

2018

Elasticity Measurement in CaaS Environments - Extending the Existing BUNGEE Elasticity Benchmark to AWS's Elastic Container Service

Nora Limbourg

Technological University Dublin, Dublin

Follow this and additional works at: <https://arrow.tudublin.ie/scschcomdis>



Part of the [Computer Engineering Commons](#)

Recommended Citation

Limbourg, N. (2019). Elasticity Measurement in CaaS Environments - Extending the Existing BUNGEE Elasticity Benchmark to AWS's Elastic Container Service. M.Sc. Dissertation in Computing (Advanced Software Development), DIT, 2018.

This Dissertation is brought to you for free and open access by the School of Computer Science at ARROW@TU Dublin. It has been accepted for inclusion in Dissertations by an authorized administrator of ARROW@TU Dublin. For more information, please contact arrow.admin@tudublin.ie, aisling.coyne@tudublin.ie, vera.kilshaw@tudublin.ie.

Elasticity Measurement in CaaS Environments - Extending the Existing BUNGEE Elasticity Benchmark to AWS's Elastic Container Service



Nora Limbourg

A dissertation submitted in partial fulfilment of the requirements of
Dublin Institute of Technology for the degree of
M.Sc. in Computing (Advanced Software Development)

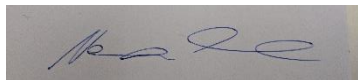
June 2018

I certify that this dissertation which I now submit for examination for the award of MSc. in Computing (Advanced Software Development), is entirely my own work and has not been taken from the work of others save and to the extent that such work has been cited and acknowledged within the text of my work.

This dissertation was prepared according to the regulations for postgraduate study of the Dublin Institute of Technology and has not been submitted in whole or part for an award in any other Institute or University.

The work reported on in this dissertation conforms to the principles and requirements of the Institute's guidelines for ethics in research.

Signed:

A rectangular box containing a handwritten signature in blue ink. The signature is stylized and appears to be 'K. R. L.' or similar.

Date:

15 June 2018

ABSTRACT

Rapid elasticity and automatic scaling are core concepts of most current cloud computing systems. Elasticity describes how well and how fast cloud systems adapt to increases and decreases in workload.

In parallel, software architectures are moving towards employing containerised microservices running on systems managed by container orchestration platforms. Cloud users who employ such container-based systems may want to compare the elasticity of different systems or system settings to ensure rapid elasticity and maintain service level objectives while avoiding over-provisioning.

Previous research has established a variety of metrics to measure elasticity. Some existing benchmark tools are designed to measure elasticity in “Infrastructure as a Service” (IaaS) systems, but no research exists to date for measuring elasticity in systems based on containers and container orchestration.

In this dissertation, an existing benchmark designed for IaaS systems, the BUNGEE benchmark developed at the University of Würzburg, was extended to be applicable to Amazon’s Elastic Container Service, a container-based cloud system. An experiment was conducted to test if the extension of the BUNGEE benchmark described in this dissertation delivers reproducible results and is therefore valid.

For validation, the crucial phase of the benchmark - the system analysis phase - was run 32 times. It was established with statistical tests if the results vary by more than the acceptable level.

Results indicate that there is some amount of variability, but it does not exceed the acceptable level and is consistent with the amount of performance variability encountered by other researchers in Amazon’s cloud systems.

Therefore, it is concluded that the BUNGEE benchmark is likely applicable to container-based cloud systems. However, some parameters and configuration settings specific to container orchestration systems were identified that could impede reproducibility of results and should be considered in future experiments.

Key words: *Elasticity, BUNGEE, containers, benchmark, ECS, Elastic Container Service*

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Dr Patrick Tobin. With his enthusiasm about the topic, his help and advice, he crucially contributed to this dissertation.

I would also like to express my gratitude towards Prof Sarah Jane Delany, Deirdre Lawless and David Ng of the Dublin Institute of Technology, for supporting me with different aspects of this work.

Special thanks also to Nikolas Herbst, André Bauer, Veronika Lesch and Jóakim von Kistowski of the University of Würzburg for their continued and exceptional support, feedback and input throughout the dissertation.

Further, I would like to express my sincere gratitude towards Prof Dr Christof Menzel who, some years back, sparked my interest in Computer Science with his amazing lectures and support.

I would like to acknowledge Amazon Web Services Inc. who contributed by providing cloud credits.

Further thanks also to my fellow students Fiona Delaney and Jefferson Ferreira, who were a dream team during the studies and the dissertation.

Many thanks also to my parents Prof Dr Maria Limbourg and Dr Kurt Limbourg, who never stopped believing in me, however complicated the circumstances.

I also would like to thank my partner Fernando Morais who, with emotional support and practical advice, contributed to the success of this work.

TABLE OF CONTENTS

ABSTRACT	ii
LIST OF ABBREVIATIONS	vii
TABLE OF FIGURES.....	viii
TABLE OF TABLES	x
1 INTRODUCTION	1
1.1 Background	1
1.2 Research Project.....	2
1.3 Research Objectives	3
1.4 Research Methodologies	4
1.5 Scope and Limitations.....	5
1.5.1 Scope.....	5
1.5.2 Limitations	5
1.5.3 Risks	6
1.6 Document Outline	7
2 LITERATURE REVIEW	8
2.1 Cloud computing.....	8
2.1.1 Definition of Cloud Computing.....	8
2.1.2 Advantages of Cloud Computing	9
2.1.3 Service Models - IaaS, PaaS, SaaS, CaaS	10
2.1.4 Cloud Provider, Cloud User and End User.....	12
2.2 Forms of virtualisation	13
2.2.1 Operating system virtualisation and containers	14
2.3 Container-based systems.....	14
2.4 Container orchestration platforms.....	16
2.4.1 Container as a Service	18

2.5	Performance variability in public clouds	19
2.6	Benchmarking	22
2.7	Elasticity	25
2.8	Measuring elasticity and elasticity benchmarks.....	28
2.9	Measuring elasticity in container-based environments	29
2.10	The BUNGEE benchmark.....	29
2.10.1	Phases of the BUNGEE benchmark	30
2.10.2	Metrics captured by the BUNGEE benchmark	31
2.10.3	Technical details of the BUNGEE benchmark.....	33
2.10.4	Data collected by the BUNGEE benchmark	34
3	DESIGN AND METHODOLOGY	37
3.1	AWS Elastic Container Service (ECS)	37
3.2	Experimental setup.....	39
3.2.1	Experiment Summary	39
3.2.2	Extension of the BUNGEE framework to facilitate AWS ECS	40
3.2.3	Modifications to the existing BUNGEE code	42
3.2.4	Network specifications	42
3.2.5	Load driver machine specifications	42
3.2.6	AWS cloud environment setup.....	44
3.2.7	Alternative experimental setup	44
3.2.8	Further configuration	44
3.2.9	DNS issues encountered during initial tests	46
3.3	Statistical methods for evaluation	47
3.3.1	One sample, two tailed t-test	47
3.3.2	Repeated Measures ANOVA.....	48
4	IMPLEMENTATION AND RESULTS	50

4.1	Implementation	50
4.1.1	Observations during the system analysis phase.....	51
4.2	Results.....	55
4.2.1	Descriptive Statistics	55
4.2.2	Container placement on virtual machines.....	57
4.2.3	CPU utilisation of individual containers.....	59
4.2.4	Hypothesis H-A	60
4.2.5	Hypothesis H-B	61
4.2.6	Results in comparison to existing research.....	63
4.2.7	Results of a BUNGEE benchmark run	64
5	ANALYSIS, EVALUATION AND DISCUSSION	66
5.1	Discussion of results	66
5.2	Suggestions for further research	67
6	CONCLUSION	69
	BIBLIOGRAPHY	71
	APPENDIX I – Data Captured by the BUNGEE Benchmark	77
	APPENDIX II – AWS configuration	79
	APPENDIX III – Results of the system analysis - ECS.....	81
	APPENDIX IV – Results of the system analysis - EC2.....	82
	APPENDIX V – Analysis results EC2 only	83
	APPENDIX VI – Comparison virtual & physical load driver	85

LIST OF ABBREVIATIONS

ANOVA.....	<i>Analysis of variance, Analysis of variance</i>
AWS	<i>Amazon Web Services</i>
CaaS.....	<i>Container as a Service</i>
CL	<i>Confidence level</i>
CPU	<i>Central processing unit</i>
DIT.....	<i>Dublin Institute of Technology</i>
DNS	<i>Domain name system</i>
EC2	<i>Elastic Compute Cloud</i>
ECR.....	<i>Elastic Container Registry</i>
ECS	<i>Elastic Container Service</i>
Gbit	<i>Gigabit</i>
HTTP	<i>Hypertext Transfer Protocol</i>
IaaS	<i>Infrastructure as a Service</i>
ID	<i>Identifier</i>
MBps	<i>Megabytes per second</i>
MiB	<i>Mebibyte</i>
ms.....	<i>millisecond</i>
MVC	<i>Model view controller</i>
NIST	<i>National Institute of Standards and Technology</i>
NTP.....	<i>Network time protocol</i>
OS	<i>Operating system</i>
PaaS	<i>11, Platform as a Service</i>
QOS	<i>Quality of service</i>
RAP	<i>Rich ajax applications</i>
RCP.....	<i>Rich client applications</i>
RPS	<i>Requests per second</i>
SaaS	<i>11, Software as a Service</i>
SDK	<i>Software Development Kit</i>
SLA.....	<i>Service level agreement</i>
SLO.....	<i>Service level objective</i>
SUT.....	<i>30, 31, System under test</i>
VM.....	<i>Virtual machine</i>

TABLE OF FIGURES

Figure 2.1: Provisioning and under-provisioning. Source: (Armbrust et al., 2010, p. 54)	10
Figure 2.2: Service models and associated components in cloud computing. Source: (Zhang et al., 2010, p. 9)	12
Figure 2.3: Cloud components and their user types by service model. Source: (Jennings & Stadler, 2015, p. 6)	12
Figure 2.4: Full vs Operating System Virtualisation (adapted from Bauer & Adams, 2012, p. 21)	14
Figure 2.5 Reference architecture using microservices and containers. Source: Microsoft	16
Figure 2.6: Benchmark experiment designs – Source (Abedi & Brecht, 2017, p. 2)	21
Figure 2.7: Phases of the BUNGEE framework	30
Figure 2.8: Accuracy and Timeshare metrics. Source: Herbst et al., 2015, p.48	33
Figure 2.9: Output files and folders generated by BUNGEE system analysis phase	35
Figure 3.1: ECS instances, tasks, container agent. Source: Amazon Web Services Inc.	38
Figure 3.2: BUNGEE ECS code examples	41
Figure 3.3: Elastic Container Service: screenshot of AWS user interface	45
Figure 3.4: Dockerfile to used create docker image	45
Figure 3.5: Bash script to configure instance for ECS and set up chrony NTP implementation	46
Figure 4.1 CPU utilisation of the ECS service during system analysis	51
Figure 4.2: AWS application load balancer metrics	52
Figure 4.3: AWS target group metrics 1	52
Figure 4.4: 4XX an 5XX errors during system analysis phase	53
Figure 4.5: 5XX an 4XX error during the benchmark phase	54

Figure 4.6: Latency by load intensity (SLO met, left. SLO failed, right)	54
Figure 4.7: Mean load intensity (RPS) handled during each system analysis \pm standard error (31 df, $p < 0.05$)	56
Figure 4.8: Mean load (requests/s*resources) handled per resource \pm standard error (31 df, $p < 0.05$)	57
Figure 4.9: Mean additional load handled per new resource \pm standard error (31 df, $p < 0.05$).....	59
Figure 4.10: Benchmark results ECS	65

TABLE OF TABLES

Table 2.1 Stakeholders in cloud systems (Jennings & Stadler, 2015, p. 4).....	13
Table 2.2: Selection of container orchestration platforms.....	17
Table 2.3: Selection of cloud benchmarks.....	23
Table 2.4: Definitions of elasticity	25
Table 2.5: Metrics directly related to elasticity. Source: (Coutinho, Sousa, et al., 2015)	26
Table 2.6: Data captured in BUNGEE measurement phase	35
Table 3.1: Network specifications DIT library	42
Table 3.2: Specifications of the physical load driver machine	43
Table 3.3: Specifications of the virtual load driver machine	43
Table 4.1: Descriptive statistics	57
Table 4.2: Results in wide format for repeated measures ANOVA	61
Table 4.3: Mean load, confidence intervals and boundaries for accepted error	63
Table 4.4: Benchmark run results	65

1 INTRODUCTION

This chapter explains the motivation and subject of the research undertaken for this dissertation. The subject of the dissertation, elasticity measurement, is placed in context with the current cloud computing landscape. A research question is developed to explore if a benchmark tool identified for this dissertation can measure the elasticity of container-based cloud computing environments. Hypotheses are stated, limitations and scope of this research are covered, and risks of this research highlighted.

1.1 Background

Nowadays, many companies provide a wide variety of cloud computing offerings which developers of web services or applications can choose from:

- Virtual servers in the cloud provide practically unlimited computing capacity (Binnig, Kossmann, Kraska, & Loesing, 2009, p. 2).
- Ready to use cloud development platforms let developers write their software without having to provision servers or virtual machines.
- Managed container orchestration frameworks allow developers to place their code into Docker containers (i.e. virtualised operating systems)¹ and run multiple copies of these in an encapsulated and coordinated way. Container orchestration frameworks are used to manage the creation and administration of containers in a cloud system.

The services listed above are offered by a variety of providers such as Google Cloud Platform, Amazon Web Services (AWS), IBM Cloud, Microsoft Azure etc.

The providers promise an unlimited and instant scalability of the above-mentioned offerings. Scalability refers to dynamically adding or removing resources according to the user demand (Islam, Lee, Fekete, & Liu, 2012). But how can the scalability they offer be evaluated and compared? Can the systems really shrink and grow instantly?

¹ A more detailed explanation of container orchestration frameworks and Docker containers can be found in chapter 2.3 “Container-based systems” and 2.4 “Container orchestration platforms”.

These questions can be re-phrased into “How elastic is a system”. Elasticity² is important to businesses who want to meet their service level objectives (SLO). Businesses might want to compare systems by different providers, or just different auto-scaling settings on a platform they have already chosen. They might want to ensure the system adapts to the user demand and always fulfils the SLOs while at the same time not over-provisioning.

