SDQR10, DPC10, MH12, KKR14]) and its relevance in practice. In contrast to Amazon EC2, which started its service in 2007, GCE offers its service only since 2013 [Ryz15] and is given centre stage only in few performance benchmarking studies.
This chapter is dedicated to stating the details of the experimental setup, the individual experiments and their execution. The collected data is presented in Chapter 6.

We used CWB to set up the SUT and to collect a relevant amount of performance measurements. Some complete examples of used CWB benchmark definitions can be found in the appendix, namely in Section 9.4.

5.1 Used Workload

The workload we used for the experiment is derived from the workload Spyker [Spy13] used. We created a Chef cookbook for it. This allows us to implement the workload in a configurable manner and, in a later phase, to create easily different versions of the workload without changing any code. Additionally, we also fixed a minor bug.

As outlined in Section 4.4.3, does the original workload test mainly the API service. Therefore, we call our adapted version of the workload originally proposed by Spyker [Spy13] simply "API" and the related workload coobook *jm-acmeair-api*.

To reduce code duplicity and to ease configuration even further, we split the workload into two cookbooks: one cookbook for the JMeter Test Plan *(jm-acmeair-api)* and another cookbook for static files, such as custom implementations of post-processors, CSV files, and so forth. Since the originally proposed workload depends on the same files, we can simply reuse the *jm-acmeair-default-assets* cookbook.

5.2 Bugfix and Adaptations

During some pre-test runs we observed a reduction in active JMeter threads as soon as the server experienced heavy load. This reduction in active threads was caused by requests, which tried to access non-existent data collected by prior requests. When the web service is not able to answer a request or the request timed out, an error code is registered without any response body. In some

cases, subsequent requests tried to access data in the response body, which, of course, were not present. We fixed this bug by changing the behaviour on failure. Concretely, we changed the following in the JMeter Test Plan:

```
<stringProp name="ThreadGroup.on_sample_error">continue</stringProp>
```

to:

```
<stringProp name="ThreadGroup.on_sample_error">startnextloop</stringProp>
```

Moreover we change the behaviour of the cookie manager. In Order to simulate an authentic login behaviour, we changed the cookie-clearing policy of the cookie manager to clear all cookies at the end of an iteration. Concretely, we changed the following lines of code in the JMeter Test Plan:

```
<boolProp name="CookieManager.clearEachIteration">false</boolProp>
```

to:

```
<boolProp name="CookieManager.clearEachIteration">true</boolProp>
```

5.3 Monitoring

To be able to monitor the web application as well as the different JMeter instances, we developed the *cwb-monitoring* cookbook, a simple wrapper cookbook that runs all commands required to (1) be able to instantly connect VisualVM[1] to the instance and (2) to start vmstat[2].

VisualVM Java applications are only allowed to use a limited amount of memory, which is further divided into Heap Space and Permgen. Both can be tuned by the developer. VisualVM allows to monitor and track the memory and related information by accessing the Java Virtual Machine (JVM). This was especially useful to determine the maximum number of JMeter Threads a single JMeter instance is able to run without breaking down.

vmstat vmstat reports information about processes, memory, paging, block IO, traps, and cpu activity and thus can be used to monitor the actual resource consumption during the test run. We configured vmstat to log the current stats to a file in order to be able to post process this file in the case of irregularities. We used vmstat especially to make sure, that the VMs performance is not throttled by the hypervisor (as for instance in the case of bursting instance types running out of credit [LS15]) and to find bottlenecks in the configuration.

5.4 Tuning

At some point in our pre-testing phase, we encountered an error message indicating that the operating system was not able to allocate enough file handles. To tune the OSs, we developed the *cwb-tuning* cookbook. It simply increases the allowed maximum number of open file handles.

5.5 Cookbooks and Configuration

This section provides the details on which instance was provisioned with what cookbooks and what configuration is used.

[1] https://visualvm.java.net/
[2] https://wiki.ubuntuusers.de/vmstat/

5.5.1 System under Test

For our experiments, we used the distributed AcmeAir architecture, deploying the web application on a dedicated instance and the database to another instance. While the configuration varied slightly from provider to provider, we provisioned all instances (of the same function) with the same cookbooks. Examples of complete CWB benchmark definitions for each provider and showing all components, can be found in the appendix in Section 9.4.

Web Application Instances The web application has for both providers been provisioned with the *acmeair_wlp_morphia_distributed* cookbook, which sets up the Websphere Liberty server as well as the AcmeAir web application. Moreover, we provisioned the wep application instances with the *cwb-tuning* and the *cwb-monitoring* cookbooks.

The web application has to know the internal or external ip address from the instance hosting the database. The Vagrant provider (cloud provider plugin) for Amazon EC2 allows to setup a virtual private cloud (VPC) in a certain availability zone. This is allows to setup the different instances with fixed internal ip addresses. These are determined by a simple calculation, but could also be configured manually. In contrast to Amazon EC2, the Vagrant provider for GCE does not allow to setup VPCs or any other kind of fixed internal ip addresses. Therefore, we had to implement an additional script for the lookup of the internal ip address through the GCE metadata service. In consequence, the configurations a slightly different, but do not influence the experiments in any way. For each instance size, we also tuned the heap space.

However, apart from the ip address configuration and updates to the heap space configuration, all configuration are identical. The differing parts in the configuration are depicted in Listings 5.1 and 5.2.

```
1  chef.json = {
2    'config' => {
3      'tuning' => {
4        #adapted for each instance size: max memory - 512mb
5        'heap_xms' => '512m',
6        'heap_xmx' => '3g'
7      }
8    },
9    'mongodb' => {
10     #'ip_from_file' => false,
11     #'ip_file_path_name' => '/home/admin/ip.env',
12     'ip' => IP_DB,
13     'name' => 'acmeair',
14     'port' => 27017,
15     'user' => {
16       'name' => 'acmeairusr',
17       'password' => 'Login4Acme!'
18     }
19   }
20 }
```

Listing 5.1: Webapp Configuration EC2

```
1  chef.json = {
2    'config' => {
3      'tuning' => {
4        #adapted for each instance size: max memory - 512mb
5        'heap_xms' => '512m',
6        'heap_xmx' => '3g'
7      }
8    },
9    'mongodb' => {
10     'ip_from_file' => true,
11     'ip_file_path_name' => '/home/admin/ip.env',
12     'ip' => 'value should be overwritten!',
13     'name' => 'acmeair',
14     'port' => 27017,
15     'user' => {
16       'name' => 'acmeairusr',
17       'password' => 'Login4Acme!'
18     }
19   }
20 }
```

Listing 5.2: Webapp Configuration GCE

Database Instances In contrast to the web application instances, the configuration for the database instance did not change at all. The database instance were provisioned only with the *acmeair_mongodb* cookbook.

5.5.2 Driver and Workload

During our experiments, we used both, the simple and the distributed JMeter mode. Nevertheless did the configuration and the selected cookbooks for the JMeter master instance only vary in few points, namly in the configuration parameter specifying if the master instance has slaves or executes the benchmark by itself.

To collect the data which is used later to evaluate the performance of the different instance types, we only used the workload described in Section 5.1 and the related workload cookbooks *jm-acmeair-api* (JMeter Test Plan) and *jm-acmeair-default-assets* (files required for the test plan).

Workload

Analogously to the web application instance which has to determine the ip address of the database instance for every provider in an appropriate way, the JMeter master has to query the ip address of the web application instance. To this end, the same mechanism (Section 5.5.1) is applied. Thus, a small difference in the configuration is required.

Note: The workload cookbook is provisioned to the JMeter master instance.

```
1  'acmeairapi' => {
2    'testplan' => {
3      'user_in_db' => 1000000,
4      'connection_timeout' => 30000,
5      'response_timeout' => 30000,
6      'target_host' => {
7        'port' => 9080,
8        'name' => 172.31.3.1,
9        #'name_from_file' => true,
10       #'file_path_name' => '/home/admin/ip.env'
11     }
12   },
13   'threadgroup' => {
14     'num_threads' => 2500,
15     'ramp_up_time' => 0,
16     'duration' => 1200,
17     'delay' => 0
18   }
19 }
```

Listing 5.3: Worklaod Configuration EC2

```
1  'acmeairapi' => {
2    'testplan' => {
3      'user_in_db' => 1000000,
4      'connection_timeout' => 30000,
5      'response_timeout' => 30000,
6      'target_host' => {
7        'port' => 9080,
8        'name' => 'this should be overwritten!',
9        'name_from_file' => true,
10       'file_path_name' => '/home/admin/ip.env'
11     }
12   },
13   'threadgroup' => {
14     'num_threads' => 2500,
15     'ramp_up_time' => 0,
16     'duration' => 1200,
17     'delay' => 0
18   }
19 }
```

Listing 5.4: Workload Configuration GCE

JMeter Master Instances To provision the JMeter master instance, the previously implemented cookbook *cwb-jmeter* has been used. Throughout all of our experiments, we used the same cookbooks with the same configuration.

The *acmeair-single* cookbook is the generic application benchmark adapter for CWB. It implements the API, which is required for CWB to be able to start and terminate the benchmark execution and which triggers the post-processing of the benchmark results and the subsequent transfer of the metrics back to the CWB server. All in all, the JMeter master instance is provisioned with the following cookbooks:

- jm-acmeair-api
- jm-acmeair-default-assets
- acmeair-single (the generic application benchmark adapter for cwb)
- cwb-tuning
- cwb-monitoring

JMetre Slaves Instances The JMeter Slave instance are also provisioned with the same cookbooks apart from the integration cookbook and the test plan cookbook. The JMeter Test Plan is at execution time sent along by the JMeter master instance. Thus, slaves instances are provisioning with the following Chef cookbooks:

- cwb-jmeter
- jm-acmeair-default-assets
- cwb-tuning
- cwb-monitoring

5.6 OS, Architectures, Storage, Images

All tests were executed with Debian 8.3 (Jessie) as operating system (OS) based on a x84_64 architecture. In Amazon EC2, we used the 8 GiB of EBS storage and HVM virtualization. On GCE we used 10 GB of type "persistent disk" storage and the default virtualization technique.

To speed up the provisioning phase, we created custom VM images for both Amazon EC2 and GCE. This is not a mandatory step, but helps to reduce the time required for provisioning and by doing so helps to prevent time-outs during the provisioning phase. However, the provisioning itself still is done by Chef. The Chef recipes are executed as in the case if the software were not installed already, but with skipping the software installation part.

Concretely, we package for each cloud provider an image with the following properties:

OS:	Debian
Architecture:	x86_64
Packages:	openjdk-7-sdk,
Commands:	apt-get update

Table 5.1: Image specification

As base images, we used *ami-47d93c28* ,and *debian-8-jessie-v20160718* respectively.

5.7 Instance Specifications and Prices

The specifications of the instance sizes and related prices for the instances used in the experiments with Amazon EC2 are depicted in Table 5.2, the ones for GCE in Table 5.3.

5.8 Deployment Details

CWB and Chef We deployed the CWB server as well as the Chef server on Amazon EC2 and used this deployment to benchmark both providers. The CWB server was deployed to a Amazon t2.medium instance, while the Chef server was deployed to a t2.small instance. Since the files submitted by the JMeter master instances can be quite big (especially for long benchmark executions), the file server has raised requirements regarding heap space.

Instance Type	Instance Size	vCPU	RAM (GiB)	Storage	Price / h*	ECU**
Bursting Performance	t2.micro	1	1	EBS	$0.015	Variable
	t2.small	1	2	EBS	$0.03	Variable
	t2.medium	2	4	EBS	$0.06	Variable
General Purpose (latest)	m4.large	2	8	EBS	$0.143	6.5
	m4.xlarge	4	16	EBS	$0.285	13
General Purpose (old)	m3.medium	1	3.75	SSD	$0.079	3
CPU Optimized (latest)	c4.large	2	3.75	EBS	$0.134	8
	c4.xlarge	4	7.5	EBS	$0.267	16

* Prices for Frankfurt region on 2016/08/08
** Elastic Compute Unit

Table 5.2: EC2 Prices [Ama16c]

Instace Type	Machine Type	vCPU	RAM (GB)	Storage	Price / h*	GCU**
Bursting Instance	g1-small	1	1.7	HDD / SSD	$0.030	1.38
CPU optimized	n1-highcpu-2	2	1.8	HDD / SSD	$0.084	5.5
	n1-highcpu-4	4	3.6	HDD / SSD	$0.168	11
General Purpose	n1-standard-1	1	3.75	HDD / SSD	$0.055	2.75
	n1-standard-2	2	7.5	HDD / SSD	$0.110	5.5
	n1-standard-4	4	15	HDD / SSD	$0.220	11

*prices for Europe/Asia on 2016/08/08
** Google Compute Unit

Table 5.3: Google Compute Engine Pricing [Goo16c]

SUT Configurations The different deployment configurations used for the SUT in the different experiments are depicted in Table 5.4. Each configuration code takes the form $provider\_type\_dbconf$, where $provider$ is either Amazon EC2 (A) or GCE (G), $type$ is bursting(b), non-bursting(nb), general purpose(gp), or computation optimized(co). $dbconf$ denotes the database instance type used, such that the higher number represents the configuration with the better specification (more cpu or more memory or better storage).

In all deployments, were the SUT and the driver deployed to the same availability zone. For Amazon EC2, this is *eu-central-1a* in the Frankfurt region (eu-central-1). For GCE, the zone is *europe-west1-b*.

5.9 Benchmark Execution Details

In Table 5.5, we summarize the details about the individual benchmark executions configurations. This table lists all benchmark executions, which contributed to the set of data evaluated in the

Configuration Code c	Webapp	DB
A_nb1m	t2.micro	t2.micro
A_b1m	t2.micro	t2.micro
A_nb1s	t2.small	t2.micro
A_b1s_1	t2.small	t2.micro
A_b1s_2	t2.small	t2.small
A_nb2	t2.medium	t2.micro
A_b2_1	t2.medium	t2.micro
A_b2_1	t2.medium	t2.small
A_gp2_1	m4.large	t2.small
A_gp2_2	m4.large	m3.medium
A_gp4	m4.xlarge	t2.small
A_co2_1	c4.large	t2.small
A_co2_2	c4.large	m3.medium
A_co4	c4.xlarge	t2.small
G_b	g1-small	g1-small
G_gp_1	n1-standard-1	n1-standard-1
G_gp_2	n1-standard-2	n1-standard-1
G_gp_4	n1-standard-4	n1-standard-1
G_co_2	n1-highcpu-2	n1-highcpu-2
G_co_4	n1-highcpu-4	n1-standard-1

Table 5.4: Instance Configuration Index

results section in Section 6.

5.10 Data Storage and Post-Processing

As introduced in the Background section (2.8.2), JMeter produces in each test run a results file. This results file contains information about every request sent during the test run and allows therefore manifold insights. Hence, the result files were transferred from the instances before these were destroyed. For this purpose, we added an external fileserver to our benchmark setup. The file server is also implemented as Chef cookbook and thus can be easily deployed with and without CWB.

5.11 Error Handling

In some cases, we had to deal with transient faults, which originated either in the cloud provider (e.g., instances not starting up correctly) or in our own tooling (e.g., timeouts from the Chef server used for instance provisioning). In these cases, we have simply cancelled the benchmarking run and discarded the resulting measurements.

Configuration Code c	Total Count	Count	JMeter Instances	Threads	Total Threads	Exec. Time	Ramp-up Time
A_b1m	24	4	12	3000	36000	600	600
		3	14	3500	49000	600	600
		11	20	2500	50000	600	600
		6	24	1500	36000	600	600
A_b1s_1	26	11	12	3000	36000	600	600
		7	14	3500	49000	600	600
		3	20	3000	60000	600	600
		5	24	2500	60000	600	600
A_b1s_2	16	11	20	3000	60000	600	600
		5	24	2500	60000	600	600
A_b2_1	11	1	14	3500	49000	600	600
		10	20	3000	60000	600	600
A_b2_2	16	16	20	3000	60000	600	600
A_nb1m	18	1	1	2500	2500	6000	0
		2	1	2500	2500	4200	0
		15	1	1500	1500	4200	0
A_nb1s	17	1	1	2500	2500	3000	0
		1	1	2500	2500	6600	0
		15	1	2500	2500	4200	0
A_nb2	14	1	1	2500	2500	6600	0
		13	1	2500	2500	4200	0
A_co2_1	35	1	24	2000	48000	600	600
		11	24	3500	84000	600	600
		23	1	5000	5000	600	0
A_co2_2	26	26	1	5000	5000	600	0
A_gp2_1	37	11	20	3000	60000	600	600
		3	24	3000	72000	600	600
		23	1	5000	5000	600	0
A_gp2_2	27	27	1	5000	5000	600	0
A_co4	19	3	24	3000	72000	600	600
		16	1	5000	5000	600	0
A_gp4	23	2	20	3000	60000	600	600
		1	20	4000	80000	600	600
		2	24	3000	72000	600	600
		1	24	3500	84000	600	600
		17	1	5000	5000	600	0
G_b1	13	13	1	2500	2500	600	0
G_gp1	26	26	1	2500	2500	600	0
G_gp2	26	26	1	2500	2500	600	0
G_gp4	24	2	1	2500	2500	600	0
		22	1	5000	5000	600	0
G_co2	18	18	1	2500	2500	600	0
G_co4	23	23	1	5000	5000	600	0

Table 5.5: Benchmark execution details

5.12 Used Metric

For the evaluation of the results, we post-processed all files gathered during the different test runs. As metric, we decided to choose the mean of the successful request counts per second the SUT can sustain. Moreover, in order to get as accurate data as possible, we developed an algorithm to extract the relevant part from all results of a benchmark execution.

5.12.1 Mean Successful Requests per Second $\overline{SRPS}$

In our benchmarking efforts, we try to gather data on the maximum throughput of the web application. Concretely, we are interested in the mean amount of successful request the web application can handle per second, i.e.,

$$MSRPS = \overline{SRPS} = \frac{1}{k}\sum_{i=1}^{k} SRPS_i \tag{5.1}$$

Where

$k \in \mathbb{N}$ is the execution duration in seconds
$SRPS_i$ is the successful requests count for second i

A similar metric is also used by TPC-W. The primary metrics of TPC-W are the "WIPS rating" and "system cost per WIPS". In TPC-W, WIPS is defined as the number of web interactions per second that can be sustained by the SUT [Smi00].

The WIPS metric has several advantages compared to other common performance metrics such as response time or count of concurrent users. Firstly, WIPS is not network sensitive. For our use-case, this has the benefit that we can setup the driver and the SUT at the same cloud provider which allows a fully automated configuration. If the SUT and the driver are not hosted at the same cloud provider, some kind of registry service has to be implemented in order to make the driver aware of the target's ip address. Secondly, the arrival rate of the requests has no impact on the maximum throughput, as long as there are more requests arriving than can be served. If more requests arrive than the server can serve, they are queued up. This has an impact on the observed response time, but not on the MSRPS metric, as long as the server is not overloaded (which results in an immediate drop).

5.12.2 Extraction Algorithm

To ensure our results are accurate and represent the actual performance of the different deployment configurations, we developed an algorithm to extract a relevant part of the data. We especially aim at eliminating the warm-up phase as well as potential shut-down disruptions. Our algorithm uses three input parameters, namely (1) how long the relevant sampling duration is, and (2) how much padding should be added at the end of the sampling period, representing the time a typical shut down takes and (3) a gap size used to ignore gaps of that size in the data. An application example is shown in Table 5.6. Figure 5.2 depicts a real application. The length of the black bar indicates the length and the actual position of the identified relevant data. The hight of the bar indicates the $\overline{SRPS}$ if computed over the extracted data.