Multiple research papers exist establishing metrics for elasticity (Coutinho, Sousa, Rego, Gomes, & Souza, 2015). Several application benchmarks contain some aspect of measuring elasticity (Al-Dhuraibi, Paraiso, Djarallah, & Merle, 2017, p. 8). Apart from these large application benchmarks, a micro-benchmark exists to measure elasticity in isolation: the BUNGEE benchmark (Herbst, Kounev, Weber, & Groenda, 2015). An extensive explanation of the aforementioned benchmarks can be found in chapter 2.

The BUNGEE benchmark was chosen for this dissertation because it is a promising way to measure elasticity in isolation of other factors. It is flexible and easy to use. It has been applied to cloud systems based on virtual machines (Weber, 2014), but has not yet been applied to cloud systems based on the operation of containers.

This dissertation is a proof of concept to determine if the BUNGEE benchmark can measure the elasticity of systems that operate based on containers. Container based systems are further described in 2.3 “Container-based systems”. To the authors best knowledge, this is the only work that attempts to create a mechanism for measuring elasticity in container-based systems or to adapt an existing mechanism to such systems.

1.2 Research Project

This dissertation strives to verify if the BUNGEE micro-benchmark is applicable to container-based cloud environments. A wide variety of systems using containers is commercially available. This research investigates BUNGEE applied to one specific, commercially available container orchestration system: AWS Elastic Container Service (ECS).

² (see 2.7 “Elasticity”)

The first step in verifying the compatibility of the benchmark with ECS is to extend the BUNGEE benchmark to work with ECS. The extended benchmark must then be validated. It must be ensured that it produces reproducible and plausible results.

This benchmark consists of multiple phases. The crucial phase for the benchmark is the so-called system analysis phase. If this phase yields reproducible results, it can be assumed that the other phases of the benchmark will also yield reproducible results. A detailed explanation of this assumption and the different phases of the BUNGEE benchmark can be found in 2.10.1 “Phases of the BUNGEE benchmark”.

To generate a proof of concept of BUNGEE’s compatibility with ECS, an extension was created for the benchmark. To validate that the extension works correctly, the below research questions were formulated.

The main research question is the following:

1. Can the BUNGEE framework reproducibly measure the elasticity of a system built with AWS Elastic Container Service, producing results with no statistically significant difference (CL 95%) during several runs of the BUNGEE system analysis?

A secondary research questions that could follow the first one is:

2. In case the BUNGEE benchmark cannot reproducibly measure the elasticity in the system under test (SUT), what are the causes? Can it be adapted to produce reliable results?

1.3 Research Objectives

To answer research question 1 with yes, and thus verify reproducibility, it is necessary for the benchmark to produce consistent results in the system analysis phase of the benchmark (see 2.10.1 “Phases of the BUNGEE benchmark”). The system analysis produces a file which maps each resource level (number of virtual machines / containers) to the maximum load intensity (requests per second) which this resource level can handle.

For the elasticity benchmark to generate reproducible results, this mapping should be consistent when the system analysis is conducted multiple times on the same system: A

consistent mapping means the same number of resources can always handle the same number of requests per second before failing the service level objectives.

If the mapping is consistent, this enables the benchmark to produce reliable and meaningful results.

If the mapping is different each time, this indicates there must be confounding factors influencing the result, which could hinder the benchmark from producing meaningful results. These confounding factors could be related to the implementation of the benchmark or they could be related to the system under test (SUT).

The following hypotheses have been set:

H-A₀: With a probability of $\geq 95\%$ there is no statistically significant difference between the resource-load curves produced by running BUNGEE's system-analysis several times on the same system.

H-A₁: With a probability of $\geq 95\%$ there is a statistically significant difference between the resource-load curves produced by running BUNGEE's system-analysis several times on the same system

To make the results of this investigation comparable to the results of research previously conducted (Weber, 2014), additionally the following hypotheses have been set:

H-B₀: The error of the system analysis phase is smaller than 5% of with a confidence level of 95%.

H-B₁: The error of the system analysis phase is larger than 5% with a confidence level of 95%.

1.4 Research Methodologies

To investigate the research objectives stated in 1.3, the BUNGEE benchmark was extended to interface with the AWS Elastic Container Service (ECS). To collect meaningful results, a controlled experiment was conducted, running the system analysis phase of the benchmark 32 times.

To answer *H-A* a “repeated measures analysis of variance” (ANOVA) was performed to determine if a significant difference can be detected between the resource-load curves of each run. To answer *H-B*, the confidence intervals for the mean maximum load determined at each resource level were computed. It was determined if those deviate

more than 5% from the sample mean, meaning they exceed the acceptable level of variation.

1.5 Scope and Limitations

This section briefly outlines the scope of this work and explores any limitations that constrain this research and its results.

1.5.1 Scope

This research is aimed at understanding if the BUNGEE benchmark can be applied to cloud systems that make use of containers and container orchestration platforms. The goal is not to measure the elasticity of a specific cloud platform or to compare two cloud platforms. The aim is to verify if the benchmark can produce reliable results on container-based systems.

Based on this, the following points are in scope:

- Extending the BUNGEE benchmark to work with AWS Elastic Container Service.
- Conducting an experiment to verify if the developed extension works and produces reproducible results.
- If hypotheses $H-A_0$ and $H-B_0$ are rejected, starting initial investigations into research question 2, exploring the reasons why the results are not consistently reproducible.

1.5.2 Limitations

After initial experiments running in the researcher's home network, it became evident that the load driver machine cannot run there. The Virgin Media 300Mbps home broadband could not cope with the number of DNS requests made by the load driver machine. Therefore, experiments were conducted in the DIT library during opening hours of the computer room, which has a connection speed of 13MB/s. Each computer in the library has one primary and two alternative DNS servers. Details about the experiments conducted in the researcher's home network and the problems encountered can be found in chapter 3.2.9.

Initial tests with the BUNGEE benchmark showed that one run of the system analysis takes between 4 and 6 hours, limiting the possible number of experiments run from the DIT library to one per day. This initially lead to a small sample size, as only 10 runs of the experiment could be conducted in the DIT library. Fortunately, the University of Würzburg later provided a virtual machine in their private cloud environment, so that additional 22 runs could be conducted. Conducting the experiment under two different experimental conditions might impact the overall results. However, a Man-Whitney-U test was conducted (see Appendix VI) and no statistically significant differences between the results conducted in DIT and in the private cloud of the University of Würzburg were found.

Another point to note is that AWS ECS allows the cloud user to set up many different configurations and settings. Due to the logistic restrictions mentioned above, it was not possible to run the experiment with different settings. One configuration had to be chosen and used for the experiment. Results might be different with other configuration settings, which could not be explored in this dissertation.

1.5.3 Risks

This research assumes that if the system analysis phase produces reliable results, the BUNGEE benchmark is likely applicable and valid for measuring elasticity in container-based cloud environments.

This claim is based upon the assumption that the measurement conditions are the same during the system analysis phase and the benchmark phase. However, this is not necessarily guaranteed as the auto-scaling settings for both the Elastic Container Service and the virtual machines which host the service are enabled during the benchmark phase but disabled during the system analysis phase.

This change in settings could introduce unanticipated variables into the process. AWS ECS is a “grey-box” system with limited insights into its functionality and parameters. It is possible that enabling auto-scaling settings modifies the load processing capacity of the system, which would render benchmark results invalid.

A second risk to consider is that factors unrelated to the Elastic Container System might influence the experiment, such as network latency, resource contention on the virtual machines or temporary issues in the AWS system. This could cause the system analysis

to yield unstable results and would lead to rejection of the null hypotheses, when they should have been accepted, causing a type 1 error. To avoid this error, literature research on performance variability in the cloud has been conducted, network and load driver specifications have been documented and further research in private cloud environments where all parameters of the experiment can be fully controlled should be carried out.

1.6 Document Outline

The remainder of this dissertation is structured as follows:

Chapter 2 “Literature Review” introduces the reader to important concepts and terminology. An introduction into the topic of cloud computing is given. Cloud computing is defined and its service models (IaaS, PaaS, CaaS, SaaS) introduced. The different actors in cloud computing are determined: Cloud Provider, Cloud User and End User. The concepts of operating system virtualisation and container technology are covered, an overview of container orchestration platforms given. An introduction to benchmarking is provided, the term elasticity defined and the literature into elasticity surveyed. The BUNGEE benchmark is described, and some technical details highlighted.

Chapter 3 “Design and Methodology” describes the experiment conducted for this research, specifies all system and setup configurations and introduces the statistical methods used for evaluation.

Chapter 4 “Implementation and Results” captures any observations made during the implementation of the experiment and lists the results, descriptive statistics and provides several graphs to visualise the results. The hypotheses are tested based on the results obtained in the experiment. The results of one full benchmark run are also presented in this chapter, to confirm that the whole benchmark is operational, not only the system analysis phase.

Chapter 5 “Analysis, Evaluation and Discussion” discusses and attempts to interpret the results, exploring possible causes for the encountered fluctuation of results. This chapter also compares and contrasts the results with findings encountered by other researchers. Suggestions for future research are captured.

Chapter 6 “Conclusion” summarises the findings and lists the contributions of this work.

2 LITERATURE REVIEW

This chapter first covers several fundamental concepts which are required to understand the goal and purpose of this work. Fundamental topics are:

- The definition of cloud computing, it's advantages and service models.
- The actors in cloud computing environments.
- Virtualisation and operating system virtualisation. Operating system virtualisation is the concept underlying containerisation.
- Containers, container orchestration platforms and Containers as a Service (CaaS).

Further, a variety of benchmarking techniques that can be used in cloud systems are introduced. The topic elasticity is covered in detail, along with techniques to measure elasticity. Lastly, the BUNGEE elasticity benchmark, which is subject of this dissertation, will be covered, some technical details explained, and the data collected by the benchmark listed.

2.1 Cloud computing

Cloud Computing is an increasingly important subject for researchers and companies today, as computing resources can be rented “on-the-fly” from cloud providers, giving companies an unprecedented flexibility when providing applications to their users. According to Gartner, an independent IT research and advisory company, the cloud computing market is projected to almost double in value from 219.6 billion USD in 2016 to 411.4 billion USD in 2020³.

This section will define cloud computing, highlight its core aspects, explain the service models of cloud computing (IaaS, PaaS and SaaS).

2.1.1 Definition of Cloud Computing

In 2010, Armbrust et al. (2010, p. 50) found that the definition of cloud computing varies between authors. The authors state that cloud computing refers to two things: Software delivered as service through the internet as well as the hardware and systems used to

³ Gartner. (2017, October). Gartner Forecasts Worldwide Public Cloud Services Revenue to Reach \$260 Billion in 2017. Retrieved March 13, 2018, from <https://www.gartner.com/newsroom/id/3815165>

provide these services. A year later the National Institute of Standards and Technology (NIST) released a special publication with a definition of Cloud Computing which is widely cited since.

Their definition states:

“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.” (Mell & Grance, 2011, p. 2)

From this definition it can be derived that an important aspect of cloud computing is the “on-demand” provisioning. Resources are rapidly available and do not have to be acquired, configured and provisioned in a lengthy process.

2.1.2 Advantages of Cloud Computing

Advantages of cloud computing include according to Zhang, Cheng, & Boutaba (2010, p. 1):

- No initial investment for infrastructure.
- A lower operational cost.
- The possibility of designing highly scalable systems.
- Easy access.

The possibility of provisioning and deprovisioning computing resources within minutes enables the development of new software services rapidly and cost efficiently (Armbrust et al., 2010, p. 50).

The need for rapidly provisioning resources arises from the fluctuating nature of traffic to most services. Before cloud computing was available, enterprises had to provision enough resources to meet the demand at peak times, leading to resources being underutilised at non-peak times (Baun, Kunze, Nimis, & Tai, 2011) as cited in (Bellenger et al., 2011, p. 2).

The load and resources of a system which is equipped to handle peak loads but is therefore over-provisioned at certain times (yellow), are shown in Figure 2.1a).

In contrast, Figure 2.1b) displays load and resources of a system which is under-provisioned and therefore unable to provide sufficient resources at peak times (yellow) while still being over-provisioned at non-peak times.

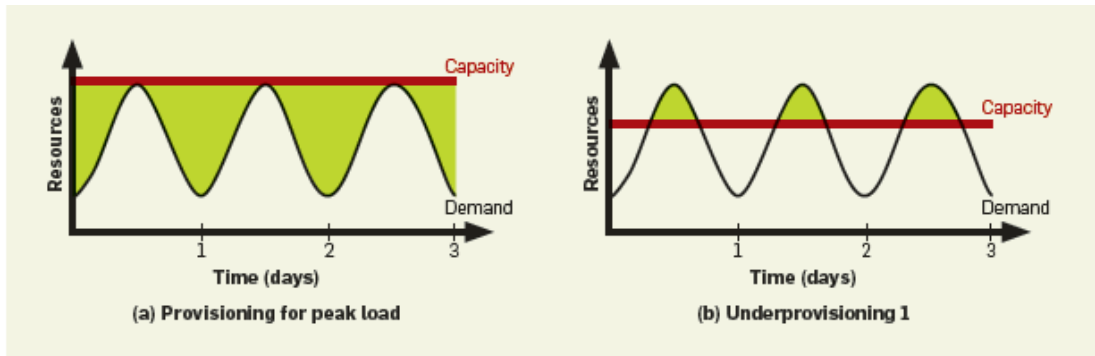


Figure 2.1: Provisioning and under-provisioning. Source: (Armbrust et al., 2010, p. 54)

Cloud computing introduces the ability to provision computing resources such as servers, virtual machines and application instances as needed and pay for only the amount of computing resources used.

Armbrust et al. (2010, p. 53) contend that using cloud resources is not cost saving compared to purchasing and provisioning own hardware, but that the mitigation of the risk of under-provisioning provides enough benefit to justify the (at that time) higher cost.

The concept of scaling rapidly in response to demand has also been called “Elasticity”. Mell & Grance (2011, p. 2) list elasticity as one of the essential characteristics of cloud computing. Elasticity describes how well and how fast cloud systems adapt to increases and decreases in workload. Elasticity is defined and explored further in chapter 2.7 “Elasticity”.

2.1.3 Service Models - IaaS, PaaS, SaaS, CaaS

Mell & Grance (2011, p. 3) formalise different service models used in cloud computing:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)

Even though the line between these services is not always clear (Armbrust et al., 2010, p. 2), this categorisation gives a good overview of the type of services offered by providers of public clouds. These services are briefly described as:

Software as a Service (SaaS)

Software as a Service (SaaS) is defined as applications provided to a consumer without the consumer having control over the underlying infrastructure or application capabilities (Mell & Grance, 2011, p. 2). Examples include: Gmail, Facebook, Salesforce, WorkDay.

Platform as a Service (PaaS)

Platform as a Service (PaaS) facilitates the creation and deployment of applications and software services to the end user, without the consumer controlling the underlying cloud infrastructure e.g. servers, operating systems and storage (Mell & Grance, 2011, p. 2). Examples include: Amazon Elastic Beanstalk, Heroku, Google App Engine.

Infrastructure as a Service (IaaS)

Facilitates the provisioning of computing resources e.g. processing, storage, networks. In this model, the consumer can deploy any software including operating systems. The consumer has no control over the underlying technical infrastructure but does control operating systems, storage and some networking components (Mell & Grance, 2011, p. 2). Examples include Amazon AWS EC2, Google Compute Engine, Microsoft Azure.

Figure 2.2 gives a breakdown of the different components which SaaS, PaaS and IaaS services consist of.

Containers as a Service (CaaS)

The term “Containers as a Service” (CaaS) has been introduced by the industry to describe managed container orchestration services. The term CaaS occurs in the scientific literature, but a widely accepted definition does not seem to be in use yet. Containers are explored more fully in Section 2.3 “Container-based systems”. Section 2.4 “Container orchestration platforms” will describe container orchestration platforms and attempt to define the term “Containers as a Service (CaaS)”.

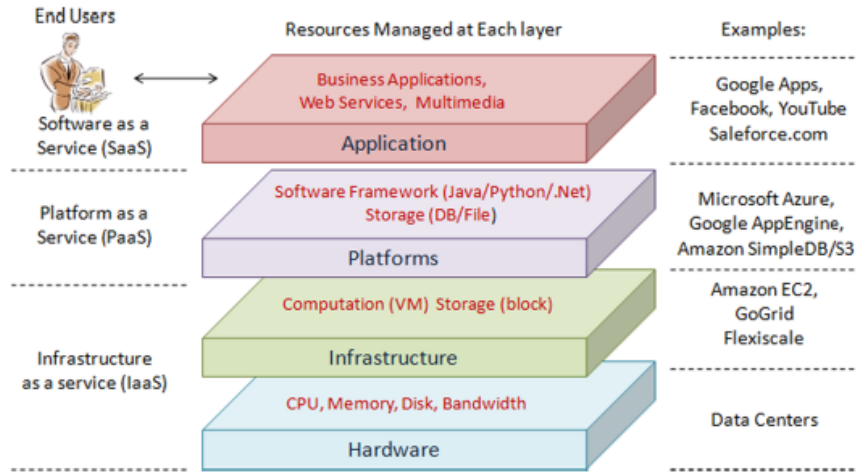


Figure 2.2: Service models and associated components in cloud computing. Source: (Zhang et al., 2010, p. 9)

2.1.4 Cloud Provider, Cloud User and End User

Jennings & Stadler (2015, p. 4) introduce a helpful terminology to understand the roles in the different service models: The End User, the Cloud User and the Cloud Provider. Figure 2.3 depicts which parts are usually managed by which category of users. Figure 2.3a depicts an IaaS system, Figure 2.3b a PaaS system and Figure 2.3c a SaaS system.

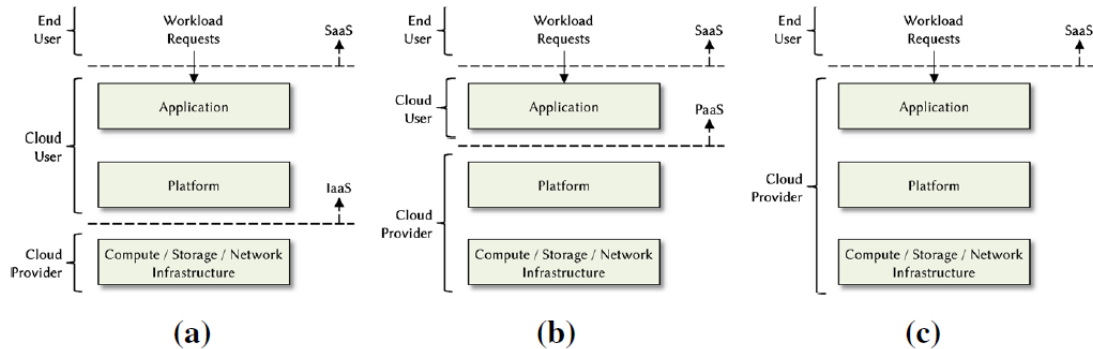


Figure 2.3: Cloud components and their user types by service model. Source: (Jennings & Stadler, 2015, p. 6)

Table 2.1 defines the terms “cloud user”, “end user” and “cloud provider”. This terminology will be used in this dissertation going forward.

Table 2.1 Stakeholders in cloud systems (Jennings & Stadler, 2015, p. 4)

Term	Definition
End user	The actual user of the application. Generates the workloads / uses the application. Does not contribute to resource management.
Cloud user	Creates applications for the end users using a public cloud. Is responsible for scaling according to end user demand.
Cloud provider	Manages systems to provide a public cloud to Cloud User (IaaS, PaaS or SaaS).

Which service model a cloud user selects depends on the degree of control of the underlying processes the cloud user needs. In an IaaS System, the cloud provider needs to interact with and administer the operating system while in a PaaS system, these components are abstracted away. The trade off in this case is, that the PaaS system might be restricted to certain programming languages or configurations supported by the PaaS provider (Rosenberg & Mateos, 2011, p. 16).

2.2 Forms of virtualisation

Virtualisation can be defined as follows:

“Virtualization is the logical abstraction of physical assets, such as the hardware platform, the operating system (OS), storage devices, data stores, or network interfaces.” (Bauer & Adams, 2012, p. 16).

Per above definition, different types of resources can be virtualised such as networks, memory, storage or processors. The following relates to server virtualisation.

Bauer & Adams (2012, p. 18) distinguish between full virtualisation, hardware assisted virtualisation, paravirtualisation and operating system virtualisation.

In the context of this research, full, hardware-assisted and paravirtualisation can be regarded as similar: A piece of software called hypervisor runs on a computer and manages the virtualisation (Figure 2.4, left). The hypervisor manages the host system’s resources and emulates one or more guest operating systems running on emulated hardware. The guest operating systems can then run applications. This type of virtualisation can be used simply by installing a hypervisor on any computer. It is also typically used on cloud IaaS platforms.

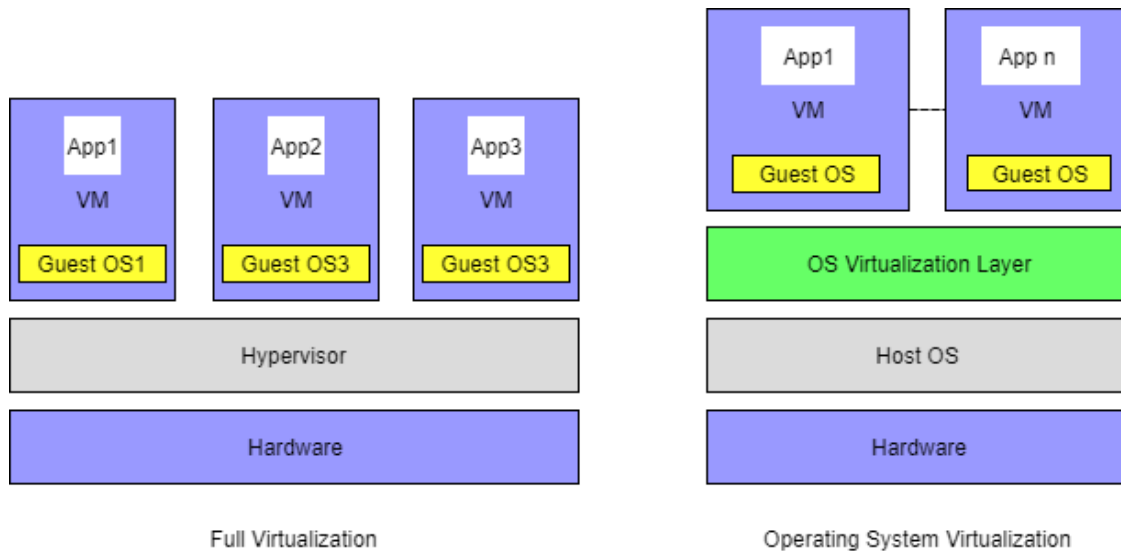


Figure 2.4: Full vs Operating System Virtualisation (adapted from Bauer & Adams, 2012, p. 21)

2.2.1 Operating system virtualisation and containers

In operating system virtualisation, a virtualisation layer runs on the host OS. The virtualisation layer manages isolated instances called containers. A container encapsulates an OS process which has limited central processing unit (CPU) and memory resources assigned to it (Khan, 2017, p. 44). It can have its own file system, libraries and other components (Bauer & Adams, 2012, p. 22).

Figure 2.4 (right) depicts a system with OS virtualisation. An important point is that in OS virtualisation the host OS and the guest OS must be identical, while in traditional virtualisation, they can be different. The reason is that in operating system virtualisation, all guest systems use the host operating system as their base.

2.3 Container-based systems

Operating system virtualisation has drawn increased attention since the Docker open source project was launched in 2013 (Casalicchio & Perciballi, 2017). Docker facilitates the automated deployment of applications inside of containers (Bernstein, 2014, p. 82). Docker containers can be simple, virtual operating systems or can be set up to contain preinstalled and configured applications (Bernstein, 2014, p. 82), which can then be easily deployed and scaled in diverse systems.

Docker is not the only, but the most common container management software. Alternatives include rkt⁴ and LXC Linux Containers⁵.

Advantages of using container-based systems are (Casalicchio, 2017):

- Lower overhead: They use less of the host system's resources compared to virtual machines.
- They encapsulate applications, enabling each application to have its own set of libraries, avoiding incompatibilities (also called the "Dependency Hell Problem").

Disadvantages of using container-based systems:

- A host OS can only host containers with the same operating system as itself.
- Security concerns: It is not possible to completely isolate the containers from each other, except for running one container per host (Bernstein, 2014, p. 83).
- The technology is relatively new and has not matured yet.

The availability of containers led to a shift in the way application architectures are designed (Pahl, 2015, p. 28). Containers often host so called "microservices". An application consists of multiple microservices, stateless services that communicate with messages between each other. This concept lets application architectures shift away from monolithic structures, towards a Service Oriented Architecture (SOA) with loosely coupled components (Pahl, 2015, p. 28).

Figure 2.5 shows a reference architecture published by Microsoft where various microservices, encapsulated in containers, form an application.

In this reference architecture, the Model View Controller (MVC) component lives inside a container, as well as several other microservices such as the basket microservice, the ordering microservice etc. All these services can run independently from each other, communicating via messages. This guarantees that different versions of libraries needed by the components will not create conflicts as it could happen if all components were running on one virtual machine. It also ensures that each microservice has its own allocated quota of resources that the other services cannot impact.

⁴ CoreOS. (2018). CoreOS. Retrieved March 22, 2018, from <https://coreos.com/rkt/>

⁵ LXD. (2018). Linux Containers - LXD - Introduction. Retrieved March 22, 2018, from <https://linuxcontainers.org/lxd/>

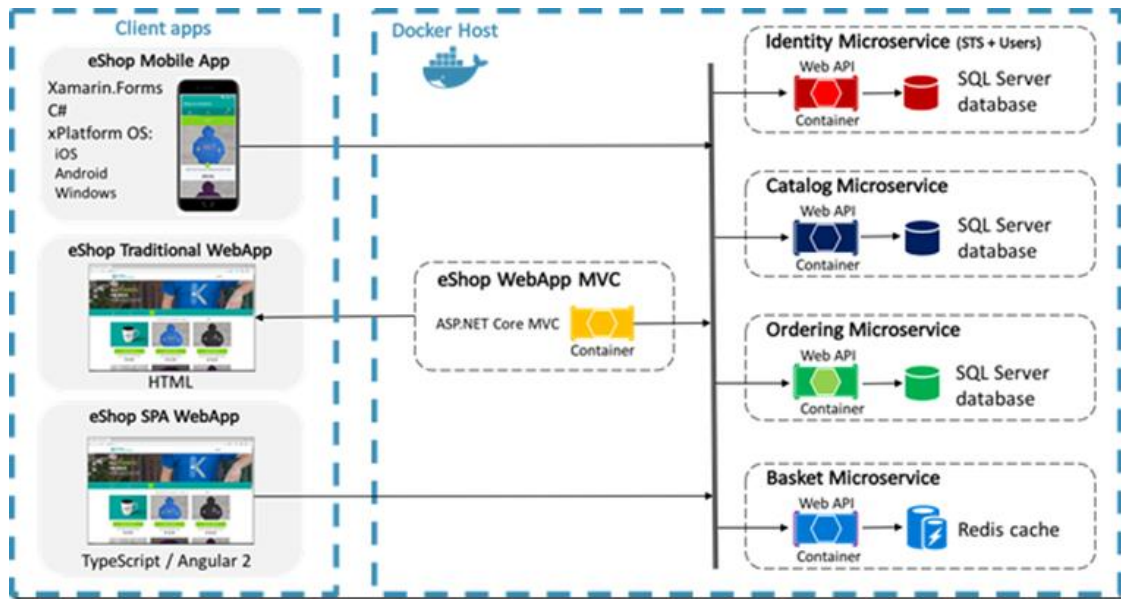


Figure 2.5 Reference architecture using microservices and containers. Source: Microsoft⁶

2.4 Container orchestration platforms

In the previous sections, containers have been introduced and their role in microservice-based architectures explained. This section will cover how containers can be managed with container orchestration platforms.

While it is possible to create an application architecture with only one container per service, in most scenarios the application will need to be scaled horizontally, which requires having multiple instances of the same container. This makes a container orchestration system necessary.

A container orchestration platform is defined as “a system that provides an enterprise-level framework for integrating and managing containers at scale” (Khan, 2017, p. 44). Container orchestration platforms are needed to ensure the specified number of containers is running and that the containers can communicate with each other and the outside.

Typical features of a container orchestration platform include (Khan, 2017, p. 44):

⁶ Microsoft. (2017, May 10). Free eBook/Guide on ‘.NET Microservices – Architecture for Containerized .NET Applications’ – Cesar de la Torre [Microsoft] – BLOG. Retrieved March 22, 2018, from <https://blogs.msdn.microsoft.com/cesardelatorre/2017/05/10/free-ebookguide-on-net-microservices-architecture-for-containerized-net-applications/>

- Managing the cluster state and container scheduling.
- Ensuring high availability and fault tolerance.
- Managing security.
- Enabling service discovery.
- Facilitating continuous deployment.
- Facilitating monitoring and governance.