We list in Table 5.6 some sample data. For our example, we want to extract only the relevant data, namely the data from second 5 to second 9. Therefore we specify as sampling duration 8 and as padding 3. For our example, the resulting $\overline{SRPS} = 1300$ and the standard deviation $\sigma_{SRPS} \simeq 31.622$

SRPS	100	400	700	1250	1300	1300	1250	1350	1300	800	400	20
time in seconds	1	2	3	4	5	6	7	8	9	10	11	12
Sampling Duration			5			$\overline{SRPS}$			1300			
Padding			3			σ_{SRPS}			31.6227766			

Table 5.6: Extraction Algorithm Example

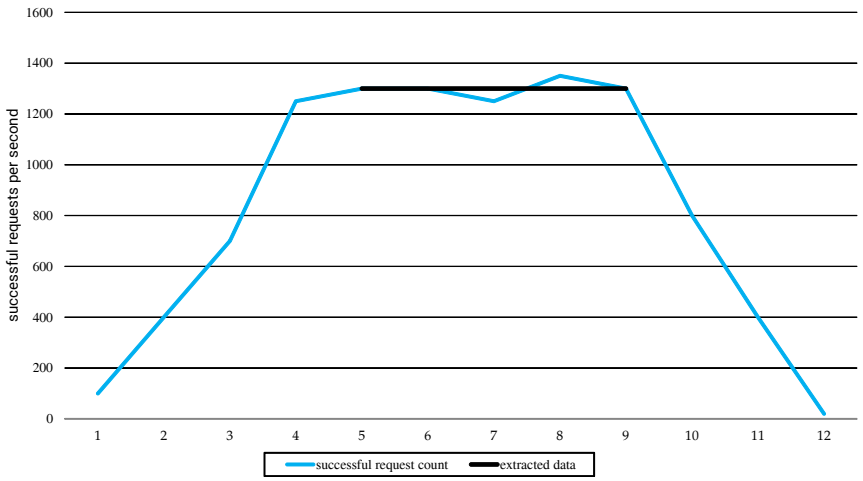


Figure 5.1: Visualization of the example

For all of our calculations and further analysis, we used a sampling period of 130 seconds and a padding of 10 seconds, which corresponds to a nett sampling period of 120 seconds. Figure 5.2 shows the application of the algorithm to the single measurements from different configurations. The length of the black bar indicates the length and the actual position of the identified relevant data. The hight of the bar indicates the $\overline{SRPS}$ if computed over the extracted data.

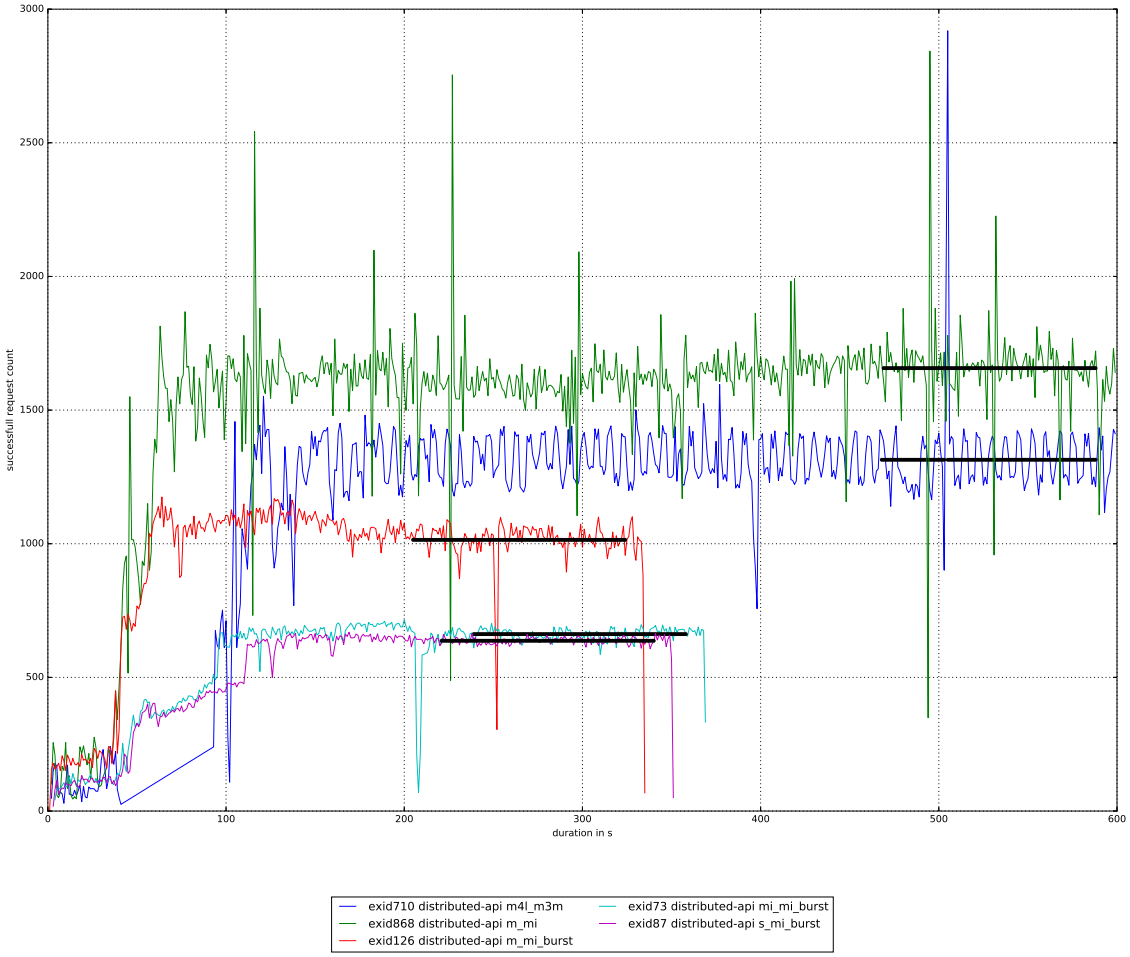


Figure 5.2: Visualization of real data

Chapter 6

Experimental Results and Discussion

Leitner and Cito [LC16] conducted a principled literature review and present the state of the art in existing research on performance variation in public IaaS offerings. Thereby, they formulate 15 hypothesis relating to possible factors influencing the experienced performance.

In the following sections, we start out by verifying that we collected a relevant amount of data. Then we evaluate, if the size of the database instance represents a bottleneck. If so, the results obtained from our benchmarks cannot be accredited only to the performance of the web application instance.

Having these basic preconditions covered, we continue by drawing on some of the 15 hypothesis constructed by Leitner and Cito [LC16] and discuss and compare their reported results with the results revealed in this work. To this end, we first briefly explain the elimination of hypothesis that cannot be answered with our own data, and then validate the remaining hypothesis one after another with one exception: we conduct the evaluation of the performance of larger instance types compared to smaller instance types (H:4.2) just after introducing the formulas for instance stability (H:2.2) and discussing the special case of bursting instance types (H:2.3).

6.1 Preliminaries

In the following two sections, we cover the preconditions we base our evaluation on and which have to be met, in order to compare our results to the results obtained by Leitner and Cito [LC16].

6.1.1 Relevant Amount of Data

In total we tested 20 different configurations. Table 6.1 summarizes the conducted experiments. For each configuration, the number of executed benchmarks is indicated as #. Moreover, the columns I and II show how many benchmark executions of the same configuration had to be run, in order to ensure, that the population mean lays within a confidence interval of size 5% (also known as margin of error MOE) at a confidence level of 95% (i.e., 5% of the samples will lay beyond the confidence interval) in column I and 99% in column II respectively. To calculate this statistic, we used the common equation to estimate a population mean based on a confidence interval of a fixed size (e.g., [Pat11]). The equation is given in Equation 6.1.

$$N = \left\lceil \left(\frac{z^* \cdot \sigma}{T} \right)^2 \right\rceil \tag{6.1}$$

Where

$N =$ required sample size

$z^* =$ z*-value from the confidence level table

$\sigma =$ standard deviation (as number)

$T =$ the margin of error (as number)

For our experiments, we used a confidence interval of size 5% of the sample mean and therefore we used for the calculations $T = \overline{x} \cdot p$, where $\overline{x}$ is the sample mean and $p = 0.05$. Moreover we used $Z = \{1.96, 2.576\}$ and $z^* \in Z$.

Table 6.1 summarizes the results, if we consider the default sampling duration to be 120s and the extraction algorithm is used to determine the relevant set of data, as described in Section 5.12. Table 6.4 shows the same configurations, but with configuration optimal sampling durations in combination with visual inspection to determine the relevant set of data. The optimal sampling duration was found via a thorough visual inspection.

Configuration Code	#	σ	95%CL ($z^* = 1.96$) I	#−I	99%CL ($z^* = 2.576$) II	#−II
A_b1m	24	63.69	10	14	17	7
A_b1s_1	26	47.03	7	19	12	14
A_b1s_2	16	72.27	12	4	21	-5
A_b2_1	11	254.08	69	-58	119	-108
A_b2_2	16	262.15	70	-54	121	-105
A_nb1m	18	2.67	2	16	3	15
A_nb1s	17	4.33	1	16	2	15
A_nb2	14	11.56	2	12	3	11
A_co2_1	35	147.19	19	16	33	2
A_co2_2	26	26.76	1	25	1	25
A_gp2_1	37	120.63	16	21	27	10
A_gp2_2	27	31.26	1	26	2	25
A_co4	19	138.88	7	12	12	7
A_gp4	23	119.89	7	16	11	12
G_b1	13	17.84	5	8	9	4
G_gp1	26	18.24	1	25	2	24
G_gp2	26	45.02	3	23	5	21
G_gp4	24	79.26	3	21	5	19
G_co2	18	37.06	2	16	4	14
G_co4	23	85.68	4	19	6	17

Table 6.1: Statistical Significance

6.1.2 Relevance of the Database-instance Size

In order to be able to draw meaningful conclusions, we analyse whether the size of the instance hosting the database has an impact on the performance of the web application, i.e., if the database instance size constitutes a performance bottleneck and thus plays a relevant role. If we can proof the database size to influence the performance of the web application, this would prohibit to relate the performance of the web application to the performance of the instance hosting the web application.

To compare the means of the configurations to each other, we use the Mann-Whitney U-test. The Mann-Whitney U-test (also called Wilcoxon rank sum test, or Wilcoxon-Mann-Whitney test) is used, when it is asked to compare the non-normally distributed means of two independent samples. It is a non-parametrical test and it is the equivalent of the t-test but applied to independent samples and with fewer preconditions [BCMS14]. The goal consists of comparing the central tendencies of the two samples, in order to test whether the locations of the respective populations are equal, thus the assumed null hypothesis H_0 is that there is *no* difference in the underlying distribution. The alternative hypothesis H_1 is consequently that of a *relevant* difference in the $\overline{SRPS}$ between configurations (two-sided test), i.e.,

$$H_0 : \overline{SRPS_j} \stackrel{d}{=} \overline{SRPS_k} \text{ and } H_1 : \overline{SRPS_j} \stackrel{d}{\neq} \overline{SRPS_k}, \forall j, k \in C, j \neq k \tag{6.2}$$

Where the operator $\stackrel{d}{=}$ indicates the equality in the underlying distribution and C is the set of all benchmarked configurations.

6.1.3 Bursting Instance Types

We tested on a significance level of $\alpha = 0.01$. The resulting p-values for all configurations are listed in Table 6.2 and all statistically significant p-values (i.e., p-values < 0.01) are for readability reasons coded as $*$. If the p-value of the observed test statistic is smaller than α, this leads to reject the null hypothesis H_0 in favour of the alternative hypothesis H_1 and indicates, that the $\overline{SRPS}$ statistically differ on a confidence level of $1 - \alpha = 0.99$.

c_1	c_2	p-value
A_b1s_1	A_b1s_2	0.444826999
A_b1s_1	A_b2_1	*
A_b1s_1	A_b2_2	*
A_b1s_2	A_b2_1	*
A_b1s_2	A_b2_2	*
A_b2_1	A_b2_2	0.335921852

Table 6.2: Mann-Whitney-U for bursting instance types

In the case of the bursting instance types, we can summarize that similar configurations, which only differ in the size of the instance hosting the database, cannot be shown to be statistically different. Also, the visualization (Figure 6.1 and 6.2) of the $\overline{SRPS}$ looks as expected. In Figure 6.1, the upper error bars represent the 75^{th} percentile and the lower error bars the 25^{th} percentile.

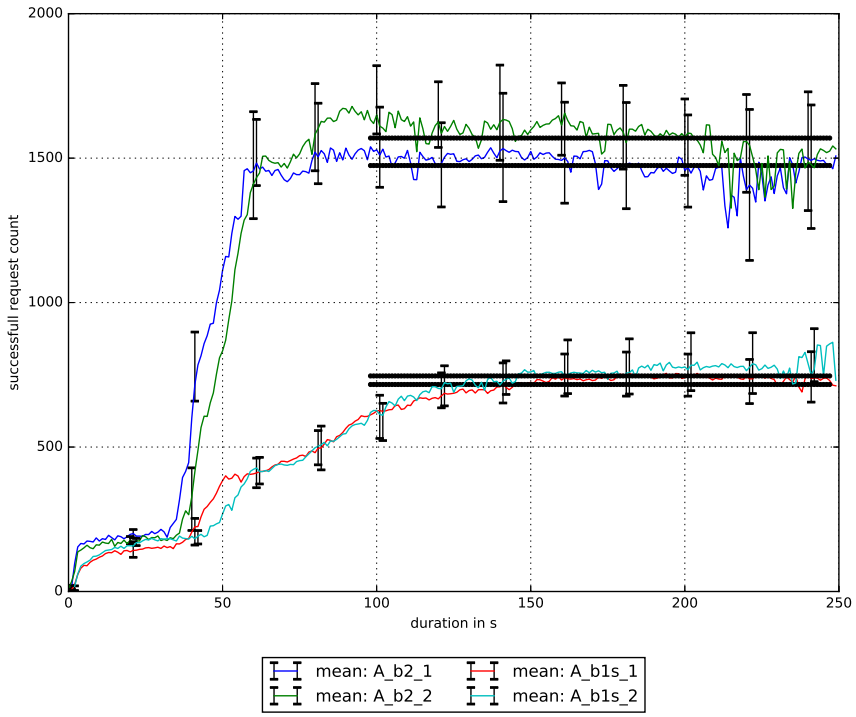


Figure 6.1: Deployment configurations for bursting instances, lineplot

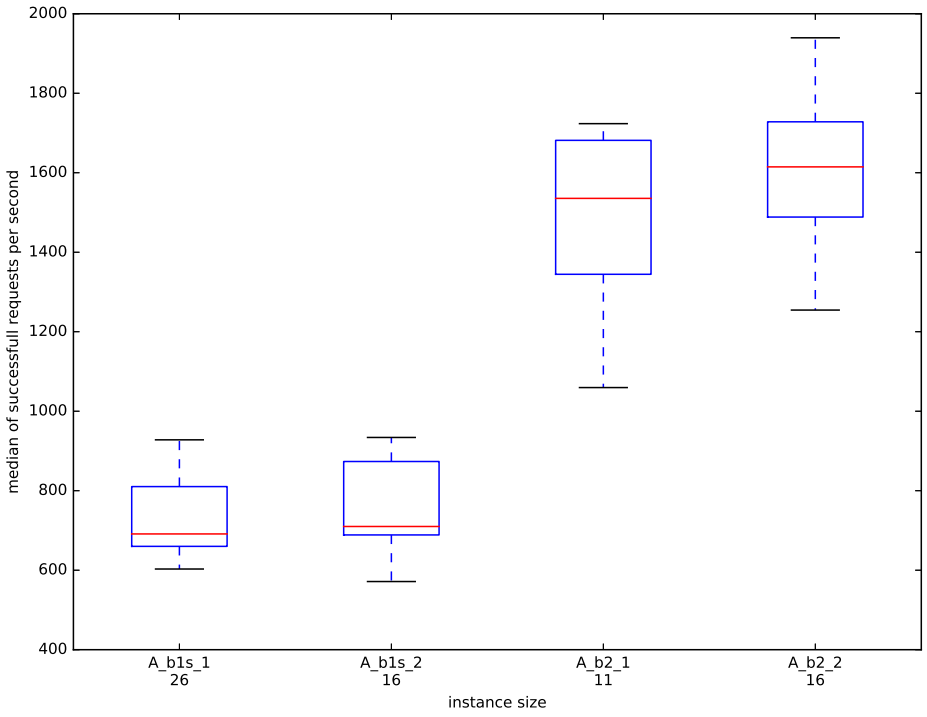


Figure 6.2: Deployment configurations for bursting instances, boxplot

6.1.4 Non-Bursting Instance Types

We also compared the $\overline{SRPS}$ for the non bursting instance types. As for bursting instance types, we also tested in the case of non-bursting instance types on significance level $\alpha = 0.01$. The resulting p-values for all configurations are listed in Table 6.3 and all statistically significant p-values (i.e., p-values < 0.01) are for readability reasons coded as $*$.

c_1	c_2	p-Value
A_co2_1	A_co2_2	0.097864455
A_co2_1	A_gp2_1	*
A_co2_1	A_gp2_2	*
A_co2_2	A_gp2_1	*
A_co2_2	A_gp2_2	*
A_gp2_1	A_gp2_2	0.037534628

Table 6.3: Mann-Whitney-U for non bursting instance types

In the case of the non-bursting instance types, we can also conclude that similar configurations, which only differ in the size of the instance hosting the database, cannot be shown to be statistically different. Also the visualizations of the throughput point in the same direction 6.3 6.4. In Figure 6.3, the upper error bars represent the 75^{th} percentile and the lower error bars 25^{th} percentile.

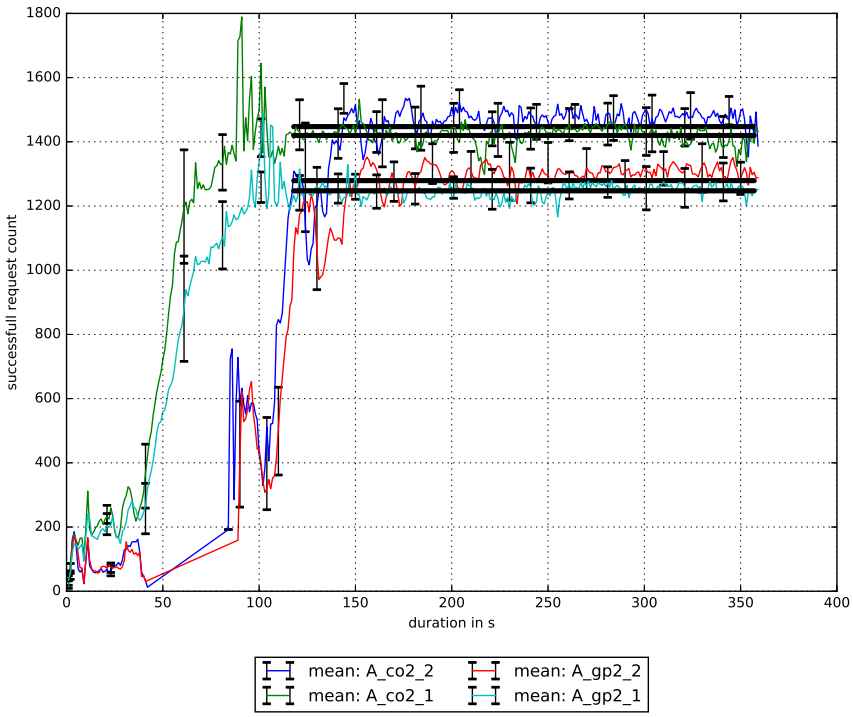


Figure 6.3: Deployment configurations for non-bursting instances, lineplot

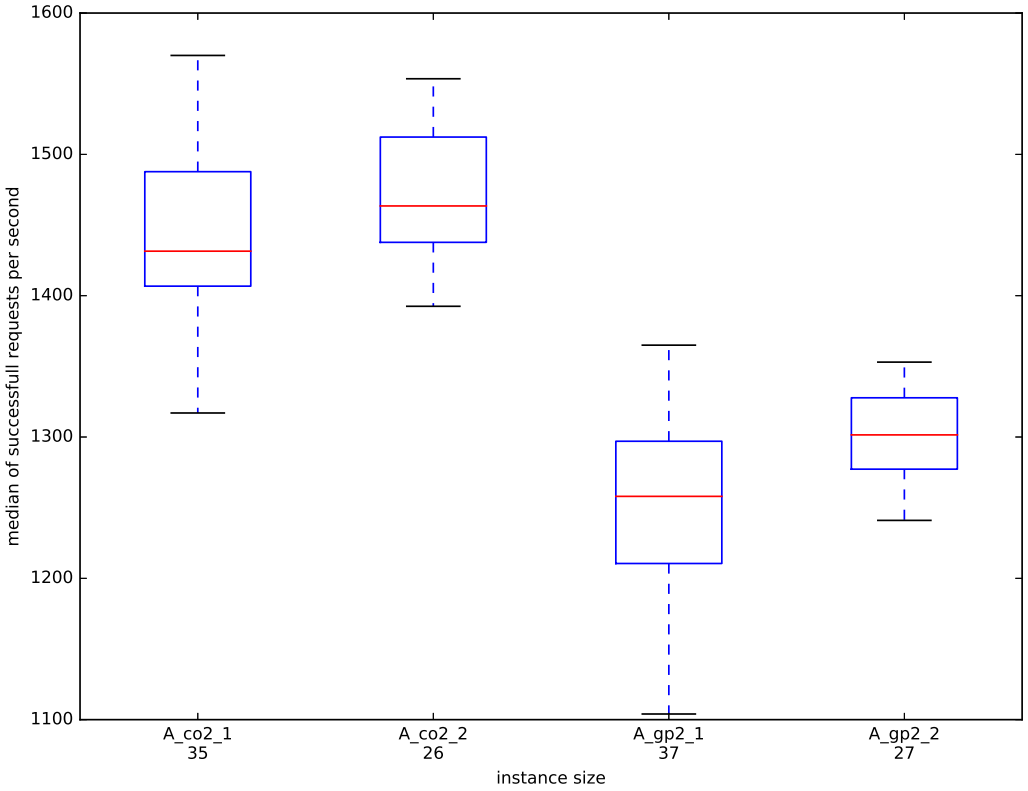


Figure 6.4: Deployment configurations for non-bursting instances, boxplot

6.2 Hypothesis Validation

The 15 hypothesis reported by Leitner and Cito [LC16] belong to four groups: Performance Predictability (H1), Variability Within Instances (H2), Temporal and Geographical Factors (H3), and Instance Type Selection (H4). From group H1, we can only validate hypothesis H1.1. "Performance Varies Between Instances" is treated in Section 6.2.1. H1.2 "CPU-Bound Applications" cannot be validated, since our data does not reveal any insights about the underlying CPU model and H1.3 "IO-Bound Applications" cannot be validated because our application is mainly CPU intensive. This is equally true for H2.1 from group H2. H2.2 "Intra-Instance Stability of CPU-Bound Applications" is approached in Section 6.2.2 and H2.3 "Intra-Instance Variability of Bursting Instance Types" in Section 6.2.4. Group H3 has to be excluded as a whole, since we did not collect any data on time or location. Moreover, for each cloud provider we conducted all of our experiments in the same region and in the same availability zone. From group H4, hypothesis H4.1: "Diseconomies of Scale of Larger Instance Types" is validated in Section 6.2.5, H4.2: "Stability of Larger Instance Types" in Section 6.2.3 and H4.3: "Price of Specialization" in Section 6.2.6.

6.2.1 Instance Size Performance Predictability

In order to answer the question about the predictability of the performance of an instance exhibiting a certain instance size specification, we interpret the performance predictability as the inverse of the intra instance-size performance variability: the lower the variability, the higher the predictability. Relating to the intra instance-size variability, Leitner and Cito [LC16] construct the following hypothesis: *H1.1: Performance Varies Between Instances – the performance of cloud instances using the same configuration tends to vary relevantly [LC16].*

Leitner and Cito [LC16] provide in their work the required calculations, which are as described in the following. In the context of intra instance-size variability, a configuration $c \in C$ is as a triple of cloud provider, web application instance size and db instance size, where C is the set of all tested configurations. Every configuration has an associated set of instances $I_c \subseteq I$ where I_c is the set of instances with configuration c and I is the set of all instances.

For every instance i with a given configuration c, i.e., every i in I_c , we collected performance measurements $m(i) = \{m_1(i), m_2(i), \ldots, m_n(i)\}$[1] where $m_k(i)$ is in $\mathbb{R}$ and all k in $\{1, \ldots, n\}$. Leitner and Cito [LC16] define M_c to be the union of all measurements of instances of configuration $c \in C$, i.e.,

$$M_c = \bigcup_{i \in I_c} m(i) \tag{6.3}$$

Note: To simplify the notation, we allow duplicates in the resulting set M_c.

Leitner and Cito [LC16] use in their work the relative standard deviation as a measure of performance variability. Analogously, we calculate the relative standard deviation (c_{RSD}) of the measurements collected for each configuration with $\overline{M_c}$ referring to the arithmetic mean of M_c, and σ_{M_c} referring to the standard deviation of M_c.

$$\forall c \in C : c_{RSD} = 100 \cdot \frac{\sigma_{M_c}}{\overline{M_c}} \tag{6.4}$$

[1] We look at a special the special case, where $m(i) = \{m_1(i)\}$ and $m_1(i)$ is the $\overline{SRPS}$ of an instance i.

Additionally to the calculations, Leitner and Cito [LC16] assume that for most use cases a relative standard deviation of more than 5% constitutes a relevant variability in performance and indicate, that specific applications may take higher or lower variability [LC16]. All resulting relative standard deviations c_{RSD} of all configurations are shown in Table 6.4. The mean relative standard deviation ($(\overline{c_{RSD}})$) is already shown in the table but is only going to be used in the Section 6.2.2.

Instance Type	c	#	sampling length in seconds	I*	II**	$\overline{M_c}$	$\tilde{M}_c$	σ_{M_c}	c_{RSD}	$\overline{c_{RSD}}$
bursting	A_b1m	18	1200	1	2	940.91	944.33	20.39	2.17	8.75
	A_b1s	17	1800	2	2	929.16	931.88	25.08	2.70	9.56
	A_b2	14	1800	3	5	1762.16	1765.39	72.68	4.12	14.79
baseline	A_nb1m	15	1500	7	12	95.62	94.57	6.25	6.54	15.42
	A_nb1s	16	1500	26	45	191.52	184.63	24.87	12.99	19.27
	A_nb2	13	900	3	4	393.06	393.31	15.00	3.82	13.64
general purpose	A_gp2_1	26	300	1	1	1247.37	1251.63	24.16	1.94	8.70
	A_gp2_2	26	300	1	1	1302.83	1306.67	24.74	1.90	9.00
	A_gp4	23	480	2	3	1939.74	1944.35	63.76	3.29	14.31
compute optimized	A_co2_1	29	300	2	2	1417.09	1424.81	36.16	2.55	11.00
	A_co2_2	26	300	1	2	1472.01	1474.56	35.33	2.40	10.45
	A_co4	19	480	2	3	2192.07	2200.71	70.80	3.23	11.40
bursting	G_b1	13	420	8	13	321.73	319.58	22.29	6.93	23.05
general purpose	G_gp1	26	420	2	3	722.21	721.31	22.68	3.14	14.33
	G_gp2	26	420	2	4	1102.49	1098.69	38.66	3.51	16.67
	G_gp4	24	420	3	5	1888.37	1888.63	77.19	4.09	18.85
compute optimized	G_co2	18	420	2	4	1095.28	1096.91	38.23	3.49	13.38
	G_co4	23	420	4	7	1791.98	1783.20	87.32	4.87	21.95

* 95 % CL (5% MOE) ** = 99 % CL (5% MOE)

Table 6.4: (c_{RSD}) and $(\overline{c_{RSD}})$ for each configuration

Leitner and Cito [LC16] propose the relative stand deviation c_{RSD} (defined in equation 6.4) to compare the performance of instance exhibiting the same configuration. A variation of 5% and more is considered to be a relevant.

The performance of instances exhibiting the same configuration varies only for instances of bursting instance types relevantly. At Amazon EC2, these are instances of configuration A_nb1s performing at baseline performance, with a relevant variation of 12.99% as well as instances of configuration A_nb1m performing at baseline performance with a variation of 6.54%. At GCE, the only bursting instance type is covered by configuration G_b1. G_b1 and varies relevantly by 6.93%. Table 6.4 summarizes the results.

If we compare the results with the results reported by Leitner and Cito [LC16], we observe a clear pattern. In Amazon EC2's Frankfurt and GCE's europe-west1 regions, only bursting instance types show a relevant performance variability between instances of the same configuration.

Figures 6.5 and 6.6 show both providers in comparison. As can be seen from these figures, are instances from Amazon EC2 in general more predictable than instances from GCE. This general

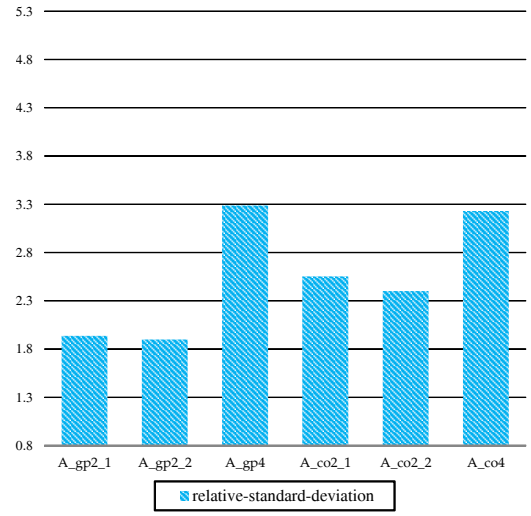


Figure 6.5

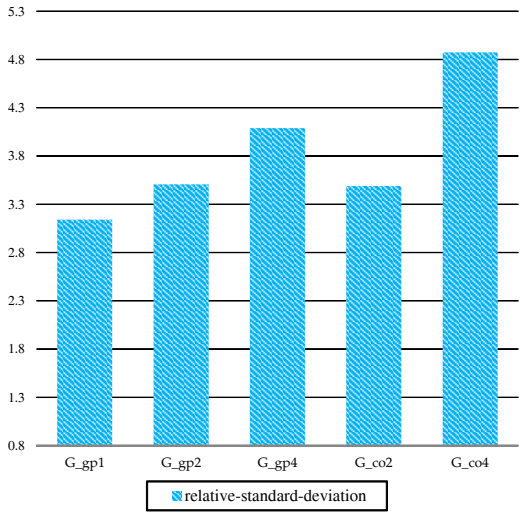


Figure 6.6

observation is also in line with the results of Leitner and Cito [LC16].

Therefore, based on our analysis, we conclude that general purpose and compute optimized instances are predictable, at least in this regions.

Figure 6.7 shows the performance of the compute optimized instance type with 2 cores. As can be seen from the figure, have all instance more or less the same $SRPS$ i.e., all lines are clustered. In contrast to this figure, Figure 6.8 shows some examples of the performance of bursting instances at GCE. Clearly visible, the instances are not as predictable as in the case of non-bursting instance types, i.e., the shown lines are not as clustered as for the non-bursting instance types in Figure 6.7.

Analogous for Amazon EC2: compute optimized instances with 2 cores are shown in Figure 6.9 and some bursting instances in Figure 6.10

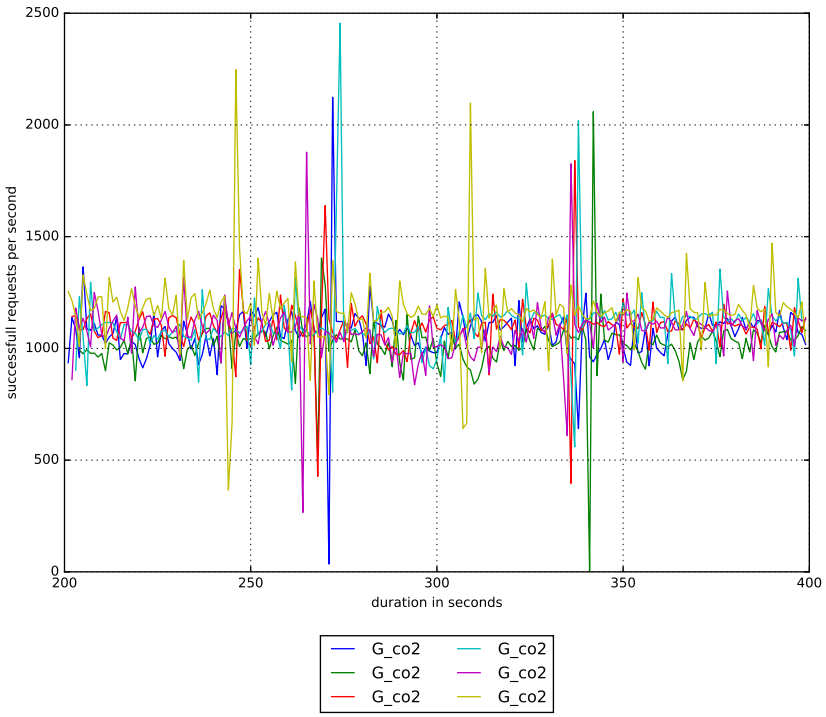


Figure 6.7: Predictability GCE

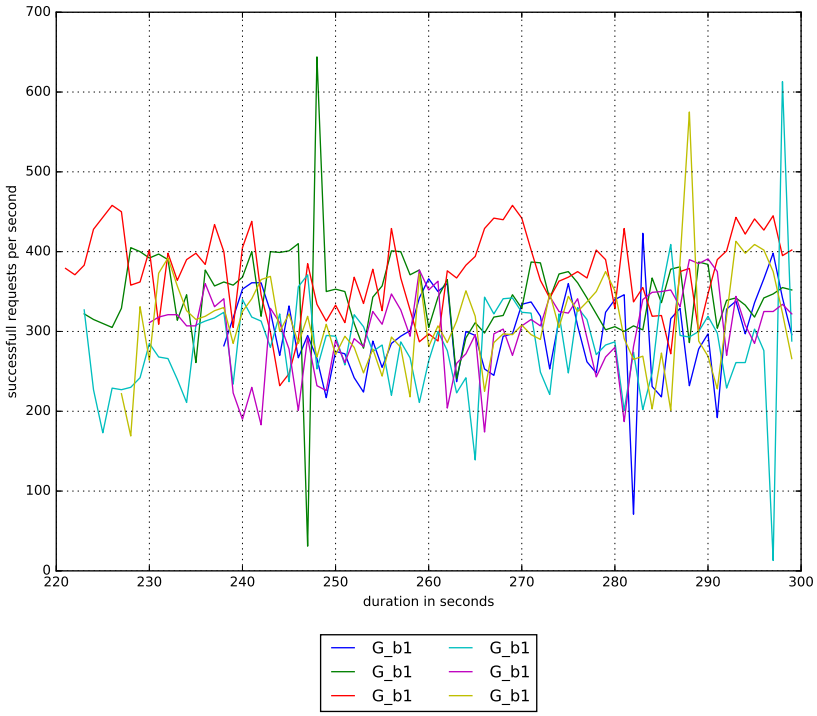


Figure 6.8: Predictability Bursting Instance Type GCE (baseline performance)

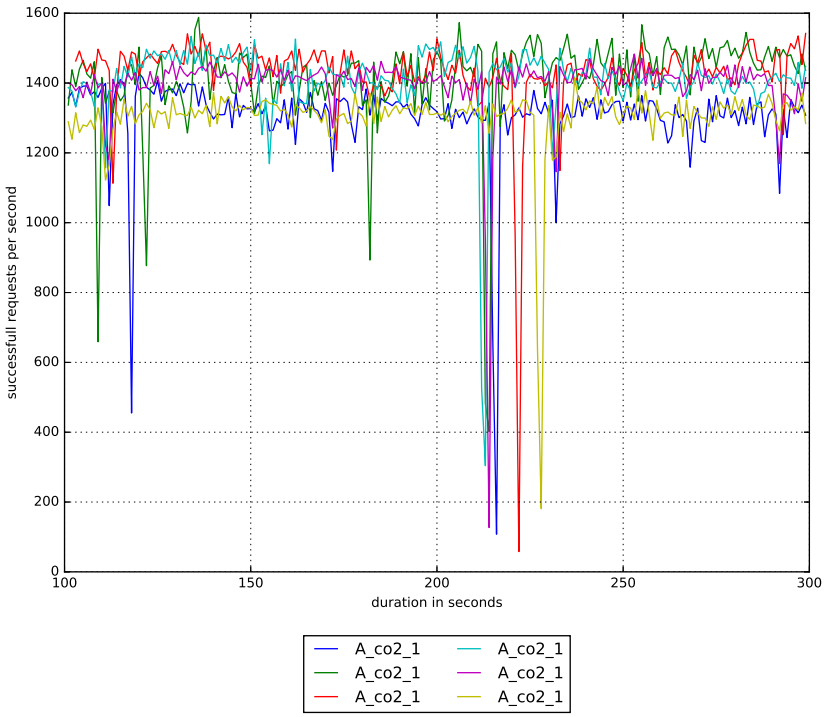


Figure 6.9: Predictability AWS

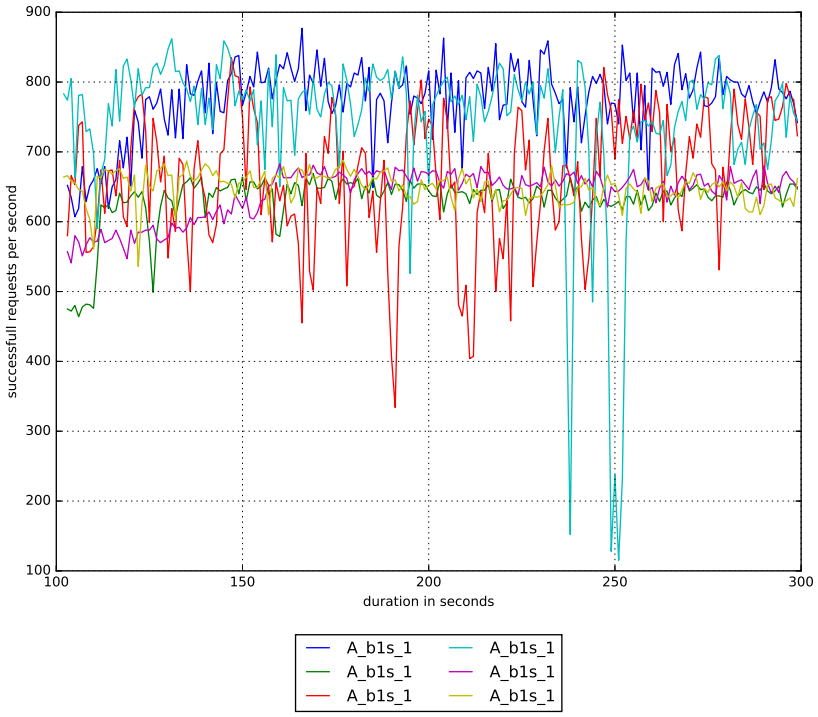


Figure 6.10: Predictability Bursting Instance Type AWS

6.2.2 Instance Performance Stability

In the context of intra-instance performance variability, we interpret the instance stability over time as the inverse of the intra-instance performance variability over time. Thus, the higher the instra-instance variability the lower the instance's performance stability. In this regard, Leitner and Cito [LC16] offer the following hypothesis: *H2.2: Intra-Instance Stability of CPU-Bound Applications – the performance of CPU-bound applications tends not to vary relevantly within the same instance for instances with a dedicated CPU [LC16].*

In order to calculate the performance variability of a single instance, we again use the approach provided by Leitner and Cito [LC16]. As a measure of performance variability, we use again the relative standard deviation c_{RSD}, which was already defined in Equation 6.4, this time of the variability within a single cloud instance, as follows:

$$\forall c \in C \forall i \in I_c : i_{RSD} = 100 \cdot \frac{\sigma_{m(i)}}{\overline{m(i)}} \tag{6.5}$$

According to Leitner and Cito [LC16], the individual performance variabilities can be summarized as the mean relative standard deviation ($\overline{c_{RSD}}$), namely as the arithmetic mean of all i_{RSD} values for a given configuration.