A variety of container orchestration platforms are available. Some are open source platforms that the cloud users can install themselves, others are commercially developed systems. Table 2.2 shows a selection of current container orchestration platforms.

Table 2.2: Selection of container orchestration platforms

Provider	Platform name	Description
Cloud native computing foundation	Kubernetes	Popular open source container orchestration system. Originally developed by Google (Khan, 2017, p. 44).
Mesosphere	Mesosphere	Container orchestration system based on the open source project Apache Mesos (Khan, 2017, p. 44)
Docker Inc.	Docker Swarm	The container orchestration mode of the popular container management system Docker ⁷ .
Google	Google Kubernetes Engine, formerly named Google Container Engine ⁸	Managed container orchestration system provided by Google. Developed based on an internal system called “Borg” which introduced container orchestration over 10 years ago ⁹ .

⁷ Docker Inc. (2018, March 21). Swarm mode overview. Retrieved March 22, 2018, from <https://docs.docker.com/engine/swarm/>

⁸ Denniss, W. (2017, November 13). Introducing Certified Kubernetes (and Google Kubernetes Engine!). Retrieved March 23, 2018, from <https://cloudplatform.googleblog.com/2017/11/introducing-Certified-Kubernetes-and-Google-Kubernetes-Engine.html>

⁹ McLuckie, C. (2016, July 22). From Google to the world: the Kubernetes origin story. Retrieved March 23, 2018, from <https://cloudplatform.googleblog.com/2016/07/from-Google-to-the-world-the-Kubernetes-origin-story.html>

Provider	Platform name	Description
Amazon	Elastic Container Service	Container Orchestration Service which supports Docker containers ¹⁰ .
Microsoft	Azure Container Service	Managed Kubernetes Service ¹¹ . Microsoft offers the possibility of deploying alternative container orchestration systems such as Docker or DC/OS ¹² .

2.4.1 Container as a Service

Where do containers and container orchestration platforms fall in the traditionally referenced service models IaaS, PaaS and SaaS? If cloud users set up their own container orchestration service on top of virtual machines, one could argue they are using IaaS. But what about managed container orchestration services such as Google Kubernetes Engine, Azure Container Service or Amazon Elastic Container Service? Some of them run on top of virtual machines and can be considered an additional functionality on top of IaaS, but virtual machines do not necessarily need to be involved.

The industry has started to use the term “Container as a Service” (CaaS)¹³. Some scientific publications also use this term with varying or no definitions. The term CaaS has been used to describe PaaS systems that use container solutions “under the hood” (Kratzke & Peinl, 2016), for example Amazon Elastic Beanstalk uses Amazon Elastic Container Service “under the hood”¹⁴.

¹⁰ Amazon Web Services, Inc. Amazon ECS Features - run containers in production. Retrieved April 2, 2018, from <https://aws.amazon.com/ecs/features/>

¹¹ Monroy, G. (2017, October 24). Introducing AKS (managed Kubernetes) and Azure Container Registry improvements. Retrieved March 23, 2018, from <https://azure.microsoft.com/en-us/blog/introducing-azure-container-service-aks-managed-kubernetes-and-azure-container-registry-geo-replication/>

¹² DC/OS. The Definitive Platform for Modern Apps. Retrieved March 23, 2018, from <https://dcos.io/>

¹³ Burns, B. (2017, February 21). Containers as a Service, the foundation for next generation PaaS. Retrieved March 23, 2018, from <http://blog.kubernetes.io/2017/02/caas-the-foundation-for-next-gen-paas.html>

¹⁴ Amazon Web Services, Inc. Amazon ECS Frequently Asked Questions - run containers in production. Retrieved March 23, 2018, from <https://aws.amazon.com/ecs/faqs/>

CaaS has also been described as the middle layer between IaaS and PaaS (Piraghaj, Dastjerdi, Calheiros, & Buyya, 2015, p. 368). This is the case AWS's offering, where virtual machines must be assigned as hosts for the containers.

Although container orchestration systems usually seem to be running on top of virtual machines, this does not necessarily have to be the case. As discussed in chapter 2.2, one advantage of operating system virtualisation is reduced overhead compared to virtual machines. Scenarios could be envisioned in which the container orchestration platform runs on "bare metal" (physical servers) directly.

In the absence of any strong definitions for CaaS in the literature, the following definition is suggested by the author of this dissertation. It will be used in the remainder of this dissertation.

Container as a Service (CaaS) is an offering with which a cloud user can benefit from automatised deployment, operation and scaling of clusters of containers without having to install and maintain a container orchestration software.

2.5 Performance variability in public clouds

One factor that could affect the outcome of this research negatively is the performance variability in public clouds. Performance variability means that with the same number and configuration of resources, a system in the cloud will generate different performance metrics without obvious cause. This chapter provides a fundamental understanding of performance variability in public clouds and an overview of the literature that exists to date.

Research has shown that performance of cloud resources by most cloud providers fluctuates in daily or yearly patterns (Iosup, Yigitbasi, & Epema, 2011, p. 1)

This performance variability is caused by various factors such as virtualisation overhead and resource time sharing (Iosup et al., 2011, p. 1). The degradation of performance due to resource time sharing has also been called "resource contention" in the literature (Anwar, Cheng, & Butt, 2016)

Resource contention and performance variability in private and public clouds have been researched in the recent years. Iosup et al. (2011) investigated several cloud services, among those the following AWS services:

- Elastic Compute Cloud (EC2)
- S3 Storage Service (S3)
- Simple Queue Service (SQS - message queuing and synchronisation)
- Simple DB (SDB - database)
- Flexible Payments Service (FPS)

The authors found that the performance of all these services fluctuates according to one or more time patterns and shows special behaviours at certain times (Iosup et al., 2011, p. 1).

Other researchers have developed resource freeing attacks. They found that one tenant using a VM on a shared physical machine can intentionally hijack resources from another tenant who has his/her VM on the same physical hardware in a cloud environment (Varadarajan, Kooburat, Farley, Ristenpart, & Swift, 2012). The researchers could improve benchmark performance by 13% when using AWS EC2 instances (Varadarajan et al., 2012, p. 1) and performing the resource freeing attacks.

Leitner & Cito (2016) conducted detailed research into the performance variability of Amazon Web Service and Google Cloud Platform. They break down performance variation by instance type and workload type (CPU bound or IO bound). Their research indicates that inter-instance performance variability in CPU bound tasks is mostly due to the differences in underlying hardware, which on AWS affects the EC2 instance types m1.small and t1.micro. M1.small instances are the ones chosen for comparability reasons for this research.

Once the researchers controlled their analysis for differences in underlying hardware, the inter-instance variability of EC2 instances was low for CPU bound workloads. The variability was high for IO-bound workloads (Leitner & Cito, 2016, p. 10). This means that if choosing an instance type of t1.micro or m1.small, there is a higher chance of experiencing performance variability for CPU intensive workloads than when choosing other instance types.

The high performance variability of t1.micro instances is partially explained by a “bursting” feature (Leitner & Cito, 2016, p. 12), which allows a virtual machine to use more resources of its underlying host, if the resources are available.

Dealing with the performance variability in experiments and benchmarks conducted in cloud computing environments is a difficult task. In experiments with Amazon EC2,

researchers found that the Unix Benchmark Utility (Ubench)¹⁵ run in a “Multiple Consecutive Trials Design” on the same system shows up to 38% performance difference between the system and itself (Abedi & Brecht, 2017, p. 1). In their experimental design, two identical cloud setups A and B were compared, running the same benchmark 20 consecutive times with setup A and 20 consecutive times with setup B (Abedi & Brecht, 2017, p. 2). Since they were identical setups, there shouldn’t have been any difference.

Their results show that when comparing two systems and conducting the same experiments multiple times, statistical evaluations at a 95% confidence interval can lead to incorrect conclusions due to the inherent variability in cloud computing environments.

Abedi & Brecht (2017) suggest designing experiments using “Randomised Multiple Interleaved Trials”, where benchmark runs are randomly interleaved. Figure 2.6 shows the experiment designs analysed by the researchers. If three systems are benchmarked, a single trial design would run each benchmark once (A), a multiple consecutive trials design would run it several times consecutively (B). Interleaved trials would execute the different benchmark runs in an interleaved fashion, either in an ordered (C) or a random way (D).

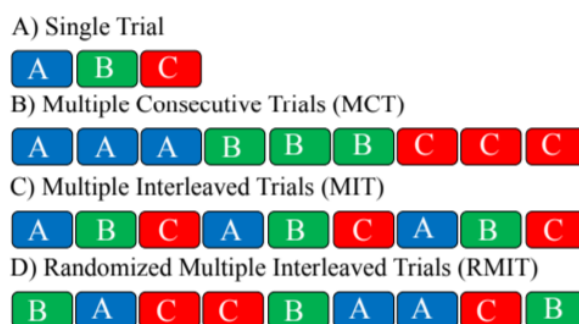


Figure 2.6: Benchmark experiment designs – Source (Abedi & Brecht, 2017, p. 2)

For this research, interleaved randomised trial design unfortunately could not be used for logistic reasons and for the fact that no two systems were benchmarked. A multiple

¹⁵ ubench(8) - Unix Benchmark Utility. Retrieved June 4, 2018, from <https://www.gsp.com/cgi-bin/man.cgi?section=8&topic=ubench>

interleaved trial design is recommended for future measurements in public cloud systems using the BUNGEE benchmark.

2.6 Benchmarking

As this dissertation is concerned with extending an existing benchmark to a CaaS system, a brief introduction to the subject of benchmarking in cloud systems is given in this section.

With the shift to cloud systems, benchmarks had to be re-developed and the requirements re-thought. This area of work is still relatively new, with Folkerts et al. (2012, p. 1) being the first to write about the subject in 2012. They list several challenges that the cloud inherently poses on the development of a benchmark. Some of them are:

- Whether price or performance should be included in the benchmark.
- How the elasticity in a cloud system can be measured.
- How the scaling boundaries of a system can be tested when cloud systems have practically unlimited scalability.
- How the performance variability in cloud systems affects the repeatability of benchmarks.

(Folkerts et al., 2012, pp. 9–15)

Since then, several benchmarks have been proposed in the literature. They attempt to measure one or multiple aspects of a system, such as storage, computing performance, scaling or cost (Vazquez, Krishnan, & John, 2014, p. 3).

V. Kistowski et al. (2015) list the following criteria for a good benchmark:

- Relevance
- Reproducibility
- Fairness
- Verifiability
- Usability

In this research, the BUNGEE benchmark is extended, therefore emphasis is placed on maintaining the reproducibility of the benchmark despite the extension. The other points were already considered when originally developing the BUNGEE benchmark and are unlikely to be significantly affected by the extension to ECS.

Li et al. (2013, p. 14) have used the below terminology to distinguish between different types of benchmarks:

- Micro-benchmark: A simple program that attempts to measure a particular aspect of a cloud service.
- Synthetic benchmark: A program used to represent operations and workload of a typical application, but which was specifically created for the benchmark.
- Application benchmark: A real world application deployed to the cloud.

Two organisations have made it their goal to provide several comprehensive benchmarks for various types of applications: The Standard Performance Evaluation Corporation (SPEC)¹⁶ and the Transaction Processing Performance Council (TPC)¹⁷. These organisations offer application benchmarks for server side java, database management systems, web servers and many more (Kounev, 2007, p. 3). These benchmarks are full applications that can be deployed to and executed on IT systems.

Table 2.3: Selection of cloud benchmarks

Authors	Benchmark
(Varghese, Akgun, Miguel, Thai, & Barker, 2014)	Benchmarks virtual machines without having to run a workload, by implementing a weighted ranking mechanism. Goal: Find the most suitable virtual machine for a given application. No elasticity metrics included.
(Cooper, Silberstein, Tam, Ramakrishnan, & Sears, 2010)	YCSB: Benchmark that tests cloud data serving systems such as BigTable, PNUTS, Cassandra, HBase, Azure, CouchDB, SimpleDB. Measures elasticity.
(Moldovan, Copil, Truong, & Dustdar, 2013)	MELA: Benchmark that allows Cloud Users to evaluate the financial aspects of elasticity.

¹⁶ SPEC - Standard Performance Evaluation Corporation. Retrieved April 5, 2018, from <https://www.spec.org/>

¹⁷ TPC-Homepage V5. Retrieved April 5, 2018, from <http://www.tpc.org/>

(A. Li, Yang, Kandula, & Zhang, 2010)	CloudCmp: Measures several performance metrics of a cloud service by means of running three reference applications. Establishes metrics related to computing capacity, data storage, intra- and wide area network. Metrics for computing capacity are: benchmark finishing time, cost per benchmark, and scaling latency (A. Li et al., 2010, p. 7). This is a comprehensive benchmark suite, but it doesn't establish an exact measure for elasticity, although it measures scaling latency.
(Huang, Huang, Dai, Xie, & Huang, 2010)	HiBench: Benchmark to evaluate components of the Hadoop framework (data storage) (Vazquez et al., 2014, p. 4)
(Ferdman et al., 2012)	CloudSuite: Benchmark suite consisting of various other benchmarks which examine data serving, media streaming, web hosting, web search and some other applications. It does not contain any metrics specific to elasticity (Vazquez et al., 2014).

The TPC benchmarks can be freely downloaded¹⁸ while the SPEC benchmarks are partially available for free for non-commercial organisations and partially available for a fee.

The SPEC Cloud benchmark addresses the topic elasticity. The SPEC Cloud benchmark makes use of two other benchmarks, the HiBench and YCSB (see also Table 2.3), and wraps them in an interface (SPEC, 2016, p. 9,12). It computes 8 metrics, one of which is elasticity. The elasticity is expressed in percent (SPEC, 2016, p. 21).

From the benchmark documentation, it does not become entirely clear how the elasticity metric is computed. The metric computed seems to be closer related to scalability than to elasticity. The non-commercial fee for the SPEC cloud benchmark is 500\$ and could therefore not be practically evaluated in this dissertation.

Table 2.3 lists several other cloud benchmarks. Since this work is concerned with measuring elasticity, the factor elasticity was of interest when looking at available cloud

¹⁸ TPC - Current Specifications. Retrieved April 5, 2018, from http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp

benchmarks. Some benchmarks measure elasticity, but those either measure the elasticity of database systems or they measure elasticity based on cost.