All resulting mean relative standard deviations ($\overline{c_{RSD}}$) of all configurations are shown in Table 6.4. Note: in Table 6.4 the ($\overline{c_{RSD}}$) is calculated for each configuration with an individual sampling durations.

The results revealed in Table 6.4 can not confirm this hypothesis. In contrast to Leitner and Cito, which observed a very low relative standard deviation for the same cloud instance for all CPU-bound benchmarks on non-bursting instance types in all providers, in our study, all instances of all configurations show for all providers a relevant intra-instance variability and are therefore not considered to be stable.

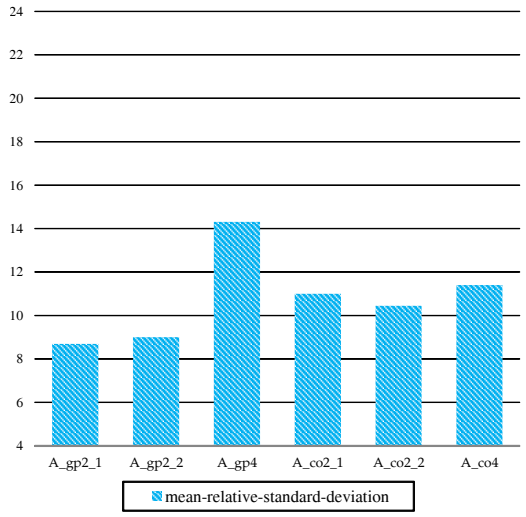


Figure 6.11: Visualization ($\overline{c_{RSD}}$) Amazon EC2

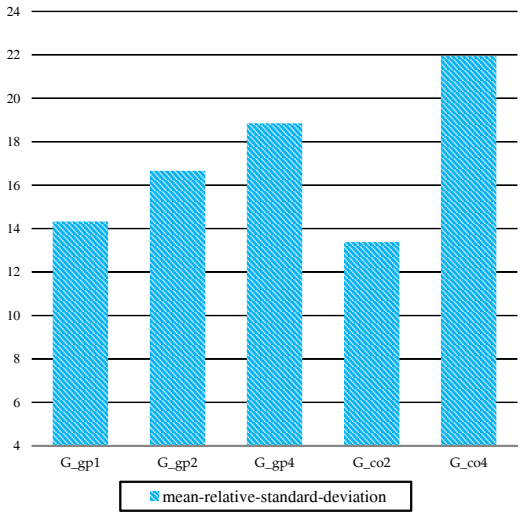


Figure 6.12: Visualization ($\overline{c_{RSD}}$) GCE

Figures 6.11 and 6.12 visualize the results. From comparing the two figures to each other, it is obvious that Amazon EC2 provides in general the better performance stability than GCE2. This is also contrary to the results Leitner and Cito [LC16] offer in their work.

Our experiments differ in two major points from the experiments conducted by Leitner and Cito [LC16]. Firstly, our application is composed of two distinct instances, whereas Leitner and Cito run a CPU microbenchmark on a single instance. Secondly, do Leitner and Cito [LC16] not provide any information about the resource consumption of their application benchmark and how long such a CPU microbenchmark had to complete. The analysed data-set of our application benchmark executions, is depending on the concrete configuration, between 300 seconds and 1800 seconds long. Figures 6.13 and 6.14 show the fluctuating performance for all non-bursting instance types at Amazon EC2 as well as GCE.

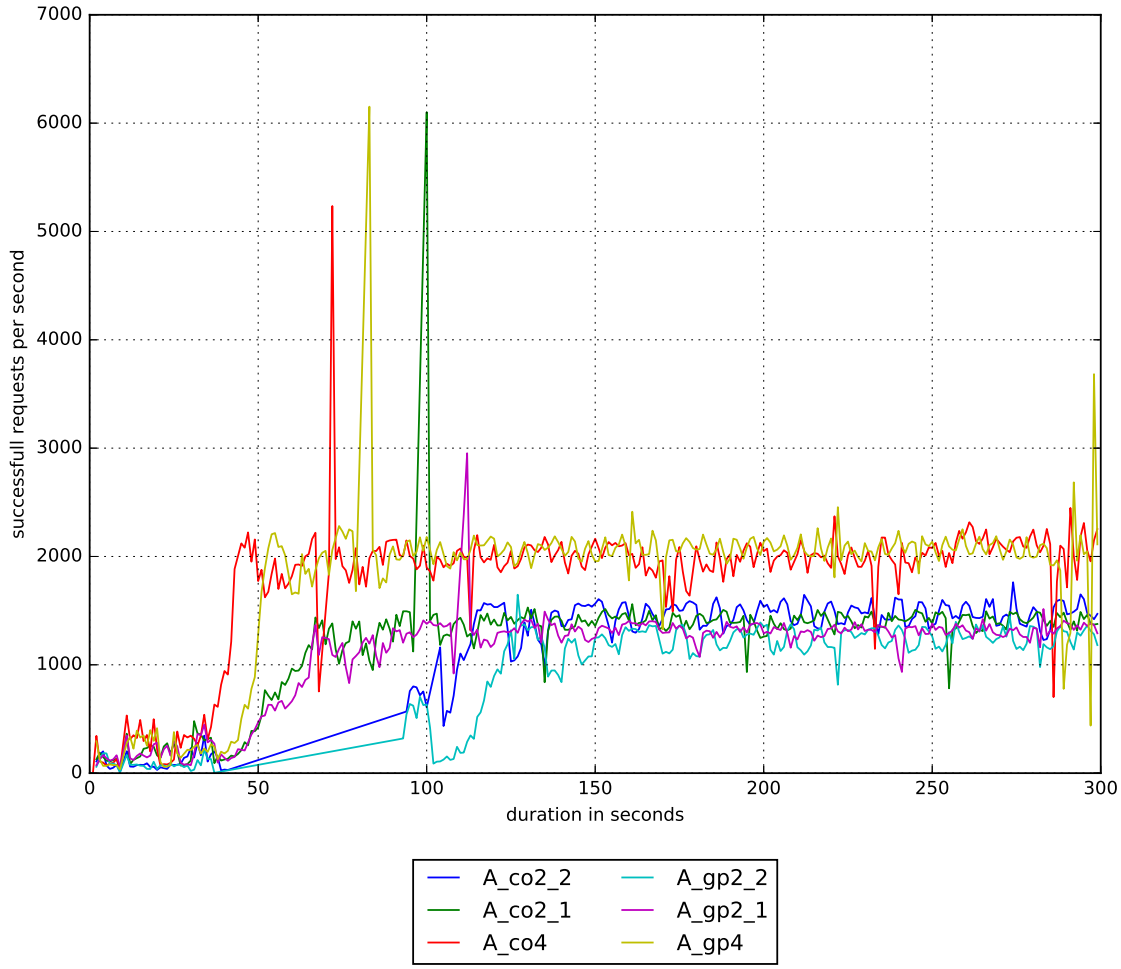


Figure 6.13: Stability of instances at Amazon EC2

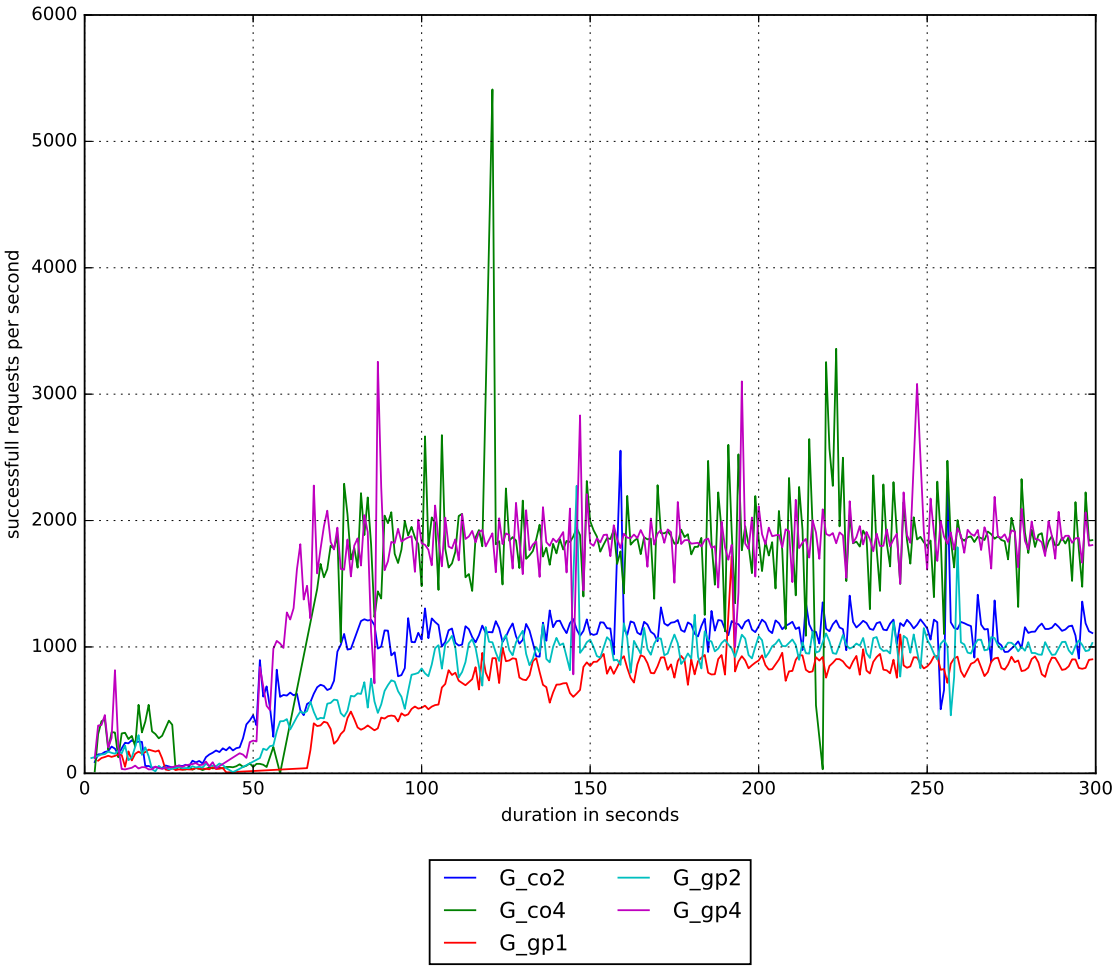


Figure 6.14: Stability of configurations at GCE

6.2.3 Stability of Larger Instance Sizes

H4.2: Stability of Larger Instance Types – the predictability of performance for any application tends to increase with increasing instance type costs [LC16].

To approach this hypothesis, and to answer if instances of a larger size perform more stable than smaller ones, we reconsider Table 6.4. Table 6.4 is ordered by instance size and the stability of the individual configurations are indicated in the row labelled $\overline{c_{RSD}}$. Figures 6.11 and 6.12 are also ordered by instance type first and instance size later. From the figures can be seen, that both overall and for each instance type larger instances are less stable.

The data collected in this study shows, that the hypothesis cannot be supported, in contrary, if we neglect the bursting instance types performing at baseline performance, we can generally state that smaller instances sizes perform more stable than larger instance sizes. Leitner and Cito [LC16] support this hypothesis in EC2 and GCE. Since our study did not primarily focus on the intra-instance stability, we would like to postpone a definitive remark on this issue to future work.

A fact which is not covered by the hypothesis is, that smaller instance types perform more predictable, i.e., show a smaller intra-configuration variability compared to larger instance sizes (i.e., more expensive instance sizes).

6.2.4 Baseline vs. Bursting Performance Stability

Bursting instance types are able to boost performance if a peak load has to be processed. If there is no load or only little load, they perform on a base level. Cloud providers such as Amazon EC2 apply a credit based system to decide, which instance is allowed to boost its performance, i.e., which instance can utilize a full CPU core. At Amazon EC2, for each minute the core utilization is above a certain threshold, a credit is consumed [Ama16a]. In order to measure the $\overline{SRPS}$, we generated a workload sufficient to require the instance to burst. After a certain burst duration, the instance runs out of credits and is throttled stepwise to baseline performance. This process is depicted in Figure 6.15

Subsequently, we measured the performance of the instance during their bursting period, and the consecutive baseline performance period. The resulting data is depicted in Table 6.6, where $\overline{SRPS}$ is the mean of the successful requests per second and $\tilde{SRPS}$ is the median of the same, both taken from Table 6.4.

Further details regarding the different execution times of the benchmarks are listed in Table 6.1. Table 6.4 summarizes the results in the rows *bursting performance* and *baseline performance*. For the evaluation of the bursting and baseline performance variability, we considered the same set of instances. While for the evaluation of the bursting performance all instances in the set could be used, we had to exclude some instances from the evaluation of the baseline performance. This, since they did not run out of credits and thus did not perform on baseline performance after the execution run was stopped.

The concrete hypothesis of Leitner and Cito [LC16] regarding the intra-instance variability of bursting instance types is *H2.3: Intra-Instance Variability of Bursting Instance Types – the performance of any application using a bursting instance type tends to vary relevantly within the same instance [LC16].*

As already explained in the last section, varies the performance of all instances of all configurations relevantly over time. In contrast to the common expectation, does the bursting performance of Amazon EC2's bursting instance types vary *less* than the performance of most non bursting instances. A possible explanation could be, that a busting instance type at bursting performance utilizes the full CPU core. This could mean that possible tenants are excluded from the utilization and hence the "noisy neighbour" problem is mitigated.

A visual inspection of the $\overline{SRPS}$ evolution over time reveals, why the variability of the base performance is fluctuating: Since bursting instances receive a certain amount of credits per hour, the instance starts, as soon as a new credit is available, to burst. This results in an approximately binary performance behaviour characterized by a temporary stable performance but a variable performance over time. The behaviour can be seen in Figure 6.15.

Leitner and Cito [LC16] support the hypothesis only for Amazon EC2. This, since their results show only a minor performance variability and hence a relatively stable performance for all GCE intance types, including the bursting one.

Amazon provides in their user documentation [Ama16e] some information about the baseline and the bursting performance. The values for each instance size are depicted in Table 6.5. By comparing these values with the values from Table 6.6, we noticed that they fit for t2.micro and t2.small instances, but diverge considerably for t2.medium instances. Amazon EC2 claims a baseline performance of 40% of the bursting performance. Our experiments revealed a baseline performance of only about 22% of the bursting performance. Further research in this issue is therefore required.

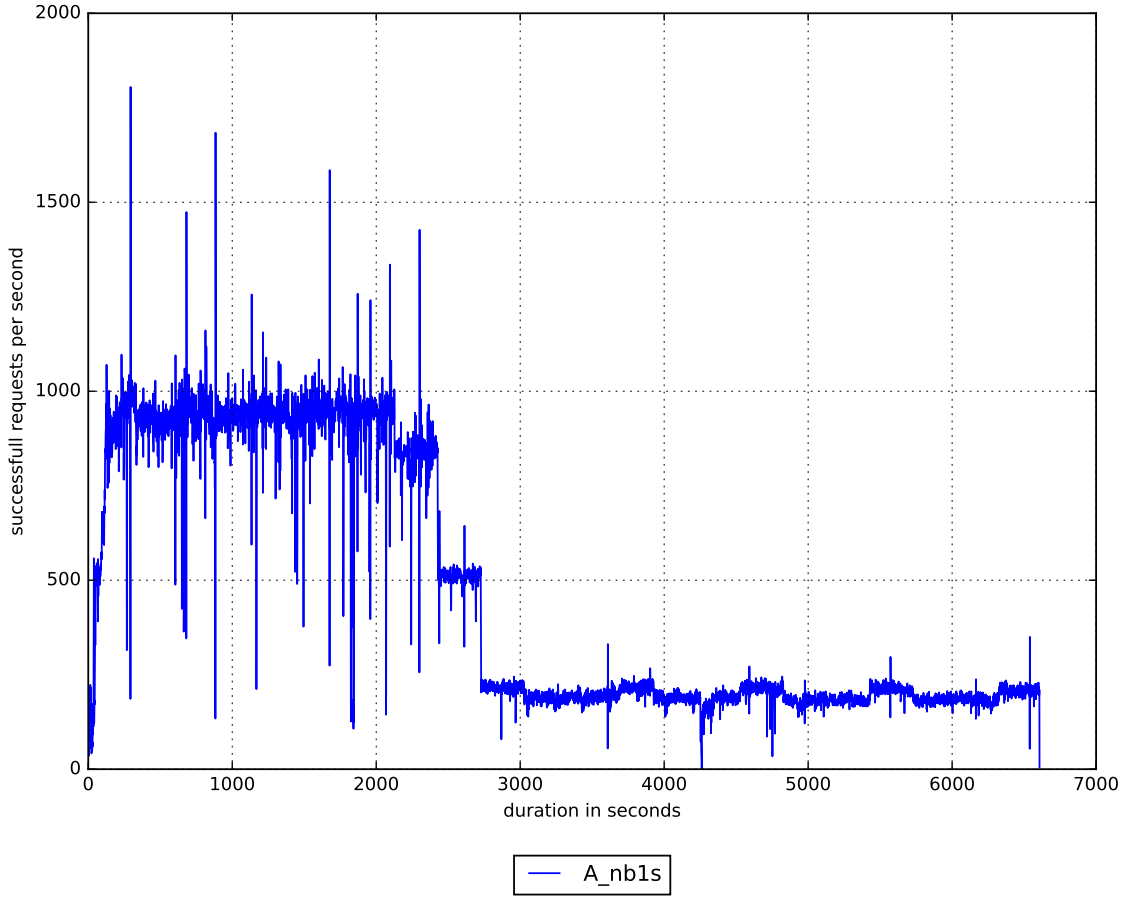


Figure 6.15: Bursting EC2 instance $\overline{SRPS}$ evolution

Instance type	Initial CPU credit	CPU credits earned per hour	Base performance (CPU utilization)
t2.micro	30	6	10%
t2.small	30	12	20%
t2.medium	60	24	40%

Table 6.5: Performance ratio according to Amazon EC2 [Ama16e]

c_1	$\overline{SRPS_{c_1}}$	$S\tilde{R}PS_{c_1}$	c_2	$\overline{SRPS_{c_2}}$	$S\tilde{R}PS_{c_2}$	$\frac{\overline{SRPS_{c_2}}}{\overline{SRPS_{c_1}}}$	$\frac{S\tilde{R}PS_{c_2}}{S\tilde{R}PS_{c_1}}$
A_b1m	940.91	944.33	A_nb1m	95.62	94.57	0.10	0.10
A_b1s	929.16	931.88	A_nb1s	191.52	184.63	0.21	0.20
A_b2	1762.16	1765.39	A_nb2	393.06	393.31	0.22	0.22

Table 6.6: Performance ratio baseline vs. bursting performance

6.2.5 Diseconomies of Scale

In this section, to allow a comparison of bursting and non-bursting instance types, we first try to find an appropriate way to estimate the performance of the bursting instance types for a whole hour. After that, we address if there are diseconomies of scale for larger instance sizes.

Theoretical Mixed Instance Type

In order to be able to compare bursting instance types to non-bursting instance types, we propose a mixed instance type. To this end, we take into account the amount of credits (ϵ) a certain instance size receives per hour. At Amazon EC2, one credit corresponds to full CPU utilization for a whole minute. Consequently, the instance can run $\frac{\epsilon}{60}$ of an hour on bursting performance. Moreover, if we allege that the instance does not receive any credits at start-up and the performance behaves in a binary mode, such that an instance always performs either on baseline or bursting performance, the performance $\overline{M_c}$ of the mixed instance type can then be calculated as follows:

$$\forall c \in C : \overline{M_c} = \frac{\epsilon}{60} \cdot \overline{M_{c_h}} + \left(1 - \frac{\epsilon}{60}\right) \cdot \overline{M_{c_l}} \tag{6.6}$$

Where

$\overline{M_{c_l}}$ is in our case the $\overline{SRPS}$ at baseline performance

$\overline{M_{c_h}}$ is in our case the $\overline{SRPS}$ at bursting performance

ϵ it the credits the web application instance types gets per hour

Based on this equation, we can compute for each bursting instance configuration a mixed instance configuration. A_b1m : $\epsilon = 6$, A_b1s : $\epsilon = 12$, A_b2 : $\epsilon = 24$. For the bursting instances in this work, the resulting mixed instance types are depicted in Table 6.7.