2.7 Elasticity

Section 2.1.2 “Advantages of Cloud Computing” has already touched on the importance of cloud systems adapting to spikes in traffic. How well and how fast systems adapt to those increases in load is captured by the term “elasticity”. Despite being named as one of the main characteristics of cloud computing (Mell & Grance, 2011), elasticity is still a relatively new research topic with a lack of detailed analysis in the literature (Coutinho, de Carvalho Sousa, Rego, Gomes, & de Souza, 2015, p. 1).

Various definitions for elasticity in cloud systems have been proposed. Coutinho, de Carvalho Sousa, et al. (2015) conducted a literature survey and found 9 different definitions. Most of the definitions included the concept of scalability¹⁹. Some of them include a concept of timing or speed of adapting to changes in workload (Coutinho, de Carvalho Sousa, et al., 2015).

Table 2.4: Definitions of elasticity

Source	Definition
Mell & Grance, (2011, p. 2) NIST definition	Rapid elasticity: “Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time”.
N. Herbst, Kounev, & Reussner (2013, p. 24)	“Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.”
Cooper, Silberstein, Tam, Ramakrishnan, & Sears (2010, p. 144)	“Elasticity means that we can add more capacity to a running system by deploying new instances of each component, and shifting load to them”

¹⁹ Scalability is the ability to adapt to increased workload by adding a proportional amount of resources (Islam, Lee, Fekete, & Liu, 2012).

Muñoz-Escoí & Bernabéu-Aubán (2017, p. 3)	“Broadly defining, elasticity is the capability of delivering preconfigured and just-in-need virtual machines adaptively in a cloud platform upon the fluctuation of the computing resources required. Practically it is determined by the time needed from an underprovisioning or overprovisioning state to a balanced resource provisioning state.”
Jennings & Stadler, (2015, p. 5)	“The ability to immediately make available additional resources to accomodate [sic] demand surges and release them whenever demand abates.”

Table 2.4 lists the most relevant definitions of elasticity. The definition which will be considered for this work is the one proposed by N. Herbst, Kounev, & Reussner (2013)

“Elasticity is the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.” (Herbst et al., 2013, p. 24)

From this definition, we can conclude that a system with optimal elasticity spends as little time as possible in an over-provisioned or under-provisioned state. Spending time in an under-provisioned state would impair the system’s operation and possibly violate the service level objectives (SLOs). Spending time in an over-provisioned state would either leave existing resources unused or the system provider would have to pay for renting unused resources.

Table 2.5: Metrics directly related to elasticity. Source: (Coutinho, Sousa, et al., 2015)

Group	Metrics
Allocation Capacity	Resource allocation, available supply, capacity, capacity increase, computation capacity, maximum service capacity, service available, total capacity of infrastructure
Cost	Cost/performance rate, cost bandwidth, effectiveness of time and cost (\$ hours/instances), migration cost, total cost of deployment, total price of infrastructure
QoS	% Violations, performance gain, SLA

Resource utilization	% Utilization, computing resource utilization (CRUM), demand, idleness, increase of idleness, number of over provisioned virtual machines, number of under provisioned virtual machines, number of virtual machines, over provisioning rate, over utilization, performance resource ratio (PRR), server number average, under utilization
Scalability	Effective scalable range (ESR), effective system scalability (ESS), scalability, scale-up
Time	Mean time to contract the capacity of service, mean time to expand the capacity of service, resource allocation, resource deallocation, start-up, suspension, time/resources on time, total acquisition, total release.

Several works have proposed metrics to capture elasticity. Coutinho et. al., (2015, p. 11) conclude that it is not easy to define metrics for elasticity. They compile a list of metrics described in the literature. Covering all the metrics that exist would exceed the scope of this dissertation, however following the general concept of metrics to measure elasticity will be highlighted and some examples will be given.

Coutinho et al., (2015, p. 11) establish a distinction between general metrics used in the works about elasticity and metrics that attempt to describe elasticity specifically.

Some general metrics cited are:

- Response time (e.g. latency)
- Throughput (e.g. Megabytes/second)
- Reliability (number of violations)
- Availability (downtime, uptime)
- Scalability metrics (overhead, SLA, total capacity, energy use, cost)
- QOS violations

Coutinho et al., (2015, p. 11)

Some of the metrics specifically related to elasticity are captured in Table 2.5. Due to space constraints, not all metrics discovered by Coutinho et. al. are listed. For a full list, the reader is referred to the original publication.

The selection of metrics used by various authors (Table 2.5) shows that elastically scaling up and down is done with a variety of objectives in mind:

- Knowing the exact capacity and capacity range of a given system.
- Ensuring minimal cost for a given performance level.
- Ensuring a certain level of availability of a service (Violations, Service Level Objectives).
- Avoiding underutilised resources.
- Measuring the time and extent of a scaling action.

The goal underlying the elasticity definition used in this work belongs into the last category: measuring time and extent of a scaling action. The benchmark measures the time and extent a system spent in an over- or under-provisioned state.

2.8 Measuring elasticity and elasticity benchmarks

Aside from the theoretical aspect of defining metrics for elasticity, there have been multiple efforts to measure elasticity in practice. Some of the application benchmarks mentioned in section “2.6 Benchmarking” measure elasticity. For this dissertation, application benchmarks were not practical, as they don’t measure elasticity in isolation.

When reviewing the literature for methods to measure elasticity in isolation, several were found. Describing them all in detail would go beyond the scope of this dissertation. Below, the reasons why the respective measurement method was not chosen for this dissertation are given:

- Folkerts et al. (2012) propose a method but did not implement it yet. The proposed method has not ever been implemented yet.
- Suleiman (2012) proposes a method that is still in prototype stage.
- Shawky & Ali (2012) propose a method that is designed for cloud simulators rather than real clouds.
- Islam et al.’s (2012) method of measuring elasticity doesn’t account for differences in the efficiency of underlying resources and looks at elasticity from a financial point of view.
- Beltrán’s (2016) benchmark is not publicly available (Beltrán, 2016).

A number of authors investigated the measurement of elasticity in database systems (Cooper et al., 2010), (Dory, Mejías, Roy, & Tran, 2011), (Almeida, Sousa, Lifschitz, & Machado, 2013), but their approaches are not applicable to container-based cloud systems.

One comprehensive microbenchmark was found that assesses the elasticity of a system, taking into account efficiency differences of the underlying resources and observing elasticity independently of cost, is the BUNGEE benchmark (Herbst et al., 2015). This microbenchmark was selected for attempting to measure elasticity in cloud systems using container orchestration frameworks. The benchmark was chosen because it is free, publicly available, easy to use and is not in the prototype stage.

2.9 Measuring elasticity in container-based environments

Of all the benchmarks and measurement methods reviewed in the previous chapters, none is explicitly suited for container-based environments. The SPEC Cloud benchmark (SPEC, 2016) is designed for IaaS systems. The BUNGEE benchmark (Herbst et al., 2015) was also developed for and tested on IaaS systems.

This is not surprising since containers have only recently become popular. The first work on container elasticity was published in 2017 (Al-Dhuraibi et al., 2017, p. 1).

Al-Dhuraibi et al.'s (2017) review gives a good introduction into benchmarking in general and names various works that have dealt with implementing elasticity mechanisms in container-based systems. But no benchmark is mentioned that measures elasticity in such systems.

After a thorough literature research, no benchmark was found that explicitly measures elasticity in container-based environments.

2.10 The BUNGEE benchmark

The BUNGEE benchmark harness was developed at the Karlsruhe Institute of Technology in Cooperation with the University of Würzburg. The framework is described in Weber (2014) and Herbst et al. (2015). The BUNGEE benchmark is a microbenchmark designed to measure several elasticity metrics.

During the benchmark, a system under test (SUT) is exposed to load in form of HTTP requests. The requests are generated by an application called Apache JMeter²⁰.

The requests trigger a workload on the SUT. The workload can be chosen freely. A sample workload is provided: the computation of a Fibonacci number. The scaling

²⁰ Apache JMeterTM (Version 2.11). Retrieved from <https://jmeter.apache.org/index.html>

behaviour of the SUT is observed and several metrics to describe the system's scaling behaviour are computed.

2.10.1 Phases of the BUNGEE benchmark

The BUNGEE benchmark consists of the four phases listed below. The phases are visualised in Figure 2.7.

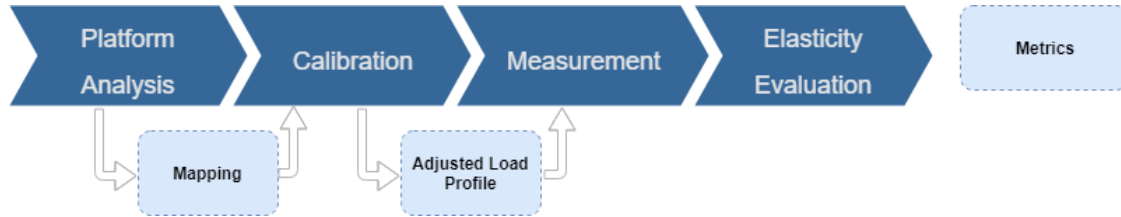


Figure 2.7: Phases of the BUNGEE framework

Phases of the BUNGEE benchmark:

1. Platform analysis phase (also called system analysis) (Weber, 2014, p. 51)

In this phase, elasticity and scaling behaviour of the system under test (SUT) are evaluated. The JMeter application on the load driver machine sends HTTP requests to one resource (i.e. VM). By evaluating the response times, the benchmark calculates whether under this load level, the SUT complies with previously defined service level objectives (SLOs). If the SLOs are met, the load is increased, else the load is decreased. The increase and decrease in load follows a binary search algorithm until the maximum load is found that one resource (i.e. VM) can handle without violating the SLO. The maximum load that one resource can handle is stored in requests per second. Then the number of resources is incremented by one and the search for the maximum load repeated. This process continues, incrementing the number of resources each time, up until the maximum number of resources defined in a configuration file. The objective of this phase is determining the maximum load each number of resources can handle.

2. Benchmark calibration phase (Weber, 2014, p. 51)

In this phase the data gathered in the platform analysis phase is evaluated. A mapping is constructed that associates each number of resources to the maximum load this number of resources could handle without violating the SLOs.

The benchmark can be supplied with a load profile of choice. This load profile is adjusted based on the mapping generated. The reason for this adjustment is to enable the benchmark to make two systems scale up to the same number of resources at the same time while running the load profile, independent of how much load one resource can handle.

3. Measurement and metric calculation phase (Weber, 2014, p. 51)

In the measurement phase, the actual benchmark is performed. The SUT is exposed to a series of requests defined by the adjusted load profile. The system then captures the response times and some other data, which enable it to calculate the elasticity metrics.

4. Elasticity Evaluation (Weber, 2014, p. 51)

In this phase, the elasticity metrics are computed and written to a file.

2.10.2 Metrics captured by the BUNGEE benchmark

The BUNGEE benchmark computes the following metrics as described in (Herbst et al., 2015, p. 48):

1. Under-provisioning accuracy ($accuracy_U$)

The sum of areas in the graph, when resources were under-provisioned (ΣU) divided by the duration of the measurement.

$$accuracy_U[resource\ units] = \frac{\Sigma U}{T}$$

(Herbst et al., 2015, p. 48)

2. Over-provisioning accuracy ($accuracy_O$)

The sum of areas in the graph, when resources were over-provisioned (ΣO) divided by the duration of the measurement (T)

$$accuracy_O[resource\ units] = \frac{\Sigma O}{T}$$

(Herbst et al., 2015, p. 48)

3. Under-provisioning timeshare ($timeshare_U$)

The time spent in an under-provisioned state (ΣA) divided by the total duration of measurement (T).

$$timeshare_U = \frac{\Sigma A}{T}$$

(Herbst et al., 2015, p. 48)

4. Over-provisioning timeshare ($timeshare_O$)

The time spent in an over-provisioned state (ΣB) divided by the total duration of measurement (T).

$$timeshare_O = \frac{\Sigma B}{T}$$

(Herbst et al., 2015, p. 48)

5. Jitter

The BUNGEE framework also computes a metric called Jitter which captures the stability vs unnecessary fluctuations of resource adaptation. Jitter will not be discussed as part of this research, details can be found in Herbst et al., (2015, pp. 48–50).

The computed metrics are best visualised using a diagram. Figure 2.8 shows a graph of a benchmark run. Time is measured on the x-Axis. The number of resources provisioned or required by the elastic system is charted on the y-Axis.

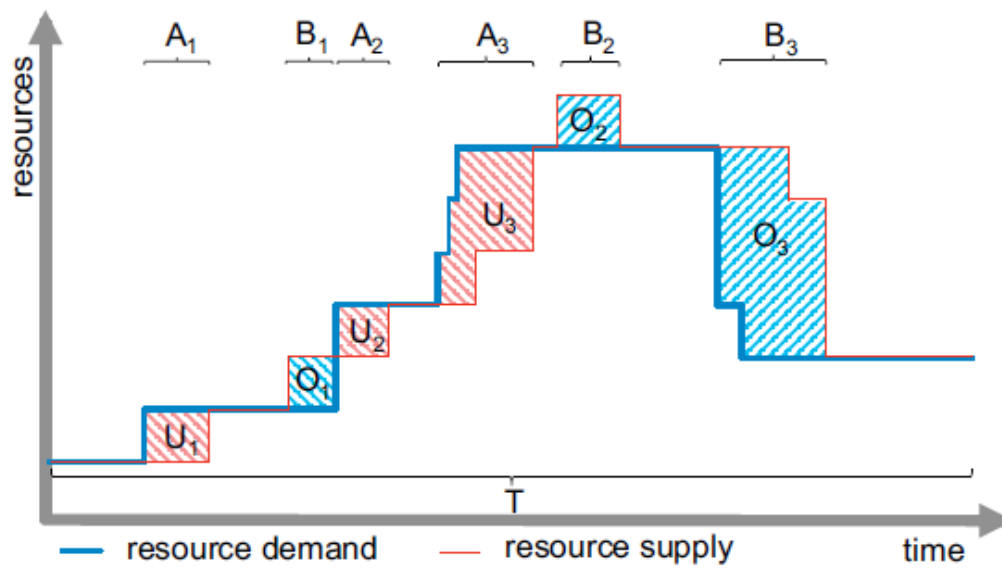


Figure 2.8: Accuracy and Timeshare metrics. Source: Herbst et al., 2015, p.48

The areas in red are the times when the system is under-provisioned. The areas in blue are the times when the system is over-provisioned. From these areas and the time metric, the metrics listed above can be computed.

2.10.3 Technical details of the BUNGEE benchmark

To run the BUNGEE benchmark, a load driver and a SUT are required. The load driver is a computer which runs the applications generating the load that is sent to the SUT. The SUT is a cloud system of which the elasticity should be measured.

Load Driver

Eclipse IDE, Limbo Load Intensity Modeling Framework, BUNGEE framework, the AWS SDK and JMeter version 2.11 must be installed on the load driver (Rauh & Herbst, 2015, p. 4)

The BUNGEE source code is opened in Eclipse and can be executed from there. Some settings can be adapted in configuration files in the folder “Property Files” (Rauh & Herbst, 2015, p. 7). Variables like the hostname and port, the path to the JMeter application, the timeout and the number of benchmarked resources can be configured in those files.

The BUNGEE source code contains some example files with the code to execute the BUNGEE benchmark on CloudStack private clouds or on AWS EC2. The source code can directly be executed from the Eclipse IDE. JMeter will be automatically executed from the code. The University of Würzburg provides detailed instructions on how to use the benchmark in a pdf document (Rauh & Herbst, 2015) downloadable on their website²¹.

System Under Test (SUT)

The SUT is a cloud system chosen by the benchmark user. It is running an application that receives and processes the traffic from the load driver. The benchmark by default contains an application that calculates a Fibonacci number which consistently tasks the CPU to approximately the same intensity and returns the result to the load driver. The load driver can then calculate the response time accurately, establish if the application met the SLOs and calculate elasticity metrics (Weber, 2014). The calculation of the Fibonacci number is a CPU intensive workload. The benchmark was designed for IaaS systems, so the SUT is usually an IaaS system.

2.10.4 Data collected by the BUNGEE benchmark

The BUNGEE benchmark collects a variety of data, both in the system analysis phase of the benchmark as well as in the measurement phase of the benchmark. The following section will present which relevant output files are collected. Files that are considered irrelevant for the evaluation of the results are omitted.

System Analysis

During the system analysis phase, the benchmark produces one folder per number of instances analysed. Each folder contains various sub-folders denoting the load intensity that was applied. In these sub-folders, the timestamps when requests were scheduled (timestamps.csv) and the responses (responses.csv) are stored.

Figure 2.9 depicts the folder structure that results as an output from the system analysis phase.

²¹ Chair of Software Engineering, University of Würzburg. Retrieved December 25, 2017, from <http://descartes.tools/bungee>

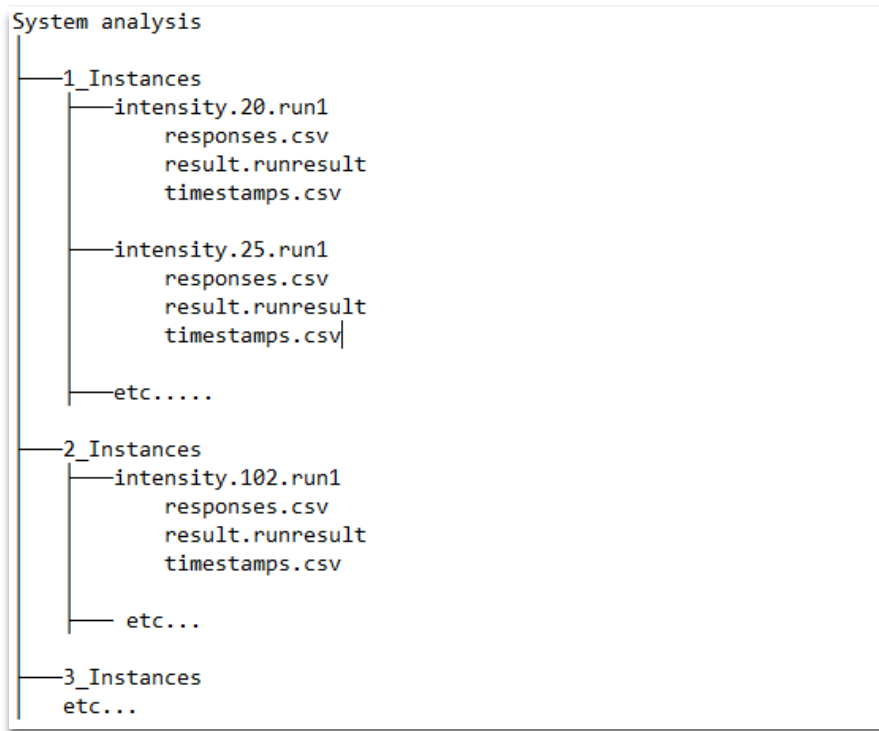


Figure 2.9: Output files and folders generated by BUNGEE system analysis phase

A full list of the contents of each of these files can be found in Appendix I. The file “responses.csv” contains the information the benchmark employs to calculate the elasticity metrics.

Measurement phase

During the measurement phase, the benchmark collects those data listed in Table 2.6.

Table 2.6: Data captured in BUNGEE measurement phase

File	Content
timestamps.csv	See Appendix I, timestamps.
responses.txt	See Appendix I, responses.
metrics.csv	Contains the output metrics from the benchmark: Accuracy_O, accuracy_U, timeshare_O, timeshare_U, jitter
violations.txt	Number of SLO violations, number of total requests, ratio of violations

Allocations subfolder

demand.csv	Contains two columns: timestamp and amount. The amount column contains the resource demand.
monitored.csv	Contains two columns: timestamp and amount. The amount column contains the current resource supply.

Many different evaluations would be possible from those data provided. This thesis evaluates the “mapping.mapping” file, as this file already aggregates raw data in a usable format, extracting from all the raw responses the relevant metrics in order to evaluate the maximum load a resource can process without violating the service level objectives.

3 DESIGN AND METHODOLOGY

This chapter introduces the AWS Elastic Container Service (ECS), furthermore the experimental setup is described in detail: The experiment is summarised, followed by a description of the source code written to extend the BUNGEE framework to ECS. Two small adjustments made to the original BUNGEE code are also described.

To ensure all conditions of the experiment are clearly outlined, the network and load driver machine specifications are captured and the parameters of the ECS configurations listed. Potential alternatives for configuring the experiment are explored. The Docker file and EC2 instance configurations are described.

3.1 AWS Elastic Container Service (ECS)

ECS is a container orchestration platform that works on top of Amazons IaaS platform EC2. ECS has two launch types: EC2 and Fargate. Both launch types allow the cloud user to provision tasks²² within Docker containers. With the EC2 launch type, the cloud user must provision EC2 infrastructure: virtual machines and load balancers. The virtual machines accommodate the containers²³.

With the Fargate launch type, the underlying VM infrastructure is abstracted away from the cloud user. Fargate was launched in November 2017 but is currently (April 13th, 2018) only available in one AWS region: US East²⁴. Due to the Fargate launch type being new and its availability restricted, the EC2 launch type was chosen for this research. The following descriptions apply to the EC2 launch type.

To run a task in ECS, several components must be created and configured. Those configurations can be made either through the AWS web interface, the AWS command line client or through the AWS SDK.

²² A task is a container running with settings specified in a “task definition” The task definition is specified in the AWS interface, via JSON or the AWS SDK. Simplified, a task approximately equals a container.

²³ Amazon Web Services, Inc. Amazon ECS Features - run containers in production. Retrieved April 2, 2018, from <https://aws.amazon.com/ecs/features/>

²⁴ Amazon Web Services, Inc. (2017, November 29). Introducing AWS Fargate. Retrieved April 13, 2018, from <https://aws.amazon.com/about-aws/whats-new/2017/11/introducing-aws-fargate-a-technology-to-run-containers-without-managing-infrastructure/>

For this research, most configurations were made through the AWS Java SDK, with some select configurations made through the AWS web interface.

ECS Components

In ECS, one or more tasks make up a service. One or more virtual machines form a cluster, which runs one or several services. Each virtual machine must run an ECS container agent for container orchestration and have a specific configuration file to join the correct cluster²⁵.

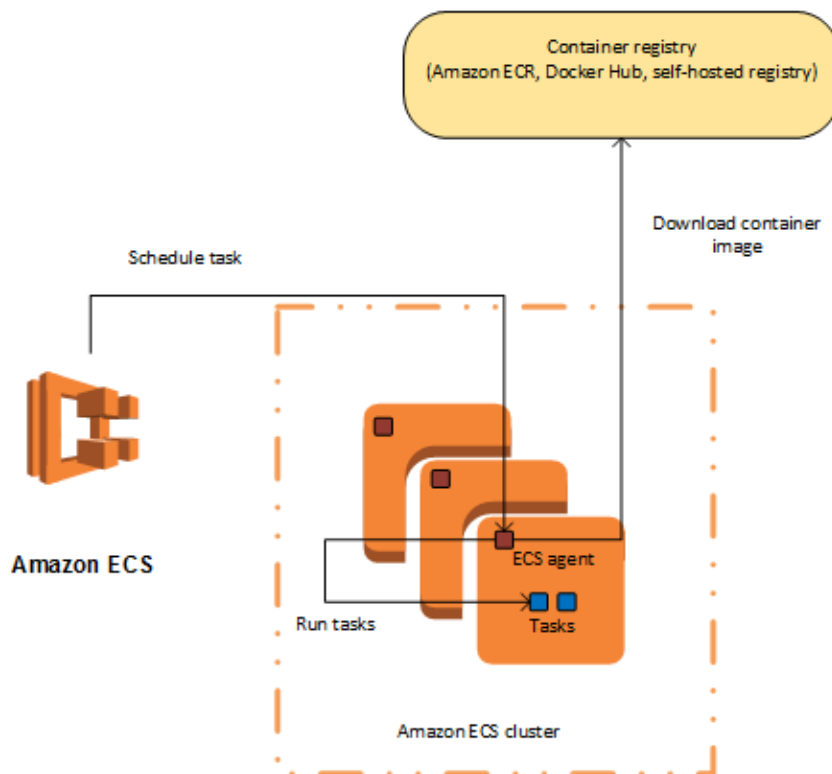


Figure 3.1: ECS instances, tasks, container agent. Source: Amazon Web Services Inc.

Figure 3.1 shows three EC2 instances (VMs) which are each running an ECS agent. Tasks are running inside those instances.

The workflow to set up and run tasks with AWS ECS is as follows:

EC2 settings:

²⁵ Amazon Web Services, Inc. What is Amazon Elastic Container Service? - Amazon Elastic Container Service. Retrieved April 17, 2018, from <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/Welcome.html>

1. One or more virtual machines must be created to accommodate the tasks. The machines must join the desired cluster.
2. An application load balancer and target group must be created to direct the traffic to the container instances.
3. If the system should auto-scale, an auto-scaling group must be created and configured.

ECS settings:

4. The application to be executed (e.g. load processor) must be added to a Docker container. This can be done on a local computer that has Docker installed.
5. The Docker container containing the application must be added to the Elastic Container Registry (ECR).
6. A task definition must be created, pointing to the Docker image in the ECR. In the task definition, CPU shares can be reserved for the container via the parameter “cpu”. The allocated number of CPU shares is the minimum CPU units the container can use. For a more detailed discussion of this parameter, see 4.2.3 “CPU utilisation of individual containers”.
7. A cluster must be created, or the default cluster must be used.
8. A service must be created, auto-scaling settings can be assigned for the service. The desired number of tasks must be specified. If auto-scaling is configured, the minimum and maximum number of tasks can be specified. Scaling policies can be flexibly assigned, specifying CPU or memory usage thresholds and actions that should be taken accordingly.

3.2 Experimental setup

This section will describe how the experiment was set up, detailing the settings that were configured in ECS, EC2 and on the load driver machine.

3.2.1 Experiment Summary

To test the hypotheses stated in section 1.3, the BUNGEE framework was extended to AWS Elastic Container Service. Where the original BUNGEE AWS implementation would consider a resource to be an instance of a virtual machine (EC2 instance), the

BUNGEE AWS ECS implementation considers one task (i.e. one Docker container) to be one resource.

This allows the BUNGEE benchmark to be directly applicable to AWS ECS without any extensive modifications, as was stated by the benchmark author (Weber, 2014, p. 53).

Once the implementation was complete, the system analysis phase of the BUNGEE Benchmark was conducted 32 times. 10 runs were conducted with a physical machine as load driver, connected in library of the Dublin Institute of Technology. As only one run per day was possible during the opening hours of the library, gathering the data proved difficult. The University of Würzburg kindly provided a virtual machine in their private computer network, so that the remaining 22 runs could be conducted using this virtual machine as a load driver. Using a virtual machine in a public cloud as load driver was out of the question, as the performance variation in public clouds could negatively impact the experiment.

During each run of the system analysis, the maximum load intensity that each number of tasks (i.e. containers) could handle was established. This was tested first for one resource and then incrementally up to 6 resources. The mappings of number of resources to load handling capability were collected.

Two statistical analyses, “repeated measures analysis of variance” (ANOVA) and t-test were conducted to determine if the outcome of each run of the system analysis significantly differs from the others.

3.2.2 Extension of the BUNGEE framework to facilitate AWS ECS

To extend the BUNGEE benchmark to ECS, the AWS SDK v. 1.11.286 was used, this code can be accessed on GitHub²⁶.

Following the structure of the original BUNGEE framework, a package “examples” was created which contains three executable programs (see also Figure 3.2):

- `AwsEcsDetailedSystemAnalysis` – runs the system analysis phase using ECS.
- `RunBenchmarkOnAwsEcs` – runs the benchmark phase using ECS.

²⁶ https://github.com/Norali81/bungee_ecs

- **SetUpEnviornment** – Facilitates setting up the components needed to run AWS ECS. This program creates the necessary security groups, an application load balancer, an ECS cluster, an ECS service and EC2 instances (virtual machines) which already have the ECS container agent installed and the configuration files in place for them to automatically join the correct cluster.

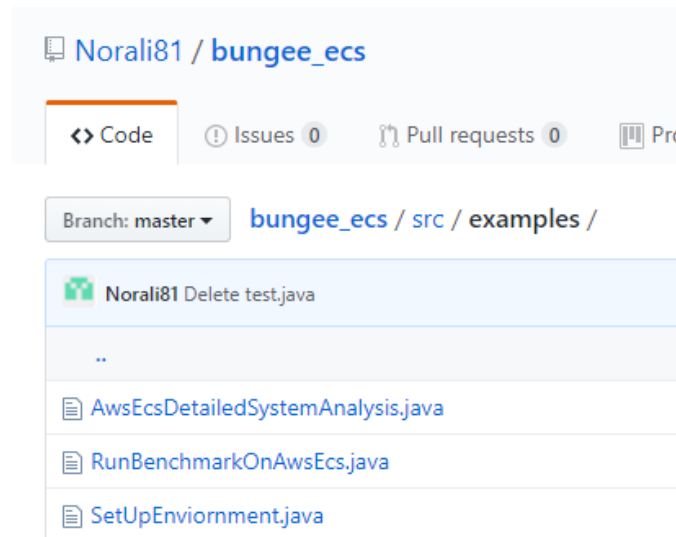


Figure 3.2: BUNGEE ECS code examples

The functionality to facilitate the above described programs can be found in the package “tools.descartes.bungee.cloud.aws.ecs” on GitHub.