Performance Type	c	$\overline{M_c}$	$\frac{p_c}{hour}$
bursting	A_b1m	940.91	\$0.015
	A_b1s	929.16	\$0.030
	A_b2	1762.16	\$0.060
baseline	A_nb1m	95.62	\$0.015
	A_nb1s	191.52	\$0.030
	A_nb2	393.06	\$0.060
mixed	A_x1m	180.15	\$0.015
	A_x1s	339.05	\$0.030
	A_x2	940.70	\$0.060

Table 6.7: Mixed instance types for Amazon EC2

Diseconomies of Scale of Larger Instance Sizes

H4.1: Diseconomies of Scale of Larger Instance Types – the ratio of performance and costs for any application tends to decrease with increasing instance type costs [LC16].

H4.1 claims that larger (i.e., more expensive) instance sizes are generally less cost-efficient than smaller instance sizes [LC16].

Leitner and Cito [LC16] used in their work the normalized cost-performance ratio. To this end, they normalize the mean of the measurements for a configuration $c \in C$ to the arithmetic mean of the measurements for configuration $n \in C$, i.e.,

$$\forall c \in C : \overline{M_c^*} = \frac{\overline{M_c}}{\overline{M_c}(n)} \tag{6.7}$$

Where

$\overline{M_c}$ is the mean of the measurements for configuration $c \in C$

$\overline{M_c}(n)$ is the mean of the measurements for configuration $n \in C$

Analogously, they normalize all hourly prices to the price of the configuration n of the same provider.

$$\forall c \in C : p_c^* = \frac{p_c}{p_n} \tag{6.8}$$

Where

p_c is the hourly price of configuration $c \in C$

p_n is the hourly price for configuration $n \in C$

The normalized cost-performance ratio cpr can then be written as:

$$\forall c \in C : cpr_c^* = \frac{\overline{M_c^*}}{p_c^*} \tag{6.9}$$

For our work, we use for Amazon EC2 $n =$ A_gp2_1 and for GCE $n =$ G_gp1. The resulting values for cpr_c^* are depicted in tables 6.8 and 6.9.

While we can confirm this general statement for Amazon EC2, we cannot for GCE. Figures 6.16 and 6.17 visualize the cost-performance ratio cpr_c^* for each configuration based on the data from Tables 6.8 and 6.9. The different configurations are depicted in ascending ordered by the price per hour. The results for Amazon EC2 (Figure 6.16) show apart from the mixed instance type (derived from the t2.medium instance size; developed in Section 6.2.5), a steadily falling cost-performance ratio. Hence, the hypothesis holds for Amazon EC2.

Although GCE (Figure 6.17) shows a similar trend, the general purpose instance with 2 cores has a worse performance ratio than the more expensive but better performing compute optimized instance with 4 cores. Hypothesis 4.1 can therefore not be supported in general. However, if we consider each instance family separately, the hypothesis is true for both, general purpose and compute optimized instances.

Leitner and Cito [LC16] do not confirm the hypothesis neither for EC2 nor for GCE, noting that according to their results, all small general-purpose instances provide the worst cost-performance ratio.

c	$\overline{M_c^*}$	p_c^*	cpr_c^*
A_x1m	0.13	0.11	1.14
A_x1s	0.24	0.22	1.07
A_x2	0.66	0.45	1.48
A_co2_1	1.00	1.00	1.00
A_co2_2	1.04	1.00	1.04
A_gp2_1	0.88	1.07	0.82
A_gp2_2	0.92	1.07	0.86
A_co4	1.55	1.99	0.78
A_gp4	1.37	2.13	0.64

Table 6.8: cpr_c^* for Amazon EC2

c	$\overline{M_c^*}$	p_c^*	cpr_c^*
G_b1	0.45	0.55	0.82
G_gp1	1.00	1.00	1.00
G_co2	1.52	1.53	0.99
G_gp2	1.53	2.00	0.76
G_co4	2.48	3.05	0.81
G_gp4	2.61	4.00	0.65

Table 6.9: cpr_c^* for GCE

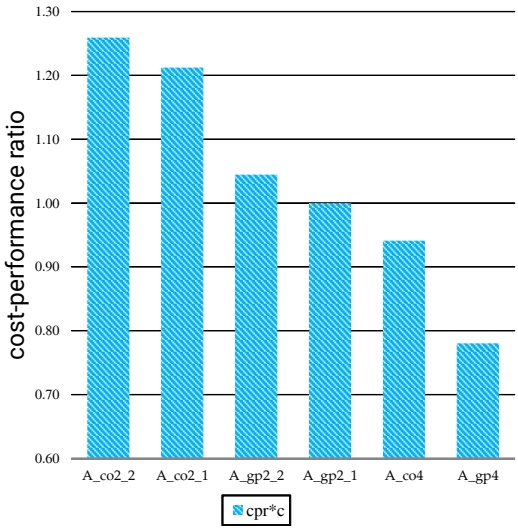


Figure 6.16: cpr_c^* trend for Amazon EC2

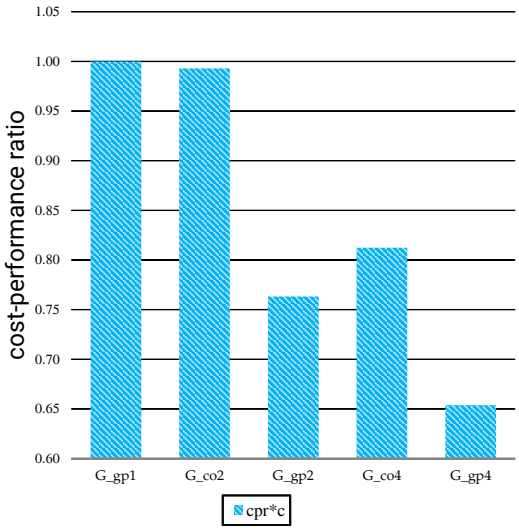


Figure 6.17: cpr_c^* trend for GCE

6.2.6 Thrift of Specialized Instances

The cost efficiency of specialized instances compared to general purpose instances for processing tasks related to their specialization is still an unanswered question. The related hypothesis constructed by Leitner and Cito [LC16] is *H4.3: Price of Specialization – specialized instance types tend to have a better ratio of performance and cost for applications related to their specialization, and worse ratio otherwise [LC16].*

To compare the price of the compute optimized instance types to the prices of the general purpose instance types, we again use the normalized cost-performance ratio cpr_c^* (Equation 6.9). This time, we use for n the configuration of the general purpose instance type we want to compare against. For considerations regarding stability and predictability, which may also have an impact on business decisions, we include also the values for the relative standard deviation c_{RSD} and the mean relative standard deviation $\overline{c_{RSD}}$ from Section 6.2.1 and 6.2.2.

In regard to this hypothesis, our study results can only reveal insights about the first part of the hypothesis, namely, if specialized instance types tend to have a better ratio of performance

c	$\overline{M_c}$	c_{RSD}	$\overline{c_{RSD}}$	$\frac{p_c}{hour}$	$\overline{M_c^*}$	p_c^*	cpr_c^*
A_co2_1	1417.09	2.55	11.00	\$0.134	1.14	0.94	1.21
A_gp2_1	1247.37	1.94	8.70	\$0.143	1.00	1.00	1.00
A_co2_2	1472.01	2.40	10.45	\$0.134	1.13	0.94	1.20
A_gp2_2	1302.83	1.90	9.00	\$0.143	1.00	1.00	1.00
A_co4	2192.07	3.23	11.40	\$0.267	1.13	\$0.94	1.21
A_gp4	1939.74	3.29	14.31	\$0.285	1.00	1.00	1.00

Table 6.10: Price of Specialization at Amazon EC2

c	$\overline{M_c}$	c_{RSD}	$\overline{c_{RSD}}$	$\frac{p_c}{hour}$	$\overline{M_c^*}$	p_c^*	cpr_c^*
G_co2	1095.28	3.49	13.38	\$0.084	0.99	0.76	1.30
G_gp2	1102.49	3.51	16.67	\$0.110	1.00	1.00	1.00
G_co4	1791.98	4.87	21.95	\$0.168	0.95	0.76	1.24
G_gp4	1888.37	4.09	18.85	\$0.220	1.00	1.00	1.00

Table 6.11: Price of Specialization at GCE

and cost for applications related to their specialization, and this only for computation optimized instance types.

For Amazon EC2, Table 6.10 indicates, that compute optimized instance types provide a 20% cost-performance bonus compared to general purpose instance types with the same number of CPU cores. These findings are in line with the findings of Leitner and Cito [LC16].

At GCE, the compute optimized instance types provide an even higher bonus, namely 30% for instances equipped with two vCPU cores and 24% for instances with four vCPUs (Table 6.11).

While the bonus in cpr_c^* of compute optimized instances at Amazon EC2 is achieved through a better performance, resulting from better hardware in combination with a slightly lower price (6% lower), the cpr_c^* bonus in GCE is realised by a 24% lower price. In GCE, compute optimized instances are not "really" optimized and use the same hardware as general purpose instances, but are due to the fewer allocated memory more cost efficient.

Chapter 7

Performance-Cost Index

All results above were discussed for each cloud provider individually, although research is aware of the struggle of practitioners to find a relevant cloud provider for their custom application. In order to support application benchmarking, we propose a generic application-benchmark which can be used to determine the performance of a custom application, across a variety of cloud providers, and in an efficient and repeatable manner. However, collecting the data is only the first step towards a business relevant decision. In order to reach an informed decision regarding cloud provider choice, practitioners need additionally an approach to compare heterogeneous offerings of different cloud providers in an objective manner. In the following section, we apply a performance-cost normalization (PCN), to show, which cloud service tested in our study offers the most "bang for the buck". But not all applications are the same. We are further interested in a general comparison of cloud service's performance, independent of a benchmark. Hence, we introduce another normalization strategy from the educational context, namely, the (exam) score normalization which is common to ensure a fair grading practice. Score-normalization allows to average the PCN-scores across several benchmarks and thus allows to establish an benchmark independent performance-cost-index.

7.1 Performance-Cost Normalization

In order to compare the "bang for the buck" of one entity to another, Vedam and Vemulapati propose in [VV12] to use *performance-cost normalization (PCN)*. The basic idea of PCN is to determine the cost of running an application in the cloud (i.e., tracking the total cost for the cloud deployment), and then to normalize the performance with the costs.

The cost of executing any task on a cloud service is according to Vedam and Vemulpati [VV12] calculated by summing up all incurring costs (for any kind of cloud resources), over a specified period (Equation 7.1). Note, in Equation 7.1 we adopt the original labels from the authors, although we have used them already in the chapters before.

$$C = (c_i \cdot T) + (c_s \cdot S \cdot T) + (c_b \cdot D) \tag{7.1}$$

Where

$C =$ cost total per hour

$c_i =$ cost of the cloud instance per hour

$c_s =$ cost of the storage per Megabits per hour

$c_d =$ cost of the data transfer per Kilobits

$T =$ Time taken for the task to run in hours

$S =$ Storage consumed in Megabits

$D =$ Data transferred in Kilobits

The performance parameter P can then be divided with the total cost C in order to get the normalized value PCN, i.e.,

$$PCN = \frac{P}{C}$$

Once we have the normalized values of PCN for all cloud offerings in question, we can simply compare the resulting values [VV12].

Although PCN is sufficient to compare the performance mathematically, a more intuitive metric to select between instance types is *performance per $ per hour* [SSS+08]. If P is measured in seconds and prices are in $, we can simply multiply PCN with 3600 in order to get this more meaningful measure.

7.2 Score Normalization

In teaching, course instructors want to evaluate students in a manner that is based upon the student's representative performance [Win02]. While fair grading is a simple task in a single assignment, it becomes a more difficult matter when multiple assignments have to be considered, and even more demanding when the assignment with the poorest performance has to be dropped [Win02]. In order allow a fair comparison of individual assignments, the scores need to be converted into a common currency. *"Ideally, we would like the distribution of individual student performance for all exams to be equal, despite differences in time, instructor, teaching assistant, and other factors. Only then can evaluations be considered comparable." [Win02].*

To achieve this, Cross [Cro95] proposes *score normalization (SN)*. In score normalization, an individual score is transformed into a context-free evaluation of relative performance. Concretely, an individual score is converted into a standard score. A standard score is then the number of standard deviations the absolute-score or percentage-score is above or below the average, and is called z-score [Cro95]. Its equation is:

$$z = \frac{X - \mu}{\sigma} \tag{7.2}$$

Where:

$z :=$ z-score

$X :=$ observed score

$\sigma :=$ standard deviation in the assignment

$\mu :=$ the average score in the assignment

By transposing the scores from different assignments onto a common scale, all scores are converted to the same underlying distribution, namely the distribution with $\mu = 0$ and $\sigma = 1$ and thus a reasonable comparison is rendered possible [Win02]. Although the z-score per se would already suffice to compare the individual assignments, Cross [Cro95] proposes (for convenience reasons) to transpose the z-score to a T-score. A T-score is a score with an dictated underlying distribution i.e,

$$T = \mu + \sigma \cdot z \tag{7.3}$$

For a calculation example we refer to Winters [Win02].

To get a composite score from which the influence of the variability of the scores has been eliminated, T-scores can according to Cross [Cro95] simply be averaged. Note, if more than two scores are to be averaged, the resulting composite score is only an approximation since the intercorrelations among the scores should be considered too. Nevertheless, Cross [Cro95] points out that also averaging of more than two scores should produce a good approximation of the precise result [Cro95].

Another important feature of T-scores is, that they, such as absolute scores for instance, may be weighted differentially. If one out of two assignments should be weighted double, we can simply multiply its T-score by two and divide the sum of the scores by three [Cro95].

7.3 Comparable Benchmark Score (CBS)

In the last paragraph, SN was introduced in the context of teaching. Nevertheless can SN also be applied to enable an objective comparison of cloud services, namely to compare the performance-scores a specific cloud service scored in different cloud assignments (i.e., benchmarks, of course). In contrast to the geometric mean, which can be used to compare the performance of two instances across several benchmarks [FW86], SN can be used to objectively compare the individual performance a single instance delivered in each benchmark, for example to determine which is the core-competency of the instance.

In order to compare different benchmark-scores with each other, we first apply PCN to the performance results obtained from each benchmark, and then apply SN to receive a normalized and thus benchmark independent score. We coin the term *comparable benchmark score (CBS)* to refer to it. The cbs is defined analogous to SN's T-score (Equation 7.3), i.e.,

$$\forall c \in C : cbs_c = M + S \cdot \frac{PCN_c - \overline{PCN}}{\sigma_{PCN}} \tag{7.4}$$

Where

cbs_c is the Comparable Benchmark Score for configuration $c \in C$

PCN_c is the PCN-value for $c \in C$

PCN is the set of PCN-values after PCN (Equation 7.1)

$\overline{PCN}$ is the arithmetic mean of all PCN-values for the benchmark

σ_{PCN} is the standard deviation of all PCN-values for the benchmark

M is the new mean of the underlying distribution

S is the new standard deviation for the underlying distribution

To show that the normalization process has no effect on the ranking, we added a comparison to the appendix (Section 9.5).

7.4 Lot-size Problem

As Binnig et al. [BKKL09] point out, do cloud providers charge the services based on a certain lot-size, namely a fixed minimum run time. For example at Amazon EC2, instances are billed in "instance hours". At GCE, the smallest billing unit is a single minute, while 10 minutes are billed also if they are not used (Equations are given in 2.1 and 2.2).

While it is important to take the lot-size into account for determining what instance size and type has to be chosen to get a certain amount of work done, it can be neglected for a general comparison of the performance of single instances. If a defined amount of work has to be done, it is very likely that an instance will not use up a whole lot. Therefore, slightly more expensive instances with same performance but a better lot-size may be more cost efficient.

7.5 Comparing the Overall cbs Across Benchmarks

In order to compare the performance of multiple instance sizes over several benchmarks, we have to be thoughtful. Although the cbs-score represent the same ordering, as the real scores do, they must not be averaged with the arithmetic mean, as Fleming and Wallace elucidate and proof in their work [FW86]. The problem is the normalization, which skews the relations. In order to compare different instance competing in several benchmarks to each other, Fleming and Wallace [FW86] propose the geometric mean. The geometric mean has the advantage, that in contrast to the arithmetic mean, it can also be applied to normalized values, such as the cbs-score. Analogous to the arithmetic mean, the geometric mean also allows to weight the different benchmarks individually.

7.6 Performance-Cost Index

To establish a performance-cost index, we calculated the cbs for all benchmarked configurations with $M = 0$ and $S = 1$ ($= z$-score). Moreover, we also calculated the successful requests (sr) per dollar count for each configuration ($\frac{sr}{\$}$). By assigning each configuration a rank and ordering the cbs-ranks in ascending order, we see immediately that both rankings are identical 7.1. While $\frac{sr}{\$}$ is benchmark specific, the cbs provides the same information but is benchmark independent. Table 7.1 depicts the performance-cost-index for all 20 configurations.

Note: the cost index is based on the instance prices per hour, and on the eu-central-1 region for Amazon EC2 and the Europe prices for GCE. Also performance was measured only in these regions.

c	$\overline{M_c}$	$\frac{P_c}{hour}$	$\frac{\text{mio. sr}}{\$}$	cbs_c	$\frac{\text{mio. sr}}{\$}$-rank	cbs_c-rank
A_x2	940.70	\$0.060	56.442	2.317	1	1
G_gp1	722.21	\$0.055	47.272	1.146	2	2
G_co2	1095.28	\$0.084	46.940	1.104	3	3
A_x1m	180.15	\$0.015	43.235	0.631	4	4
A_x1s	339.05	\$0.030	40.686	0.305	5	5
A_co2_2	1472.01	\$0.134	39.546	0.160	6	6
G_b1	321.73	\$0.030	38.608	0.040	7	7
G_co4	1791.98	\$0.168	38.399	0.013	8	8
A_co2_1	1417.09	\$0.134	38.071	-0.029	9	9
G_gp2	1102.49	\$0.110	36.081	-0.283	10	10
A_gp2_2	1302.83	\$0.143	32.799	-0.702	11	11
A_gp2_1	1247.37	\$0.143	31.402	-0.880	12	12
G_gp4	1888.37	\$0.220	30.901	-0.944	13	13
A_co4	2192.07	\$0.267	29.556	-1.116	14	14
A_gp4	1939.74	\$0.285	24.502	-1.761	15	15

Table 7.1: Cost Performance Index

As indicated in Table 7.1, are GCE's G_gp1 and G_co2 configurations, i.e., n1-standard-1 and n1-highcpu-2 instance sizes most economical. Amazon EC2's most economical configuration was A_co2_1, i.e., the c4.large instance size, which is on rank 3.

Tables 7.2 and 7.3 show the rankings split by provider. This time, also the overall ranking is indicated for reference.

c	$\overline{M_c}$	$\frac{P_c}{hour}$	$\frac{\text{mio. sr}}{\$}$	cbs_c	cbs_c-rank provider	cbs_c-rank overall
A_x2	940.70	$0.060	56.442	2.317	1	1
A_x1m	180.15	$0.015	43.235	0.631	2	4
A_x1s	339.05	$0.030	40.686	0.305	3	5
A_co2_2	1472.01	$0.134	39.546	0.160	4	6
A_co2_1	1417.09	$0.134	38.071	-0.029	5	9
A_gp2_2	1302.83	$0.143	32.799	-0.702	6	11
A_gp2_1	1247.37	$0.143	31.402	-0.880	7	12
A_co4	2192.07	$0.267	29.556	-1.116	8	14
A_gp4	1939.74	$0.285	24.502	-1.761	9	15

Table 7.2: Index for Amazon EC2

c	$\overline{M_c}$	$\frac{P_c}{hour}$	$\frac{\text{mio. sr}}{\$}$	cbs_c	cbs_c-rank provider	cbs_c-rank overall
G_gp1	722.21	$0.055	47.272	1.146	1	2
G_co2	1095.28	$0.084	46.940	1.104	2	3
G_b1	321.73	$0.030	38.608	0.040	3	7
G_co4	1791.98	$0.168	38.399	0.013	4	8
G_gp2	1102.49	$0.110	36.081	-0.283	5	10
G_gp4	1888.37	$0.220	30.901	-0.944	6	13

Table 7.3: Index for GCE

Chapter 8

Threats to Validity

As with any empirical research, there are threats and limitations to this study, which we would like to outline in the following.

8.1 Construct Validity

During the development of the generic application benchmark as well as during the setup of the experiments, design decisions had to be made. Most importantly, the choice of an application benchmark to assess the performance of the individual instances. Our application benchmark measures the performance of an individual VM instance by inferring it from the performance of the benchmark web application. This bears two major threats. Firstly, the web application could have been the performance bottleneck and thus the performance for larger instance sizes be underrated. We tried to mitigate this threat by tracking resource consumption with vmstat and VisualVm. Moreover, we tuned the Java heap space as good as possible. Secondy, the performance of the database could have influenced the performance of the web application and thus the performance of the instance hosting the web application in consequence be underrated. To mitigate this risk, we deliberately benchmarked some instance configuration which only vary in the size of the instance hosting the database. We compared the means of these configurations but could not find any statistically significant performance deviation.

Another design decision is the choice of the benchmark metric (mean of the successful requests per second, $\overline{SRPS}$). We mitigated the risk to choose a problematic metric by choosing a metric, which is already used in practice. Moreover, we normalize the resulting data as described in Chapter 7.

Further design decisions were what benchmark application to choose and what workload to use. After a thorough evaluation of the requirements to a state-of-the-art cloud benchmark, we decided to use an application which fulfils these requirements and moreover has only recently been developed. Additionally, also the original workload is used with only minor modifications.

In our experiments, we only tested instances acquired with the on-demand pricing plan. While we cannot verify the equality of the performance of on-demand and other pricing-type instances, we trust the indications in the user-guides of the tested providers which indicate that there is the same underlying hardware used to run the instances.

For calculating the performance-cost ratio, we only use the hourly instance-prices, which are assumed to be the major component of the costs for a cloud application. Although we have researched the current body of knowledge, to our best knowledge there does not exist any study covering this specific topic.

To compare the performance of non-bursting instance types to bursting instance types, we

developed the construct of mixed instance types. A mixed-instance type is a theoretical instance type its performance is composed partially of bursting performance and partially of baseline performance. In practice, bursting instance types show a much higher bursting performance just after startup than in their later bursting phases. For our calculations we used the bursting performance just after start, which for our purpose is sufficient, but could be threat to the validity of results in future studies.

8.2 Internal validity

Although we collected a relevant amount of data to validate our research questions, the duration of the execution runs may not be sufficient to come to bullet-proof conclusions regarding the intra-instance performance variability, also referred to as instance stability. In order to minimize the effects of this threat, we identified the relevant part of the data by visual inspection instead of using our extraction algorithm. Thus the quality of the data is as good as possible, although to fully control this thread benchmarks with a longer sampling duration have to be run in a future study.

8.3 External Validity

An external threat to validity is, of course, the selection of cloud providers and related the selection of instance types, regions, and deployment options. However, our choice of providers included one market leaders (Amazon EC2) and one up-coming provider, namely GCE. Hence, we argue that for the purpose of this work, the coverage of cloud providers is sufficient. Nevertheless is the generalizability of the results further restricted by the chosen region. For Amazon EC2, the chosen region is *eu-central-1a* Frankfurt and for GCE, the zone is *europe-west1-b*.

Chapter 9

Closing Remarks

9.1 Conclusion

Practitioner all over the world are challenged by the task to select the most relevant cloud service for a given application. Due to a ever growing number of cloud offerings, performance variability and missing cloud benchmarks, the selection of the best matching cloud offering is still an elaborate endeavour and hence not straight forward. In order to support practitioners in their struggle, we set out to develop a cloud-native and broadly applicable application benchmark.

In this work, we lay the foundation for cross-provider and application-aware benchmarking of IaaS cloud services. We propose a generic application-benchmark which embraces the principles of infrastructure as code, levering automation and repeatability by design. The proposed benchmark treats the system under test as black-box, and thus can be used to benchmark any IaaS based application out of the box. Authentic workload is a first order citizen. Hence, the benchmark employs Apache JMeter as a load generator which not only provides flexible ways to record and replay real-life workload traces, but also supports cookies, caching and the parallel download of static files. The benchmark is cloud native. Based on Cloud Workbench, an automation framework which supports a vast selection of cloud providers, the benchmark allows to benchmark instances from several cloud providers with only minimal effort.

We used our own benchmark to benchmark two cloud providers, namely Amazon EC2 and Google Compute Engine (GCE). We conducted in total over 1000 benchmarks, thereof 372 benchmark runs we evaluated to answer our research questions, which are common questions regarding performance variability and cost of on-demand IaaS instances. We acquired instances in 20 different configurations, 14 configurations at Amazon EC2 and 6 at GCE. We are now going to answer our research questions based on the collected data.

Are there diseconomies of scale for larger instance sizes? Our results show, that at both providers, larger instance sizes have a worse performance-cost ratio than smaller instances. At Amazon EC2, there are diseconomies of scale over all instance families and types. At GCE, diseconomies of scale are restricted to instance types, where larger instances of the compute optimized instance type have the better cost performance ration than smaller instances of the general purpose type. Thus, the selection of smaller instance sizes is economically reasonable.

Is it economical to choose specialized instances for special tasks? In this regard, and as already predictable from the observations of diseconomies of scale at GCE, specialized instance types provide the better performance-cost ratio than performance wise comparable general purpose instance types. Concretely, at Amazon EC2 offer compute optimized instances up to 20% and

at GCE up to 30% more performance for the same costs. In consequence, we advice practitioners to choose specialized instance types for their special purpose.

Is the performance of an instance of a certain size predictable? At both Amazon EC2 and GCE it is in general possible to predict the performance of individual instances of non-bursting instances types. Contrary, it is not possible to predict the performance of bursting instance types. We consider a relevant performance variation to be 5%. At Amazon EC2, t2.small instances performing at baseline performance yield a relevant variation of 12.99% and also t2.micro instance performing at baseline performance vary by 6.54%. At GCE, g1.small instances yield a relevant performance variation of 6.93%. In general, our results indicate that instance sizes acquired at Amazon EC2 provide a more predictable performance than instances acquired at GCE.

Is the performance of a certain instance stable? Although we conducted a relevant amount of experiments, the chosen benchmark duration may not suffice to answer this question with certainty. In our experiments, we used rather short benchmark execution durations. From the data, the warm-up and the shut-down phase are cropped, resulting in sampling duration varying between 300 and 1800 seconds. In contrast to related research, our results indicate that all instances of all configurations show for all providers a relevant intra-instance variability and are therefore not considered to be stable. As a general pattern, we identified instance form Amazon EC2 to perform more stable than instances from GCE.

Are larger instances more stable than smaller ones? In contrast to related research, our results show that apart form some bursting instance sizes, smaller instance sizes yield a more stable performance over time than instances of a larger size. As already mentioned, did we in our experiments use a rather short benchmark execution durations, which may influence the generalizability of these results.

Additionally, we also researched a more general question, namely how the performance and cost relation of different instance types and different cloud providers can be objectively compared across multiple benchmarks. To this end, we propose a novel normalization strategy yielding comparable benchmark scores. Comparable benchmark scores indicate the relative performance-cost ratio of an instance size compared to the entirety of benchmarked instances sizes. In order to compare the performance across several instance sizes participating in several benchmarks, we propose the geometric mean over the comparable benchmark scores. Our results show, the following ranking: 1. GCE's n1-standard-1, closely followed by GCE's n1-highcpu-2 on rank 2. The best performing Amazon instance is on rank 3, namely c4.large but provides 20% less performance compared to the leaders.

In conclusion, summing up the answers and their implications, we would advice practitioners to always choose specialized instance types, while preferring smaller instance sizes over larger. This, only due to economical reasons, since larger instance types are not able to make up the higher costs with better stability or higher predictability. Our results indicate, that Amazon EC2 provides the better predictability and also more stable instances, nevertheless provides GCE the better performance-cost ration. In consequence, the question "Who provides most bang for the buck" can not be answered conclusively.

9.2 Future Work

We see different possibilities for future work: obviously, and as already described in the threats to validity, research has to re-evaluate the results regarding the stability of bursting and non-bursting instance types. We propose to re-use our experimental setup and to conduct a relevant amount of experiments, but with a much longer execution duration.

Also obvious and of course the logic next step is to repeat our experiments for a greater selection of cloud providers. We have in our study shown, that an additional cloud providers causes a minimal change in the existing CWB benchmark definition. The crucial point will be to setup the SUT at the new provider, since every provider applies another method to manage ip addresses. As possible consequence, we also see the need to split the AcmeAir cookbook into sub-cookbooks, which handle provider specific ip address resolutions but are wrapped by a wrapper cookbook setting up all other AcmeAir related software and configurations.

To compare the performance of non-bursting instance types to bursting instance types, we tried to estimate the performance of the non-bursting instance types by a mixed-instance type, which is a theoretical instance type composed partially of bursting performance and baseline performance. To increase the accuracy of this construct, the defined assumption of a binary performance distribution has to be overworked. In practice, bursting instance types show a much higher bursting performance just after startup than in their later live. For our calculations we used the burst performance just after start. This could be improved for future studies.

Amazon EC2 provides the "Elastic Compute Unit" to describe the performance of their instance sizes. Analogously, GCE uses the "Google Compute Units" for their instances. Future work could try to find a correlation between measured performance and these provider specific measurements, in order to infer the real performance of an instance size based on the provider measure.

Further did our experiments regarding bursting and baseline performance of Amazon EC2 bursting instance types reveal, that the actual delivered baseline performance for t2.medium instances (measured 20%) is not as specified in the user documentation ($\%40$). We can currently not provide any reasons, hence we propose to investigate this in future work.

We investigated common questions regarding cloud provider choice. Future work will also need to investigate the reasons for the observed results, stating not only the "that" as in this work, but also the "why".

Acronyms

AMI amazon machine image

CBS Comparable Benchmark Score

CWB Cloud Workbench

GCE Google Compute Engine

GUI graphical user interface

HPC high performance computing

IaaS Infrastructure as a Service

MSRPS mean successful requests per second

NIST The National Institute of Standards and Technology

OLTP online-transaction processing

PCR performance cost normalization

RSD relative standard deviation

SN score normalization

SRPS successful requests per second

SUT system under test

VM virtual vachine

VMs virtual machines

Appendix

9.3 MongoDB Statistics

In the following table shows the statistics of the MongoDB dump, which is used for the mongodb cookbook. The stats can be display by entering > *mongo acmeair; db.stats()*.

```
1  > mongo acmeair;
2  > db.stats()
3  {
4    "db":"acmeair",
5    "collections":6,
6    "objects":1047718,
7    "avgObjSize":495.88796985448374,
8    "dataSize":519550752,
9    "storageSize":667631616,
10   "numExtents":30,
11   "indexes":4,
12   "indexSize":44223984,
13   "fileSize":1006632960,
14   "nsSizeMB":16,
15   "dataFileVersion":{
16     "major":4,
17     "minor":5
18   },
19   "extentFreeList":{
20     "num":0,
21     "totalSize":0
22   },
23   "ok":1
24 }
```

Listing 9.1: MongoDB Database Statistics

9.4 CWB Benchmark Definitions

The following subsections show complete examples of the CWB Benchmark definition, which were used for the individual benchmark executions.

9.4.1 AWS without slaves

This is an example of a CWB benchmark definition used for benchmarking Amazon EC2 instances. In this example, we used JMeter in its distributed mode.

```
IP_DB = "172.31.2.#{benchmark_id}"
IP_WEBAPP = "172.31.3.#{benchmark_id}"
PORT_WEBAPP = 9080

JMETER_SLAVES_NUM = 2
JMETER_NUM_THREADS = 2500
JMETER_RAMP_UP_TIME = 0
JMETER_DURATION = 1200

USER_IN_DB= 1000000
BENCHMARK_ITERATIONS = 1

INSTANCE_TYPE_WEBAPP = 'c4.large'
INSTANCE_TYPE_DB = 't2.medium'

ubuntu_ami = 'ami-d19e79be'
UBUNTU_USERNAME = 'ubuntu'
debian8_4 = 'ami-e05ab38f'
DEBIAN_USERNAME = 'admin' # sudo -i

IMAGE_WEBAPP = debian8_4
IMAGE_DB = debian8_4

JMETER_BASE_IP = '172.31.15.'
IP_JMETER_MASTER = "172.31.15."+(benchmark_id+100).to_s
#JMETER_BASE_IP+(JMETER_SLAVES_NUM+1).to_s

AWS_REG = 'eu-central-1'
AWS_A_ZONE = 'eu-central-1a'
AWS_SEC_GROUPS = ['cwb-web']

def jmeter_remotes
  remotes = ""
  for i in 1..JMETER_SLAVES_NUM
    remotes += (JMETER_BASE_IP+"#{i},")
  end
  return remotes
end

INSTANCE_TYPE_WEBAPP_sanitized = INSTANCE_TYPE_WEBAPP.gsub(/[\W]+/,"_")
INSTANCE_TYPE_DB_sanitized = INSTANCE_TYPE_DB.gsub(/[\W]+/,"_")

Vagrant.configure(VAGRANTFILE_API_VERSION) do config
  config.vm.define "mongodb" do mongodb
    mongodb.ssh.username = DEBIAN_USERNAME
    mongodb.vm.synced_folder '.', '/vagrant', disabled: true
    mongodb.vm.provider :aws do aws, override
      aws.region = AWS_REG
      aws.availability_zone = AWS_A_ZONE
      aws.ami = IMAGE_DB
      aws.instance_type = INSTANCE_TYPE_DB
      aws.security_groups = AWS_SEC_GROUPS
      aws.private_ip_address = IP_DB
      aws.tags = {
        'CWB_Function' => 'acmeair-mongodb'
      }
    end

    mongodb.vm.provision 'cwb', type: 'chef_client' do chef
    chef.node_name = 'mongodb'+execution_id.to_s
    chef.add_recipe 'acmeair_mongodb'
    chef.json =
    {
      'benchmark' => {
        'logging_enabled' =>true,
        'owner' => DEBIAN_USERNAME,
        'group' => DEBIAN_USERNAME
      }
    }
  end
end

config.vm.define "webapp" do webapp
  webapp.vm.synced_folder '.', '/vagrant', disabled: true
  webapp.ssh.username = DEBIAN_USERNAME
  webapp.vm.provider :aws do aws, override
    aws.region = AWS_REG
    aws.availability_zone = AWS_A_ZONE
    aws.ami = IMAGE_WEBAPP
    aws.instance_type = INSTANCE_TYPE_WEBAPP
    aws.security_groups = AWS_SEC_GROUPS
    aws.private_ip_address = IP_WEBAPP
    aws.tags = {
      'CWB_Function' => 'acmeair-morphia-webapp'
    }
  end

  webapp.vm.provision 'cwb', type: 'chef_client' do chef
    chef.node_name = 'webapp'+execution_id.to_s
    chef.add_recipe 'acmeair_wlp_morphia_distributed'
    chef.add_recipe 'cwb-tuning'
    chef.add_recipe 'cwb-monitoring'
```

```
93            chef.json =
94            {
95              'benchmark' => {
96                'logging_enabled' =>true,
97                'owner' => DEBIAN_USERNAME,
98                'group' => DEBIAN_USERNAME
99              },
100             'firewall' => {
101               'ports' => PORT_WEBAPP
102             },
103             'config' => {
104               'webapp' =>{
105                 'port' => {
106                   'http' => PORT_WEBAPP
107                 }
108               },
109               'tuning' => {
110                 'heap_xms' => '512m',
111                 'heap_xmx' => '3g'
112               }
113             },
114             'mongodb' => {
115               'name' => 'acmeair',
116               'ip' => IP_DB,
117               'port' => 27017,
118               'user' => {
119                 'name' => 'acmeairusr',
120                 'password' => 'Login4Acme!'
121               }
122             }
123           }
124         end
125       end
126
127       config.vm.define "jmetermaster", primary: true do
              jmetermaster
128         jmetermaster.ssh.username = UBUNTU_USERNAME
129         jmetermaster.vm.provider :aws do aws, override
130           aws.region = AWS_REG
131           aws.availability_zone = AWS_A_ZONE
132           aws.ami = 'ami-d19e79be'
133           aws.instance_type = 't2.medium'
134           aws.security_groups = AWS_SEC_GROUPS
135           aws.private_ip_address = IP_JMETER_MASTER
136           aws.tags = {
137             'CWB_Function' => 'jmetermaster'
138           }
139         end
140
141         jmetermaster.vm.provision 'cwb', type: 'chef_client' do
              chef
142           chef.node_name = 'jmetermaster-'+execution_id.to_s
143           chef.add_recipe 'jm-acmeair-api'
144           chef.add_recipe 'jm-acmeair-default-assets'
145           chef.add_recipe 'acmeair-single'
146           chef.add_recipe 'cwb-tuning'
147           chef.add_recipe 'cwb-monitoring'
148           chef.json =
149           {
150             'benchmark' => {
151               'logging_enabled' =>true
152             },
153             'cwbjmeter' => {
154               'config' => {
155                 'remotes' => jmeter_remotes,
156                 'slave' => false,
157                 'ssh_username' => UBUNTU_USERNAME,
158                 'xms_heap_size' => '512m',
159                 'xmx_heap_size' => '3g'
160               }
161             },
162               'acmeairapi' => {
163                 'testplan' => {
164                   'user_in_db' => USER_IN_DB,
165                   'connection_timeout' => 30000,
166                   'response_timeout' => 30000,
167                   'target_host' => {
168                     'port' => PORT_WEBAPP,
169                     'name' => IP_WEBAPP
170                   },
171                   'threadgroup' => {
172                     'num_threads' =>
                            JMETER_NUM_THREADS,
173                     'ramp_up_time' =>
                            JMETER_RAMP_UP_TIME,
174                     'duration' => JMETER_DURATION,
175                     'delay' => 0
176                   }
177                 }
178               },
179             'acmeair-single' => {
180               'benchmark_iterations' =>
                      BENCHMARK_ITERATIONS,
181               'distributed_benchmark' => true,
182               'results_file_name' => "j_exid#{execution_id}-#{
                      benchmark_name}-#{
                      INSTANCE_TYPE_WEBAPP_sanitized}_#{
                      INSTANCE_TYPE_DB_sanitized}-j#{
                      JMETER_SLAVES_NUM}-thr#{
                      JMETER_NUM_THREADS}-dur#{
                      JMETER_DURATION}-rt#{
                      JMETER_RAMP_UP_TIME}",
183               'log_file_upload_enabled' => true,
184               'log_file_name' => "l_exid#{execution_id}-#{
                      benchmark_name}-#{
                      INSTANCE_TYPE_WEBAPP_sanitized}_#{
                      INSTANCE_TYPE_DB_sanitized}-j#{
                      JMETER_SLAVES_NUM}-thr#{
                      JMETER_NUM_THREADS}-dur#{
                      JMETER_DURATION}-rt#{
                      JMETER_RAMP_UP_TIME}",
185               'testplan_file_name' => "jmeter_testplan"
186               }
187           }
188         end
189       end
190
191   end
```