The class “AwsEcsManagement” implements the interfaces “CloudInfo” and “CloudManagement”, which enable the BUNGEE benchmark to interact with ECS.

It contains the methods “setScalingBounds()” and “getNumberOfResources()”. These two methods are all the code that needed to be written to extend the BUNGEE framework to a new cloud provider. The method “setScalingBound()” is needed for the system analysis phase to adjust the number of resources available. The method “getNumberOfResources()” is needed to monitor the number of resources available.

The code written is commented, which fully explains how it is implemented.

3.2.3 Modifications to the existing BUNGEE code

Two very small amendments had to be made to the BUNGEE code.

1. The constant `SLEEP_FOR_STABILIZATION_MILLI` was changed in the class “ResourceWatch.java”. The time to wait for stabilisation was changed from 3 minutes to 10 minutes. While it remained at 3 minutes, the system analysis would not run to completion, as the next resource level would handle less load than the previous one. The cause of this has not yet been determined. One possible hypothesis is that the task is not running yet for some time after starting, despite being indicated in the ECS interface as running.
2. The function “enquote()” used to place file paths in quotes was removed from all occurrences in “JMeterController.java”, so that file paths weren’t placed in quotes. This was necessary to make BUNGEE work with the Ubuntu file system, as the load driver machine was running Ubuntu.

Table 3.1: Network specifications DIT library

Property	Value
Connection Speed	13MBps
Network Cable to load driver	Category 5e (suitable for Gigabit Ethernet)
Ethernet Switch	Extreme 7100-Series (100Gbit/s)
Network cable from port to switch	100Gbit/s

3.2.4 Network specifications

The first part of the experiment was conducted from the DIT library. The specifications of the network are listed in Table 3.1. The second part of the experiment was conducted from a virtual machine in University of Würzburg’s network, connected at a speed of 1Gbps.

3.2.5 Load driver machine specifications

The first 10 runs of the system analysis were conducted with a physical machine as load driver. The remaining 22 runs were conducted with a virtual machine.

On both machines, “Eclipse for RCP and RAP Developers” was installed²⁷. The specifications of the load driver machines are detailed in Table 3.2 and Table 3.3.

For network time synchronisation, “chrony”²⁸ was installed. Chrony is an implementation of the Network Time Protocol (NTP)²⁹.

Table 3.2: Specifications of the physical load driver machine

Specification	Value
Model	Lenovo Legion Y520
Ethernet adapter	Connection speed up to 1000Mbps
Operating System	Ubuntu 16.04.1 64bit
Memory	8GB
Processor	Intel® Core™ i5-7300HQ CPU 2.5GHZ

Table 3.3: Specifications of the virtual load driver machine

Specification	Value
Operating System	Ubuntu 16.04.4 LTS (Xenial Xerus) 64 bit
Memory	4GB
Processor	Intel® Xenon® CPU ES-2640 v3 @ 2.60GHz
Hypervisor	Xen
Host	8 CPU cores, each core 2600Mhz CPU Speed, 32GB RAM

²⁷ Eclipse for RCP and RAP Developers. Retrieved March 26, 2018, from <http://www.eclipse.org/downloads/packages/eclipse-rcp-and-rap-developers/oxygen2>

²⁸ Churnow, R., & Lichvar, M. (2017). Chrony (Version 2). Retrieved from <https://chrony.tuxfamily.org/>

²⁹ The Network Time Protocol (NTP) is a protocol which can be used to synchronise the clocks of distributed systems. The accuracy has been described as in the order of tens of milliseconds over the internet (Coulouris, Dollimore, Kindberg, & Blair, 2011, p. 622).

3.2.6 AWS cloud environment setup

To set up the ECS environment, the script “SetUpEnvironment.java” was used. The script was written for this dissertation. The following AWS components were created: EC2 security groups, EC2 application load balancer, ECS target group, ECS cluster, ECS service, container image and EC2 instances. The exact parameters that were configured in the AWS environment for this experiment can be found in Appendix II.

3.2.7 Alternative experimental setup

During the system analysis phase, the number of virtual machines was kept stable at 5 to ensure containers always have a virtual machine they can be spawned on.

There would have been two alternative ways to provision virtual machines for the containers:

1. Provision only 3 virtual machines, as 6 containers should fit on 3 VMs given the selected settings.
2. Provision one virtual machine and set EC2 to autoscaling.

For both setup configurations, the task placement strategy should be set up at service creation. The options binpack, random and spread are available and each of these settings will cause a different allocation of containers to virtual machines³⁰.

For this research, none of the above task placement strategies was explicitly specified when the task definition was created. This means the default option was chosen, however it was retrospectively found that the system analysis might yield more stable results with the setting “binpack” (see 4.2.2. Container placement on virtual machines).

3.2.8 Further configuration

Figure 3.3 shows the user interface of the ECS service created for this experiment. The screenshot captures a starting task. The “desired count” is set to 1, a task has recently been started and has the status “pending”.

³⁰ See: Amazon ECS Task Placement Strategies - Amazon Elastic Container Service. Retrieved April 13, 2018, from <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-placement-strategies.html>

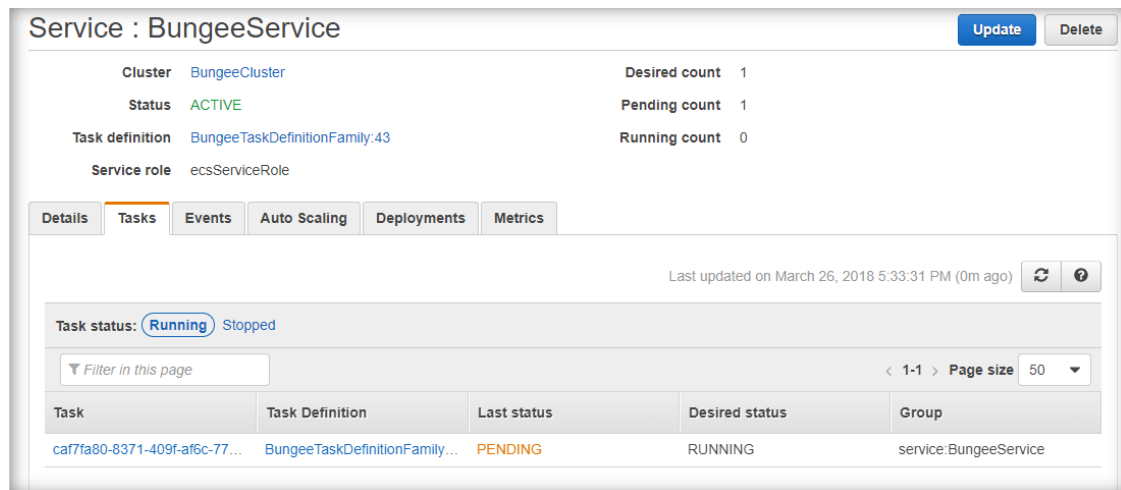


Figure 3.3: Elastic Container Service: screenshot of AWS user interface

Figure 3.4 shows the Docker file used to create the container running the application which processes the requests sent by the load driver. The Docker file retrieves the Docker image with Ubuntu 16.04, installs the latest updates and the java runtime environment. The BUNGEE simpleHTTP application, which receives and processes JMeter's HTTP requests, is copied to the Docker image and executed.

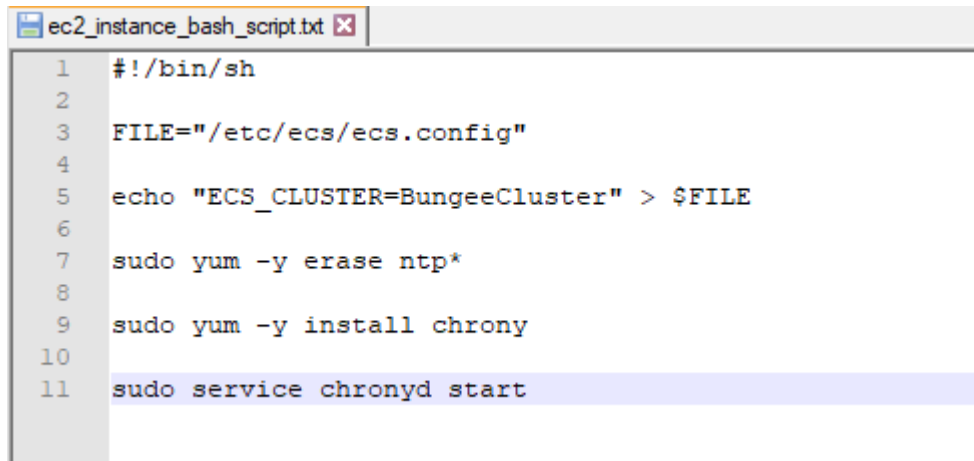
```

1 FROM ubuntu:16.04
2
3 RUN apt-get update -y
4 RUN apt-get install default-jre -y
5
6 # copy jar file into docker
7 COPY SimpleHttp.jar ~/
8
9 # execute the jar file
10 CMD ["java" , "-jar" , "~/SimpleHttp.jar"]
11

```

Figure 3.4: Dockerfile to used create docker image

Figure 3.5 shows the bash script that was executed each time when starting an EC2 instance. This script installs and starts the chrony for NTP synchronisation⁴⁰ and configures the instance to join the correct ECS cluster.

A screenshot of a terminal window titled 'ec2_instance_bash_script.txt'. The window contains a bash script with 11 lines of code. Line 1 is the shebang '#!/bin/sh'. Line 2 is empty. Line 3 sets 'FILE="/etc/ecs/ecs.config"'. Line 4 is empty. Line 5 is 'echo "ECS_CLUSTER=BungeeCluster" > \$FILE'. Line 6 is empty. Line 7 is 'sudo yum -y erase ntp*'. Line 8 is empty. Line 9 is 'sudo yum -y install chrony'. Line 10 is empty. Line 11 is 'sudo service chronyd start', which is highlighted with a light blue background.

```
1 #!/bin/sh
2
3 FILE="/etc/ecs/ecs.config"
4
5 echo "ECS_CLUSTER=BungeeCluster" > $FILE
6
7 sudo yum -y erase ntp*
8
9 sudo yum -y install chrony
10
11 sudo service chronyd start
```

Figure 3.5: Bash script to configure instance for ECS and set up chrony NTP implementation

3.2.9 DNS issues encountered during initial tests

The experiment could not be run successfully on a 300Mbps Virgin Media home broadband. During the system analysis phase, with increasing load generated by the load driver, the number of errors in the response files increased. The error code captured was “Non HTTP response code: java.net.UnknownHostException”.

Detailed investigation was carried out. Logging of all DNS requests was activated. It was found that the DNS server did not respond to all DNS requests. Therefore, the hostname of the AWS load balancer could not be resolved.

To mitigate this, two alternatives were explored:

- It was tried to contact the AWS load balancer by IP instead of by hostname, making DNS resolution unnecessary. This stopped the error “UnknownHostExcpetion” from occurring. Unfortunately, it was found that the IP address of the load balancer changed during the experiment, aborting the experiment. Identifying the AWS load balancer by IP address instead of hostname therefore wasn’t feasible.
- It was attempted to use the Google DNS at address 8.8.8.8 but this server generated the same error code. It could not be clarified why Virgin Media and Google DNS produce this error. One possible hypothesis might be that these services filter requests if the same user makes an unusually high number of requests.

Due to the above-mentioned DNS resolution issues, the experiment was initially carried out from the DIT library. In the DIT network each computer has one primary and two secondary DNS servers assigned. The “UnknownHostException” did not occur running the experiments from the DIT network. Later, the University of Würzburg kindly provided a virtual machine in their private cloud to be used as a load driver and therefore made further experiments possible.

3.3 Statistical methods for evaluation

For the evaluation of this experiment, two different statistical analyses were applied. A one sample two tailed t-test was used to generate confidence intervals for the means of load processed at each resource level. Further, a “repeated measures analysis of variance” (ANOVA) was conducted, with each benchmark run considered as one repeated measure and each intensity-load pair as one sample point.

3.3.1 One sample, two tailed t-test

The two-tailed t-test is suitable to test whether a sample with the mean \bar{x} is significantly different from a population in which the mean μ equals to a specific value (Sheskin, 2007, p. 157).

In previous research, to verify the stability of the original BUNGEE benchmark, the authors repeated the system analysis 10 times in their private cloud and conducted a two tailed t-Test. The test showed that the results don’t fluctuate by more than 5% in either direction (Weber, 2014, p. 77).

The t-test can also be used to establish the confidence interval in which the true population mean lies with a given probability (Sheskin, 2007, p. 174). This makes it possible to state, after running a t-test, if the population mean μ , with a probability of $\geq 1-p$, deviates more than 5% from the sample mean \bar{x} or not.

Considering these characteristics, the t-test is suitable for the purposes of this dissertation. The use in previous literature makes it compelling to use the t-test for comparability reasons. In chapter 4.2 “Results”, the one-sample, two tailed t-test is used for each sample of load intensity measurements per resource level. The confidence intervals are calculated. The accepted variation of the true mean from the sample mean

is 5% as in (Weber, 2014). It is tested if the true mean μ , with 95% probability, does not deviate more than the accepted variation from the sample mean \bar{x} .

It is to be expected that the confidence interval for the true mean μ will be much wider in the results obtained in this dissertation compared to the research of the original researchers creating BUNGEE. The original research was carried out in a private cloud. In public clouds, contention between virtual machines running on the same physical hardware can occur (Govindan, Liu, Kansal, & Sivasubramaniam, 2011, p. 2) and performance variability has been reported (see 2.5 “Performance variability in public clouds”).

The t-test is based on the following assumptions (Sheskin, 2000, p. 67):

- The sample has been randomly selected
- The distribution of the underlying population is normal

For this experiment, the sample selection was not entirely random, as the start time of the experiment could not be randomised. For practical reasons the experiment had to be started whenever the researcher was available. However, given the multitude of factors that can be affecting the performance of a cloud system behind the scenes, the sample selection was deemed random enough for this statistical analysis.