Listing 9.2: CWB Benchmark Definition, AWS NoSlaves

9.4.2 GCE without slaves

This is an example of a CWB benchmark definition used for benchmarking GCE instances. Note: as special script has run before the provisioning of the webapp and the jmeter instances in order to query and save the ip addresses of the mongodb instance and the web application instance respectively. In GCE we did not test JMeter in its distributed mode, since the driver can generate sufficient load to measure the maximum throughput. For enabling the distributed mode, the getIP.py script has to be adapted, such that it appends the IP addresses to the same file and then the script has to be run in a loop with the JMeter slave node names as arguments (not tested!).

```
1  PORT_WEBAPP = 9080
2
3  FILESERVER_IP = '52.59.112.61'
4
5  JMETER_SLAVES_NUM = 0
6  JMETER_NUM_THREADS = 5000
7  JMETER_RAMP_UP_TIME = 0
8  JMETER_DURATION = 600
9
10 USER_IN_DB= 1000000
11 BENCHMARK_ITERATIONS = 1
12
13 INSTANCE_TYPE_WEBAPP = 'n1-highcpu-4'
14 INSTANCE_TYPE_DB = 'n1-standard-1'
15 INSTANCE_TYPE_JMETER = 'n1-standard-1'
16
17 GCE_ZONE = "europe-west1-b"
18 GCE_SCOPES = ["cloud-platform"]
19
20 DEBIAN_USERNAME = 'admin'
21 DEBIAN_SSH_KEY_PATH = "~/.ssh/
       google_compute_engine"
22 DEBIAN_IMAGE = 'debian-8-jessie-java'
23
24 INSTANCE_TYPE_WEBAPP_sanitized =
       INSTANCE_TYPE_WEBAPP.gsub(/[\W]+/,"_")
25 INSTANCE_TYPE_DB_sanitized = INSTANCE_TYPE_DB.
       gsub(/[\W]+/,"_")
26
27 Vagrant.configure(VAGRANTFILE_API_VERSION) do
       config
28   config.vm.provider :google do google, override
29     google.google_project_id = "cwb-applicationbenchmark
         "
30     google.google_client_email = "cwb-serviceaccount@cwb
         -applicationbenchmark.iam.gserviceaccount.com
         "
31     google.google_json_key_location = "/cwb-
         applicationbenchmark-8b123a531802.json"
32   end
33
34   config.vm.define "mongodb" do mongodb
35     mongodb.ssh.username = DEBIAN_USERNAME
36     mongodb.vm.synced_folder '.', '/vagrant', disabled:
         true
37     mongodb.vm.provider :google do google, override
38       google.name = "mongodb#{execution_id}"
39       google.zone = GCE_ZONE
40       google.machine_type = INSTANCE_TYPE_DB
41       google.image = DEBIAN_IMAGE
42       google.scopes = GCE_SCOPES
43       override.ssh.username = DEBIAN_USERNAME
44       override.ssh.private_key_path =
           DEBIAN_SSH_KEY_PATH
45     end
46
47     mongodb.vm.provision 'cwb', type: 'chef_client' do
         chef
48     chef.node_name = "mongodb#{execution_id}"
49     chef.add_recipe 'acmeair_mongodb'
50     chef.json =
51     {
52     'benchmark' => {
53       'logging_enabled' =>true,
54       'owner' => DEBIAN_USERNAME,
55       'group' => DEBIAN_USERNAME
56     }
57     }
58   end
59 end
60
61 config.vm.define "webapp" do webapp
62   webapp.vm.synced_folder '.', '/vagrant', disabled: true
63   webapp.ssh.username = DEBIAN_USERNAME
64   webapp.vm.provider :google do google, override
65     google.name = "webapp#{execution_id}"
66     google.zone = GCE_ZONE
67     google.machine_type = INSTANCE_TYPE_WEBAPP
68     google.image = DEBIAN_IMAGE
69     google.scopes = GCE_SCOPES
70     override.ssh.username = DEBIAN_USERNAME
71     override.ssh.private_key_path =
         DEBIAN_SSH_KEY_PATH
72   end
73
74   #for GCE, we have to run another script that queries
         the IP of the mongo db node before provisioning
         the webapp
75   webapp.vm.provision "file", source: "~/getIP.py",
         destination: "~/getIP.py"
76   webapp.vm.provision "shell", inline: "python getIP.py
         mongodb#{execution_id} europe-west1-b"
77   webapp.vm.provision 'cwb', type: 'chef_client' do chef
78     chef.node_name = "webapp#{execution_id}"
79     chef.add_recipe 'acmeair_wlp_morphia_distributed'
80     chef.add_recipe 'cwb-tuning'
81     chef.add_recipe 'cwb-monitoring'
82     chef.json =
83     {
84     'benchmark' => {
```

```
            'logging_enabled' =>true,
            'owner' => DEBIAN_USERNAME,
            'group' => DEBIAN_USERNAME
          },
            'firewall' => {
              'ports' => PORT_WEBAPP
            },
            'config' => {
              'webapp' =>{
                'port' => {
                  'http' => PORT_WEBAPP
                }
              },
              'tuning' => {
                'heap_xms' => '512m',
                'heap_xmx' => '3g'
              }
            },
            'mongodb' => {
              'ip_from_file' => true,
              'ip_file_path_name' => '/home/admin/ip.env',
              'name' => 'acmeair',
              'ip' => 'value should be overwritten!',
              'port' => 27017,
              'user' => {
                'name' => 'acmeairusr',
                'password' => 'Login4Acme!'
              }
            }
          }
        end
      end

      config.vm.define "jmetermaster", primary: true do
          jmetermaster
        jmetermaster.ssh.username = DEBIAN_USERNAME
        jmetermaster.vm.synced_folder '.', '/vagrant', disabled:
            true
        jmetermaster.vm.provider :google do google, override
          google.name = "jmetermaster#{execution_id}"
          google.zone = GCE_ZONE
          google.machine_type = INSTANCE_TYPE_JMETER
          google.image = DEBIAN_IMAGE
          google.scopes = GCE_SCOPES
          override.ssh.username = DEBIAN_USERNAME
          override.ssh.private_key_path =
              DEBIAN_SSH_KEY_PATH
        end

        #for GCE, we have to run another script that queries the
            IP of the webapp node before provisioning the
            JMeter Test Plan
        jmetermaster.vm.provision "file", source: "~/getIP.py",
            destination: "~/getIP.py"
        jmetermaster.vm.provision "shell", inline: "python getIP.
            py webapp#{execution_id} europe-west1-b"
        jmetermaster.vm.provision 'cwb', type: 'chef_client' do
            chef
          chef.node_name = "jmetermaster#{execution_id}"
          chef.add_recipe 'jm-acmeair-api'
          chef.add_recipe 'jm-acmeair-default-assets'
          chef.add_recipe 'acmeair-single'
          chef.add_recipe 'cwb-tuning'
          chef.add_recipe 'cwb-monitoring'

          chef.json =
          {
            'benchmark' => {
              'logging_enabled' =>true,
              'owner' => DEBIAN_USERNAME,
              'group' => DEBIAN_USERNAME
            },
            'cwbjmeter' => {
              'config' => {
                'remotes_from_file' => false,
                'remotes' => 'value should be overwritten!',
                'remotes_file_path_name' => '/home/admin/ip.
                    env',
                'slave' => false,
                'ssh_username' => DEBIAN_USERNAME,
                'xms_heap_size' => '512m',
                'xmx_heap_size' => '3g'
              }
            },
              'acmeairapi' => {
                'testplan' => {
                  'user_in_db' => USER_IN_DB,
                  'connection_timeout' => 30000,
                  'response_timeout' => 30000,
                  'target_host' => {
                    'port' => PORT_WEBAPP,
                    'name' => 'value should be overwritten!',
                    'name_from_file' => true,
                    'file_path_name' => '/home/admin/ip.env'
                  },
                  'threadgroup' => {
                    'num_threads' =>
                        JMETER_NUM_THREADS,
                    'ramp_up_time' =>
                        JMETER_RAMP_UP_TIME,
                    'duration' => JMETER_DURATION,
                    'delay' => 0
                  }
                }
              },
            'acmeair-single' => {
              'benchmark_iterations' =>
                  BENCHMARK_ITERATIONS,
              'distributed_benchmark' => false,
              'results_file_name' => "j_exid#{execution_id}-#{
                  benchmark_name}-#{
                  INSTANCE_TYPE_WEBAPP_sanitized}_#{
                  INSTANCE_TYPE_DB_sanitized}-j#{
                  JMETER_SLAVES_NUM}-thr#{
                  JMETER_NUM_THREADS}-dur#{
                  JMETER_DURATION}-rt#{
                  JMETER_RAMP_UP_TIME}",
              'log_file_upload_enabled' => true,
              'log_file_name' => "l_exid#{execution_id}-#{
                  benchmark_name}-#{
                  INSTANCE_TYPE_WEBAPP_sanitized}_#{
                  INSTANCE_TYPE_DB_sanitized}-j#{
                  JMETER_SLAVES_NUM}-thr#{
                  JMETER_NUM_THREADS}-dur#{
                  JMETER_DURATION}-rt#{
                  JMETER_RAMP_UP_TIME}",
              'testplan_file_name' => "jmeter_testplan",
              'fileserver' => {
                  'ip' => FILESERVER_IP
```

```
187                 }
188               }
189             }
190           end
191         end
192       end
```

Listing 9.3: CWB Benchmark Definition, GCE

9.4.3 JMeter

This is an example of the CWB benchmark definition, if the slaves are started separately.

```
1   SSH_USERNAME = 'ubuntu'
2   # after this time the slaves will terminate. actually not
        needed, cwb has also a timeout option
3   TIMEOUT_IN_MINUTES = 110
4
5   JMETER_SLAVES_NUM = 20 # will start 20 slave instances!
6
7   # the ips are calculated, make sure in the other vagrant files
        they are calculated the same!
8   JMETER_BASE_IP = '172.31.15.'
9   IP_JMETER_MASTER = JMETER_BASE_IP+(
        JMETER_SLAVES_NUM+1).to_s
10
11  AWS_REG = 'eu-central-1'
12  AWS_A_ZONE = 'eu-central-1a'
13  AWS_SEC_GROUPS = ['cwb-web']
14
15  Vagrant.configure(VAGRANTFILE_API_VERSION) do
        config
16    config.ssh.username = SSH_USERNAME
17
18    (1..JMETER_SLAVES_NUM).each do i
19      if i == 1
20        config.vm.define "jmeterslave#{i}", primary: true do
              jmeterslave
21          jmeterslave.vm.synced_folder '.', '/vagrant', disabled
                : true
22          jmeterslave.vm.provider :aws do aws, override
23            aws.region = AWS_REG
24            aws.availability_zone = AWS_A_ZONE
25            aws.ami = 'ami-d19e79be'
26            aws.instance_type = 't2.small'
27            aws.security_groups = AWS_SEC_GROUPS
28            aws.private_ip_address = JMETER_BASE_IP+"#{i}"
29            aws.tags = {
30              'CWB_Function' => "jmeterslave#{i}"
31            }
32          end
33
34          jmeterslave.vm.provision 'cwb', type: 'chef_client' do
                chef
35            chef.node_name = "jmeterslave#{i}-"+
                  execution_id.to_s
36            chef.add_recipe 'cwb-jmeter'
37            chef.add_recipe 'jm-acmeair-default-assets'
38            chef.add_recipe 'cwb-timeout'
39            chef.add_recipe 'cwb-tuning'
40            chef.add_recipe 'cwb-monitoring'
41            chef.json =
42            {
43              'cwbjmeter' => {
44                'config' => {
45                  'remotes' => '',
46                  'slave' => true,
47                  'ssh_username' => SSH_USERNAME,
48                  'xms_heap_size' => '512m',
49                  'xmx_heap_size' => '1024m'
50                }
51              },
52              'cwb-timeout' => {
53                'timeout_in_minutes' =>
                      TIMEOUT_IN_MINUTES
54              },
55              'cwb-monitoring' => {
56                'jstatd' => {
57                  'rmi_port' => 2020
58                }
59              }
60            }
61          end
62        end
63      else
64        config.vm.define "jmeterslave#{i}" do jmeterslave
65          jmeterslave.vm.synced_folder '.', '/vagrant', disabled
                : true
66          jmeterslave.vm.provider :aws do aws, override
67            aws.region = AWS_REG
68            aws.availability_zone = AWS_A_ZONE
69            aws.ami = 'ami-d19e79be'
70            aws.instance_type = 't2.small'
71            aws.security_groups = AWS_SEC_GROUPS
72            aws.private_ip_address = JMETER_BASE_IP+"#{i}"
73            aws.tags = {
74              'CWB_Function' => "jmeterslave#{i}"
75            }
76          end
77
78          jmeterslave.vm.provision 'cwb', type: 'chef_client' do
                chef
79            chef.node_name = "jmeterslave#{i}-"+
                  execution_id.to_s
80            chef.add_recipe 'cwb-jmeter'
81            chef.add_recipe 'jm-acmeair-default-assets'
82            chef.add_recipe 'cwb-tuning'
83            chef.add_recipe 'cwb-monitoring'
84            chef.json =
85            {
86              'cwbjmeter' => {
87                'config' => {
88                  'remotes' => '',
89                  'slave' => true,
90                  'ssh_username' => SSH_USERNAME,
```

```
91                  'xms_heap_size' => '512m',
92                  'xmx_heap_size' => '1024m'
93                }
94              },
95              'cwb-monitoring' => {
96                'jstatd' => {
97                  'rmi_port' => 2020
98                }
99              }
100           }
```

```
101             end
102           end
103         end
104       end
105     end
```

Listing 9.4: CWB Benchmark Definition, JMeter SlavesOnly

9.5 Score Normalization Comparison

c	$\overline{M_c}$	$\frac{P_c}{hour}$	$\frac{\text{mio requests}}{\$}$	rank	PCN	PCN-rank	CBS-score	CBS-rank
A_x2	940.70	$0.060	56.442	1	15678.309	1	2.317	1
G_gp1	722.21	$0.055	47.272	2	13131.071	2	1.146	2
G_co2	1095.276346	$0.084	46.940	3	13039.004	3	1.104	3
A_x1m	180.15	$0.015	43.235	4	12009.797	4	0.631	4
A_x1s	339.05	$0.030	40.686	5	11301.678	5	0.305	5
A_co2_2	1472.01	$0.134	39.546	6	10985.119	6	0.160	6
G_b1	321.73	$0.030	38.608	7	10724.473	7	0.040	7
G_co4	1791.975896	$0.168	38.399	8	10666.523	8	0.013	8
A_co2_1	1417.09	$0.134	38.071	9	10575.290	9	-0.029	9
G_gp2	1102.486953	$0.110	36.081	10	10022.609	10	-0.283	10
A_gp2_2	1302.83	$0.143	32.799	11	9110.705	11	-0.702	11
A_gp2_1	1247.37	$0.143	31.402	12	8722.864	12	-0.880	12
G_gp4	1888.366625	$0.220	30.901	13	8583.485	13	-0.944	13
A_co4	2192.07	$0.267	29.556	14	8210.008	14	-1.116	14
A_gp4	1939.74	$0.285	24.502	15	6806.105	15	-1.761	15

Table 9.1: Normalization Comparison

9.6 Creating Real-Workloads from Access Logs

There are some good sources, we would like to share. Last checked on: 2016/08/24

- https://lincolnloop.com/blog/load-testing-jmeter-part-3-replaying-apache-logs/
- https://pypi.python.org/pypi/createurls
- https://www.blazemeter.com/blog/stop-making-assumptions-learn-how-replay-your-production-traffic-jmeter
- https://youtu.be/8kbfseBenSI

DVD

Contents of the DVD

- Zusfsg.txt
- Abstract.txt
- Masterarbeit.pdf
- thesis-code.zip: snippets used in the thesis
- cookbooks.zip: Benchmark Cookbook Repository
- examples.zip: Examples of Vagrant Files
- benchmark-results.7z: Dump of Data used for the evaluation
- win-python.zip: Environment and scripts used for evaluation
- fileserver.zip: source code of the used fileserver
- cwb.sh: convenience shell script (Note: absolute paths!)

Bibliography

[ACC+02] Cristiana Amza, Anupam Chanda, Alan L Cox, Sameh Elnikety, Romer Gil, Karthick Rajamani, Willy Zwaenepoel, Emmanuel Cecchet, and Julie Marguerite. Specification and implementation of dynamic web site benchmarks. In *Workload Characterization, 2002. WWC-5. 2002 IEEE International Workshop on*, pages 3–13. IEEE, 2002.

[AFG+09] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy H Katz, Andrew Konwinski, Gunho Lee, David A Patterson, Ariel Rabkin, Ion Stoica, et al. Above the clouds: A berkeley view of cloud computing. 2009.

[AFG+10] Michael Armbrust, Armando Fox, Rean Griffith, Anthony D Joseph, Randy Katz, Andy Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, Ion Stoica, and Matei Zaharia. A View of Cloud Computing. *Communications of the ACM*, 53(4):50–58, 2010.

[Ama16a] Amazon Web Services, Inc. Amazon EC2 bursting instance types. https://aws.amazon.com/ec2/instance-types/$#$burst, 2016. Accessed: 2016-07-28.

[Ama16b] Amazon Web Services, Inc. Amazon EC2 instance types. https://aws.amazon.com/ec2/instance-types/, 2016. Accessed: 2016-07-28.

[Ama16c] Amazon Web Services, Inc. Amazon EC2 pricing. https://aws.amazon.com/ec2/pricing/, 2016. Accessed: 2016-07-28.

[Ama16d] Amazon Web Services, Inc. Amazon EC2 purchasing options. https://aws.amazon.com/ec2/purchasing-options/, 2016. Accessed: 2016-07-28.

[Ama16e] Amazon Web Services, Inc. Amazon EC2 user guide, bursting instances, 2016.

[BCMS14] Stefano Bonnini, Livio Corain, Marco Marozzi, and Luigi Salmaso. *Nonparametric Hypothesis Testing: Rank and Permutation Methods with Applications in R*. John Wiley & Sons, 2014.

[BKKL09] Carsten Binnig, Donald Kossmann, Tim Kraska, and Simon Loesing. How is the weather tomorrow?: towards a benchmark for the cloud. In *Proceedings of the Second International Workshop on Testing Database Systems*, page 9. ACM, 2009.

[BL10] Paul Brebner and Anna Liu. Performance and cost assessment of cloud services. In *International Conference on Service-Oriented Computing*, pages 39–50. Springer, 2010.

[BLL+14] Amir Hossein Borhani, Philipp Leitner, Bu-Sung Lee, Xiaorong Li, and Terence Hung. Wpress: Benchmarking infrastructure-as-a-service cloud computing systems for on-line transaction processing applications. In *Proceedings of the 18th IEEE International Enterprise Distributed Object Computing Conference (EDOC)*, 2014.

[BS10] Sean Kenneth Barker and Prashant Shenoy. Empirical evaluation of latency-sensitive application performance in the cloud. In *Proceedings of the first annual ACM SIGMM conference on Multimedia systems*, pages 35–46. ACM, 2010.

[BYV+09] R. Buyya, C. Yeo, S. Venugopal, J. Broberg, and I. Brandic. Cloud Computing and Emerging IT Platforms: Vision, Hype, and Reality for Delivering Computing as the 5th Utility. *Future Generation Computing Systems*, 25:599–616, 2009.

[CCE+03] Emmanuel Cecchet, Anupam Chanda, Sameh Elnikety, Julie Marguerite, and Willy Zwaenepoel. Performance comparison of middleware architectures for generating dynamic web content. In *Proceedings of the ACM/IFIP/USENIX 2003 International Conference on Middleware*, pages 242–261. Springer-Verlag New York, Inc., 2003.

[CCVK13] Mohan Baruwal Chhetri, Sergei Chichin, Quoc Bao Vo, and Ryszard Kowalczyk. Smart cloudbench - automated performance benchmarking of the cloud. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 414–421. IEEE, 2013.

[Che16a] Chef Software, Inc. Chef documentation. https://docs.chef.io/, 2016. Accessed: 2016-08-01.

[Che16b] Chef Software, Inc. Chef documentation: Attributes. https://docs.chef.io/attributes.html, 2016.

[CLN12] Sivadon Chaisiri, Bu-Sung Lee, and Dusit Niyato. Optimization of resource provisioning cost in cloud computing. *IEEE Transactions on Services Computing*, 5(2):164–177, 2012.

[Clo15] CloudHarmony, Inc. Cloud Harmony. https://cloudharmony.com, 2015. Accessed: 2016-07-28.

[CMS13] Matheus Cunha, Nabor C Mendonça, and Americo Sampaio. A declarative environment for automatic performance evaluation in iaas clouds. *IEEE CLOUD*, 2013:285–292, 2013.

[CMS16] M Cunha, NC Mendonça, and A Sampaio. Cloud crawler: a declarative performance evaluation environment for infrastructure-as-a-service clouds. *Concurrency and Computation: Practice and Experience*, 2016.

[Com07] The New York Times Company. Self-service, prorated supercomputing fun! http://open.blogs.nytimes.com/2007/11/01/self-service-prorated-supercomputing-fun, 2007. Accessed: 2016-07-31.

[Con05] ObjectWeb Consortium. Objectweb implementation of the tpc-w benchmark. http://jmob.objectWeb.org/tpcw.html, 2005. Accessed: 2016-07-28.

[Con09] OW2 Consortium. Rubis: Rice university bidding system. http://rubis.ow2.org/, 2009. Accessed: 2016-08-05.

[Cor09] Standard Performance Evaluation Corporation. SPEC specweb2009. https://www.spec.org/web2009/, 2009. Accessed: 2016-07-28.

[Cor16] Standard Performance Evaluation Corporation. SPEC cloud iaas 2016. https://www.spec.org/cloud_iaas2016/, 2016. Accessed: 2016-08-08.

[CRB+11] Rodrigo N Calheiros, Rajiv Ranjan, Anton Beloglazov, César AF De Rose, and Rajkumar Buyya. Cloudsim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. *Software: Practice and Experience*, 41(1):23–50, 2011.

[Cro95] Lawrence H Cross. *Grading students*. Department of Education, Catholic University of America, 1995.

[CST+10] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 143–154. ACM, 2010.

[CUW+11] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, Prashant Shenoy, Fabien Mottet, Vivien Quema, and Guillaume Pierre. Benchlab: Benchmarking with real web applications and web browsers. Eurosys, 2011.

[CUWS11] Emmanuel Cecchet, Veena Udayabhanu, Timothy Wood, and Prashant Shenoy. Benchlab: an open testbed for realistic benchmarking of web applications. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 4–4. USENIX Association, 2011.

[Dis10] Distributed Management Task Force, Inc. Open virtualization format specification. http://www.dmtf.org/sites/default/files/standards/documents/DSP0243_1.1.0.pdf, 2010.

[DPC10] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Ec2 performance analysis for resource provisioning of service-oriented applications. In *Service-Oriented Computing. ICSOC/ServiceWave 2009 Workshops*, pages 197–207. Springer, 2010.

[DPCCM13] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudre-Mauroux. Oltp-bench: An extensible testbed for benchmarking relational databases. *Proc. VLDB Endow.*, 7(4):277–288, December 2013.

[DRK14] Rajdeep Dua, A Reddy Raja, and Dharmesh Kakadia. Virtualization vs containerization to support paas. In *Cloud Engineering (IC2E), 2014 IEEE International Conference on*, pages 610–614. IEEE, 2014.

[DSP10] Peter H Deussen, Linda Strick, and J Peters. Cloud-computing für die öffentliche verwaltung. *ISPRAT-Studie, Fraunhofer Fokus*, 2010.

[Eri15] Bayo Erinle. *Performance Testing with Jmeter - Second Edition*. Packt Publishing Ltd, 2015.

[ERR10] M. A. El-Refaey and M. A. Rizkaa. Cloudgauge: A dynamic cloud and virtualization benchmarking suite. In *Enabling Technologies: Infrastructures for Collaborative Enterprises (WETICE), 2010 19th IEEE International Workshop on*, pages 66–75, June 2010.

[ETP+13] Roberto R Expósito, Guillermo L Taboada, Xoán C Pardo, Juan Tourino, and Ramón Doallo. Running scientific codes on amazon ec2: A performance analysis of five high-end instances. *Journal of Computer Science & Technology*, 13, 2013.

[FAK+12] Michael Ferdman, Almutaz Adileh, Onur Kocberber, Stavros Volos, Mohammad Alisafaee, Djordje Jevdjic, Cansu Kaynak, Adrian Daniel Popescu, Anastasia Ailamaki, and Babak Falsafi. Clearing the clouds: a study of emerging scale-out workloads on

modern hardware. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 37–48, New York, NY, USA, 2012. ACM.

[FAS+12] Enno Folkerts, Alexander Alexandrov, Kai Sachs, Alexandru Iosup, Volker Markl, and Cafer Tosun. Benchmarking in the cloud: What it should, can, and cannot be. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 173–188. Springer, 2012.

[FFH12] Florian Fittkau, Sören Frey, and Wilhelm Hasselbring. Cdosim: Simulating cloud deployment options for software migration support. In *2012 IEEE 6th International Workshop on the Maintenance and Evolution of Service-Oriented and Cloud-Based Systems (MESOCA)*, pages 37–46. IEEE, 2012.

[FJV+12] Benjamin Farley, Ari Juels, Venkatanathan Varadarajan, Thomas Ristenpart, Kevin D Bowers, and Michael M Swift. More for your money: exploiting performance heterogeneity in public clouds. In *Proceedings of the Third ACM Symposium on Cloud Computing*, page 20. ACM, 2012.

[FLR+14] Christoph Fehling, Frank Leymann, Ralph Retter, Walter Schupeck, and Peter Arbitter. *Cloud Computing Patterns: Fundamentals to Design, Build, and Manage Cloud Applications*. Springer, 2014.

[Fou16a] Apache Software Foundation. Apache Jmeter introduction. http://jmeter.apache.org/usermanual/intro.html, 2016.

[Fou16b] Apache Software Foundation. Apache Jmeter test-plan. http://jmeter.apache.org/usermanual/test_plan.html, 2016.

[Fow03] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, Boston, 2003.

[FW86] Philip J. Fleming and John J. Wallace. How not to lie with statistics: The correct way to summarize benchmark results. *Commun. ACM*, 29(3):218–221, March 1986.

[Gar15] Gartner. Gartner identifies the top 10 strategic technology trends for 2015. http://www.gartner.com/newsroom/id/2867917, 2015. Accessed: 2016-07-19.

[Gar16] Gartner. Gartner identifies the top 10 strategic technology trends for 2016. http://www.gartner.com/newsroom/id/3143521, 2016. Accessed: 2016-07-19.

[GCMS15] Marcelo Gonçalves, Matheus Cunha, Nabor C Mendonça, and Américo Sampaio. Performance inference: a novel approach for planning the capacity of iaas cloud applications. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 813–820. IEEE, 2015.

[Ger15] Shai Gershon. Society of Electrical and Electronics Engineers in Israel: sustained use discounts. http://www.seeei.org/EL2013/Program/025.htm, 2015. Accessed: 2016-07-28.

[GLOT13] Lee Gillam, Bin Li, John O'Loughlin, and Anuz Pratap Singh Tomar. Fair benchmarking for cloud computing systems. *Journal of Cloud Computing: Advances, Systems and Applications*, 2(1):1, 2013.

[Goo16a] Google, Inc. Google Compute Engine custom machine types. https://cloud.google.com/compute/docs/instances/creating-instance-with-custom-machine-type, 2016. Accessed: 2016-07-28.

[Goo16b] Google, Inc. Google Compute Engine preemptible vm instances. https://cloud.google.com/compute/docs/instances/preemptible, 2016. Accessed: 2016-07-28.

[Goo16c] Google, Inc. Google Compute Engine pricing. https://cloud.google.com/compute/pricing, 2016. Accessed: 2016-07-28.

[Goo16d] Google, Inc. Google Compute Engine standard machine types. https://cloud.google.com/compute/docs/machine-types#standard_machine_types, 2016. Accessed: 2016-07-28.

[Goo16e] Google, Inc. Google Compute Engine sustained use discounts. https://cloud.google.com/compute/pricing#sustained_use, 2016. Accessed: 2016-07-28.

[Gre13] Brendan Gregg. *Systems Performance: Enterprise and the Cloud*. Pearson Education, 2013.

[Hal08] Emily H Halili. *Apache JMeter: A practical beginner's guide to automated testing and performance measurement for your websites*. Packt Publishing Ltd, 2008.

[Has16] HashiCorp. Available vagrant plugins. https://github.com/mitchellh/vagrant/wiki/Available-Vagrant-Plugins, 2016. Accessed: 2016-08-01.

[Hil09] D. Hilley. Cloud Computing: A Taxonomy of Platform and Infrastructure-Level Offerings, 2009.

[HSS+10] Mohammad Hajjat, Xin Sun, Yu-Wei Eric Sung, David Maltz, Sanjay Rao, Kunwadee Sripanidkulchai, and Mohit Tawarmalani. Cloudward bound: planning for beneficial migration of enterprise applications to the cloud. In *ACM SIGCOMM Computer Communication Review*, volume 40, pages 243–254. ACM, 2010.

[Hup09] Karl Huppler. The art of building a good benchmark. In *Technology Conference on Performance Evaluation and Benchmarking*, pages 18–30. Springer, 2009.

[Hüt12] Michael Hüttermann. Infrastructure as code. In *DevOps for Developers*, pages 135–156. Springer, 2012.

[ICH+14] Alexandru Iosup, Mihai Capotă, Tim Hegeman, Yong Guo, Wing Lung Ngai, Ana Lucia Varbanescu, and Merijn Verstraaten. Towards benchmarking iaas and paas clouds for graph analytics. In *Workshop on Big Data Benchmarks*, pages 109–131. Springer, 2014.

[IOY+11] Alexandru Iosup, Simon Ostermann, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. Performance analysis of cloud computing services for many-tasks scientific computing. *IEEE Trans. Parallel Distrib. Syst.*, 22(6):931–945, June 2011.

[JRM+10] Keith R Jackson, Lavanya Ramakrishnan, Krishna Muriki, Shane Canon, Shreyas Cholia, John Shalf, Harvey J Wasserman, and Nicholas J Wright. Performance analysis of high performance computing applications on the amazon web services cloud. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 159–168. IEEE, 2010.

[JSM+12] Deepal Jayasinghe, Galen Swint, Simon Malkowski, Jack Li, Qingyang Wang, Junhee Park, and Calton Pu. Expertus: A generator approach to automate performance testing in iaas clouds. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 115–122. IEEE, 2012.

[KKL10] Donald Kossmann, Tim Kraska, and Simon Loesing. An evaluation of alternative architectures for transaction processing in the cloud. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, pages 579–590. ACM, 2010.

[KKR14] Jörn Kuhlenkamp, Markus Klems, and Oliver Röss. Benchmarking scalability and elasticity of distributed database systems. *Proceedings of the VLDB Endowment*, 7(12):1219–1230, 2014.

[KLIZ12] Dilip Kumar Krishnappa, Eric Lyons, David Irwin, and Michael Zink. Network capabilities of cloud services for a real time scientific application. In *Local Computer Networks (LCN), 2012 IEEE 37th Conference on*, pages 487–495. IEEE, 2012.

[Kou06] Samuel Kounev. *Performance Engineering of Distributed Component-based Systems*. Citeseer, 2006.

[Lab12] Stine Labes. *Grundlagen des Cloud Computing*. Universitätsverlag der TU Berlin, 2012.

[LBMAL12] Tania Lorido-Botrán, José Miguel-Alonso, and Jose A Lozano. Auto-scaling techniques for elastic applications in cloud environments. *Department of Computer Architecture and Technology, University of Basque Country, Tech. Rep. EHU-KAT-IK-09-12*, 2012.

[LC16] Philipp Leitner and Jürgen Cito. Patterns in the Chaos – a Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Transactions on Internet Technology (TOIT)*, 16(3):15, 2016.

[LG] Mintu M Ladani and Vinit Kumar Gupta. A framework for performance analysis of computing clouds. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, pages 245–247.

[LML+11] Alexander Lenk, Michael Menzel, Johannes Lipsky, Stefan Tai, and Philipp Offermann. What are you paying for? performance benchmarking for infrastructure-as-a-service offerings. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 484–491. IEEE, 2011.

[LS15] Philipp Leitner and Joel Scheuner. Bursting with possibilities–an empirical study of credit-based bursting cloud instance types. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*, pages 227–236. IEEE, 2015.

[LYZK10] Ang Li, Xiaowei Yang, Ming Zhang, and S Kandula. Cloudcmp: Shopping for a cloud made easy. *HotCloud*, 10:1–7, 2010.

[LZK+11] Ang Li, Xuanran Zong, Srikanth Kandula, Xiaowei Yang, and Ming Zhang. Cloudprophet: towards application performance prediction in cloud. *ACM SIGCOMM Computer Communication Review*, 41(4):426–427, 2011.

[LZO+13] Zheng Li, He Zhang, Liam O'Brien, Rainbow Cai, and Shayne Flint. On evaluating commercial cloud services: A systematic review. *Journal of Systems and Software*, 86(9):2371–2393, 2013.

[MG11] Peter Mell and Tim Grance. The nist definition of cloud computing. 2011.

[MH11] Ming Mao and Marty Humphrey. Auto-scaling to minimize cost and meet application deadlines in cloud workflows. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, page 49. ACM, 2011.

[MH12] Ming Mao and Marty Humphrey. A performance study on the vm startup time in the cloud. In *Proceedings of the 2012 IEEE Fifth International Conference on Cloud Computing*, CLOUD '12, pages 423–430, Washington, DC, USA, 2012. IEEE Computer Society.

[Moo11] Geoffrey Moore. Systems of engagement and the future of enterprise it-a sea change in enterprise it. *AIIM, Silver Spring*, 2011.

[NB09] Jeffrey Napper and Paolo Bientinesi. Can cloud computing reach the top500? In *Proceedings of the combined workshops on UnConventional high performance computing workshop plus memory access workshop*, pages 17–20. ACM, 2009.

[OG14] John O'Loughlin and Lee Gillam. Performance evaluation for cost-efficient public infrastructure cloud use. In *International Conference on Grid Economics and Business Models*, pages 133–145. Springer, 2014.

[OIY+08] Simon Ostermann, Alexandru Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. An early performance analysis of cloud computing services for scientific computing. *Delft University of Technology, Tech. Rep*, 2008.

[OIY+09] Simon Ostermann, Alexandria Iosup, Nezih Yigitbasi, Radu Prodan, Thomas Fahringer, and Dick Epema. A performance analysis of ec2 cloud computing services for scientific computing. In *International Conference on Cloud Computing*, pages 115–131. Springer, 2009.

[Pat11] R. P. Pathak. *Research in Education and Psychology*. Pearson, 2011.

[Pok16] Andrey Pokhilko. Apache jmeter plugins. https://jmeter-plugins.org, 2016.

[RGVS+12] Tilmann Rabl, Sergio Gómez-Villamor, Mohammad Sadoghi, Victor Muntés-Mulero, Hans-Arno Jacobsen, and Serge Mankovskii. Solving big data challenges for enterprise application performance management. *Proceedings of the VLDB Endowment*, 5(12):1724–1735, 2012.

[Ryz15] Ilia Ryzhov. Performance analysis of virtual machines from two major iaas providers. Master's thesis, 2015.

[Sac11] Kai Sachs. *Performance modeling and benchmarking of event-based systems*. Sierke, 2011.

[Sac12] Matthew Sacks. *Web Testing Practices*, pages 27–43. Apress, Berkeley, CA, 2012.

[SASA+11] K. Salah, M. Al-Saba, M. Akhdhor, O. Shaaban, and M. I. Buhari. Performance evaluation of popular cloud iaas providers. In *Internet Technology and Secured Transactions (ICITST), 2011 International Conference for*, pages 345–349, Dec 2011.

[Sch14] Joel Scheuner. Cloud workbench, 2014.

[SCLG15] Joel Scheuner, Jürgen Cito, Philipp Leitner, and Harald Gall. Cloud workbench: Benchmarking iaas providers based on infrastructure-as-code. In *Proceedings of the 24th International Conference on World Wide Web*, pages 239–242. ACM, 2015.

[SDQR10] Jörg Schad, Jens Dittrich, and Jorge-Arnulfo Quiané-Ruiz. Runtime measurements in the cloud: observing, analyzing, and reducing variance. *Proceedings of the VLDB Endowment*, 3(1-2):460–471, 2010.

[SKBB09] Kai Sachs, Samuel Kounev, Jean Bacon, and Alejandro Buchmann. Performance evaluation of message-oriented middleware using the specjms2007 benchmark. *Performance Evaluation*, 66(8):410–434, 2009.

[SLCG14] Joel Scheuner, Philipp Leitner, Jürgen Cito, and Harald Gall. Cloud WorkBench - Infrastructure-as-Code Based Cloud Benchmarking. In *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'14)*, 2014.

[SLSR+08] Ignacio Silva-Lepe, Revathi Subramanian, Isabelle Rouvellou, Thomas Mikalsen, Judah Diament, and Arun Iyengar. Soalive service catalog: A simplified approach to describing, discovering and composing situational enterprise services. In *International Conference on Service-Oriented Computing*, pages 422–437. Springer, 2008.

[Smi00] Wayne D Smith. Tpc-w: Benchmarking an ecommerce solution. http://www.tpc.org/tpcw/tpcw_ex.asp, 2000. Accessed: 2016-08-05.

[Spy13] Andrew Spyker. Announcing acme air .. performance sample/benchmark showcasing mobile and cloud at web scale. http://ispyker.blogspot.ch/2013/05/announcing-acme-air-performance.html, 2013.

[SSS+08] Will Sobel, Shanti Subramanyam, Akara Sucharitakul, Jimmy Nguyen, Hubert Wong, Arthur Klepchukov, Sheetal Patil, Armando Fox, and David Patterson. Cloudstone: Multi-platform, multi-language benchmark and measurement tools for web 2.0. In *Proc. of CCA*, volume 8, 2008.

[TPC15] TPC. TPC-E a new on-line transaction processing (oltp) workload developed by the tpc. http://www.tpc.org/tpce/, 2015. Accessed: 2016-08-07.

[VK14] Quoc Bao Vo and Ryszard Kowalczyk. Smart cloudbench - test drive the cloud before you buy. *Service Research and Innovation*, page 59, 2014.

[VMSK12] Marco Vieira, Henrique Madeira, Kai Sachs, and Samuel Kounev. Resilience benchmarking. In *Resilience Assessment and Evaluation of Computing Systems*, pages 283–301. Springer, 2012.

[VV12] Venu Vedam and Jayanti Vemulapati. Demystifying cloud benchmarking paradigman in depth view. In *2012 IEEE 36th Annual Computer Software and Applications Conference*, pages 416–421. IEEE, 2012.

[Wal08] Edward Walker. Benchmarking amazon ec2 for hig-performance scientific computing. *; login:: the magazine of USENIX & SAGE*, 33(5):18–23, 2008.

[Win02] R Scott Winters. *Score normalization as a fair grading practice*. ERIC Clearinghouse on Assessment and Evaluation, 2002.

[XAL10] Ye Xiaotao, Lv Aili, and Zhao Lin. Research of high performance computing with clouds. In *Proc. International Symposium Computer Science and Computational Technology*, pages 289–293. Citeseer, 2010.

[YBDS08] Lamia Youseff, Maria Butrico, and Dilma Da Silva. Toward a unified ontology of cloud computing. In *2008 Grid Computing Environments Workshop*, pages 1–10. IEEE, 2008.

[ZCB10] Qi Zhang, Lu Cheng, and Raouf Boutaba. Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1):7–18, 2010.

[Zha01] Xiaolan Zhang. *Application-specific benchmarking*. PhD thesis, Harvard University Cambridge, Massachusetts, 2001.