3.3.2 Repeated Measures ANOVA

The “repeated measures analysis of variance” (ANOVA) is also called “Single-Factor Within-Subjects Analysis of Variance” (Sheskin, 2007, p. 413). It can be applied if the same group (in this case the same system) is measured on one factor (in this case load) more than once (Salkind, 2017, p. 334). The aim is to detect a statistically significant difference between the multiple measurements.

The advantage of using this analysis over the one-sample t-test is, that the t-test, applied to this study, must look at the load-resource mappings for each resource level individually, while the repeated measures ANOVA can treat each system analysis run as one repeated measurement and can compare all samples at once.

The null hypothesis of the ANOVA typically states that the means of all repeated measurements are equal.

“Null hypothesis: $H_0: \mu_1 = \mu_2 = \mu_3$

Alternative hypothesis: $H_1: \text{Not } H_0$ ” (Sheskin, 2000, p. 627)

If the null hypothesis is rejected, this means that at least one of the means deviates from the other ones.

The ANOVA is based on the following assumptions (Sheskin, 2000, p. 626):

- The sample has been randomly selected.
- The values are normally distributed in the underlying population.
- Sphericity

Sphericity is a mathematically complex computation that assesses if variances and covariances of the underlying populations are equal (Sheskin, 2000, p. 626). A full explanation of the concept of sphericity can be found in (Sheskin, 2000, pp. 337–341).

4 IMPLEMENTATION AND RESULTS

This chapter outlines the results of the experiment and its statistical analysis. The experiment consisted of running the system analysis phase of the benchmark 32 times. First, some observations and graphs from the AWS reporting interface are presented, which capture the typical behaviour of AWS ECS during an iteration of the system analysis.

In the next step, descriptive statistics are calculated and the hypotheses stated in 1.3 “Research Objectives” are tested with the statistical methods described in 3.3 “Statistical methods for evaluation”. Further, the results are evaluated by comparing them to results obtained by other researchers (Iosup et al., 2011), (Leitner & Cito, 2016).

Lastly, the results of one run of the BUNGEE measurement phase are captured very briefly, to confirm that the whole benchmark indeed runs successfully with the ECS extension created for BUNGEE. This is necessary because the experimental data captured in this dissertation only relates to the system analysis phase of the benchmark. It must be ensured however, that the code written to extend BUNGEE to ECS works for the entire benchmark, not only in the system analysis phase.

4.1 Implementation

During the implementation phase of this dissertation, the system analysis was conducted 32 times:

- 10 times with a physical load driver based in the DIT library starting between 10 and 11 am each day.
- 22 times with a virtual machine as a load driver, running from the private cloud of the University of Würzburg at varying times.

The system analysis phase was tested from 1 to 6 resources. The load each resource level could handle was established. One system analysis run took approx. 4-6 hours, therefore only one analysis per working day could be conducted from the DIT library. For this reason, later a virtual machine was used as a load driver, which was kindly provided by the University of Würzburg.

The system analysis was configured to only use up to 6 resources, under the assumption that if the system analysis runs stably up to 6 resources, it would likely behave in a

similar fashion for more resources. Testing more than 6 resources would increase the duration and cost of the experiment and likely not produce any more useful data.

4.1.1 Observations during the system analysis phase

During the system analysis phase, the system was exposed to bursts of load following a binary search pattern, until the load was found which a given number of resources (i.e. containers) could handle without violating the SLO (see 2.10.1 “Phases of the BUNGEE benchmark”).

This binary search pattern is reflected in the CPU utilisation curve taken from the AWS monitoring system (Figure 4.1). It can be observed that the CPU utilisation has six phases of high CPU utilisation, coinciding with the six resource levels tested. Each phase of high CPU utilisation is followed by a period of low CPU utilisation, occurring when the resource level was re-adjusted and an idle-period initiated to give the system time to stabilise. Each phase of high CPU utilisation has a peak of > 90% CPU utilisation at the beginning, where the binary search algorithm tries an intensity that overloads the system. After this peak, the search algorithm slowly adjusts by increasing and decreasing the load, until it finds a level where SLOs can be maintained. The ideal CPU utilisation level seems to be somewhere between 60-70%, as observed by looking at the last level of CPU utilisation of a burst of load (see Figure 4.1).



Figure 4.1 CPU utilisation of the ECS service during system analysis

The AWS monitoring graphs for active connections, new connection count, HTTP 200 (OK) response codes, processed bytes and consumed load balancer units (Figure 4.2) all

follow the expected pattern of six bursts of traffic with increasing load (less load for one instance, more for two instances, etc).



Figure 4.2: AWS application load balancer metrics

Figure 4.3 shows the count of unhealthy and healthy hosts (tasks/containers), the average latency in seconds, the increasing number of requests and any Hypertext Transfer Protocol (HTTP) errors received.

These graphics again show the expected increase in number of containers and the increase in load to test a greater number of containers.



Figure 4.3: AWS target group metrics 1

Several HTTP 4XX (Client error) and 5XX (Server error) errors were observed (Figure 4.4, Figure 4.5). Those errors coincide with load spikes during the system analysis phase.

It could not be established which error code was returned to the client, as the response files report only a socket timeout. It is likely that it took the test application too long to respond and therefore JMeter closed the connection. The timeout period for JMeter is set to 10 seconds by the BUNGEE benchmark, although some latencies greater than 10 seconds have been observed in the response files.

Amazons documents indicate that an error 460 is received when a client closes a connection before the load balancer responds³¹. A 504 “gateway timeout” error is received if the load balancer did not establish a connection to the container (target) before the connection timeout of 10 seconds expired³¹.

Given the above, the errors seem to be related to the system not coping well with increased load. This behaviour is expected, as the benchmark is intentionally overloading the system.

Clarifying the cause of these errors with certainty would have been possible via the access logs of the application load balancer³². Unfortunately, these logs were not activated during the experiments, but for future studies, it is intended that the logs will be enabled to clarify the cause of the errors.

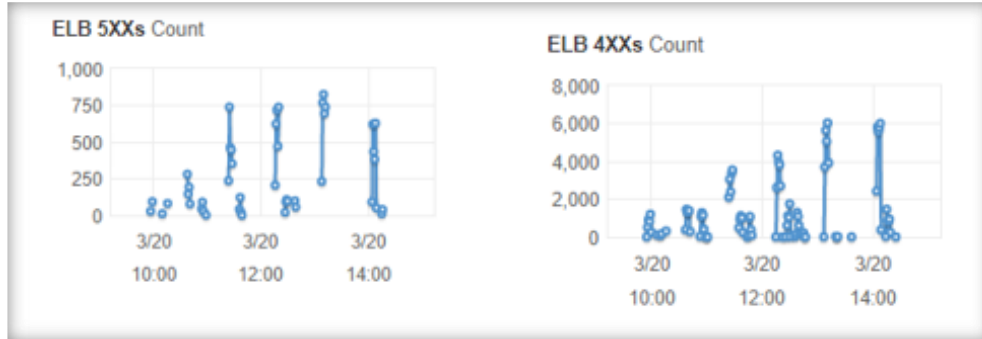


Figure 4.4: 4XX an 5XX errors during system analysis phase

³¹ Amazon Web Services, Inc. Troubleshoot Your Application Load Balancers. Retrieved March 28, 2018, from <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-troubleshooting.html#target-http-errors>

³² Amazon Web Services, Inc. Access Logs for Your Application Load Balancer - Retrieved April 28, 2018, from <https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-access-logs.html>

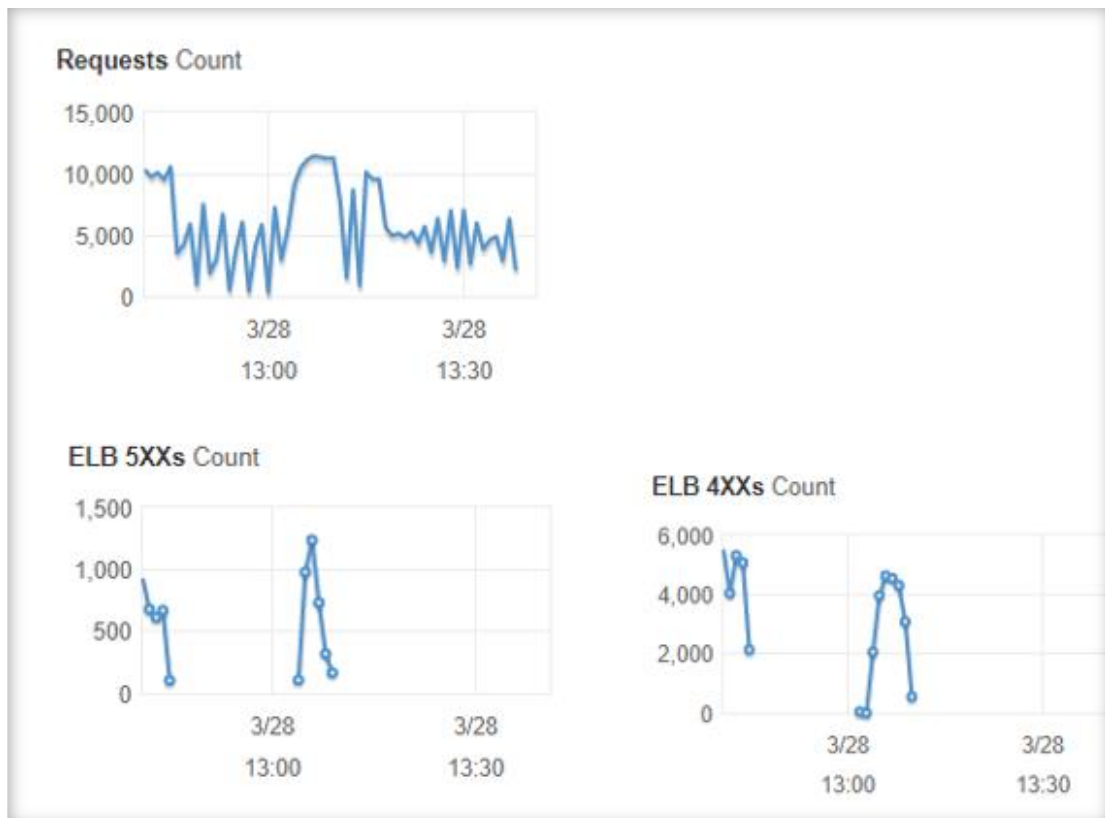


Figure 4.5: 5XX an 4XX error during the benchmark phase

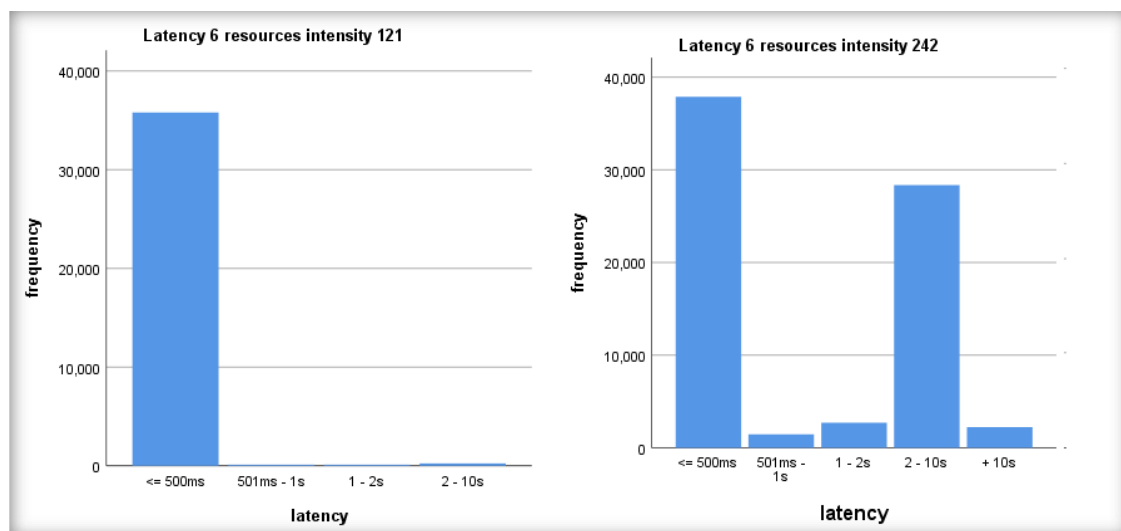


Figure 4.6: Latency by load intensity (SLO met, left. SLO failed, right)

Figure 4.6 shows the latency captured in the response file when the system was under acceptable load vs under too much load. The bar charts visualise the number of requests

per latency bucket. The left chart shows 6 containers exposed to 121 requests per second. The latency was generally $\leq 500\text{ms}$. The right chart shows 6 containers exposed to a load of 242 requests per second. The latency was $\leq 500\text{ms}$ in most cases, but ≥ 2 seconds in many cases.

4.2 Results

The system analysis was conducted 32 times in the AWS environment with ECS, one resource being one container. The results of the experiment are captured in Appendix III. The first column denotes the ID of the system analysis run, including the date and in some instances the time. The second column denotes whether the load driver was a physical or virtual machine. The following columns denote the maximum load achieved for the different resource levels.

In Appendix VI, a statistical comparison between the two sets of results is undertaken. 10 results were captured with a physical load driver, running from the DIT library. Further 22 results were captured with a VM as load driver, running from the private cloud of the University of Würzburg. A Man-Whitney-U test was conducted, and no significant difference was found between two groups. Therefore, the results will be evaluated as being one sample of 32 measurements.

4.2.1 Descriptive Statistics

The maximum load fluctuates at each resource level. The range varies between 15 requests per second (RPS) and 29 RPS, the standard deviation was between 3.64 and 7.75 RPS. In comparison, Weber, (2014, p. 77) achieved standard deviations ranging between 0 and 1.57 RPS, running the system analysis in a private cloud. Table 4.1 shows the range, minimum, maximum, mean, standard deviation and the relative standard deviation³³ of the maximum load achieved per resource level. The relative standard deviation in this case shows that the maximum load achieved with one resource had the biggest variation, followed by 2 resources. The smallest variation was achieved with 6 resources.

³³ The relative standard deviation is also called “coefficient of variation” (Sheskin, 2000, pp. 9–10).

Figure 4.7 shows the mean load intensity handled per resource level. The graph shows that the mean load intensity per resource grows in a linear fashion with the 6th resource performing slightly less. This might be the case because during the research, 6 containers were allocated on 5 virtual machines. It is possible that the 6th container was the first one to share a virtual machine with another container, having to share its resources. This seems to be supported by the graph in Figure 4.8, which shows that the 6th container on average handles significantly less additional load than the other containers.

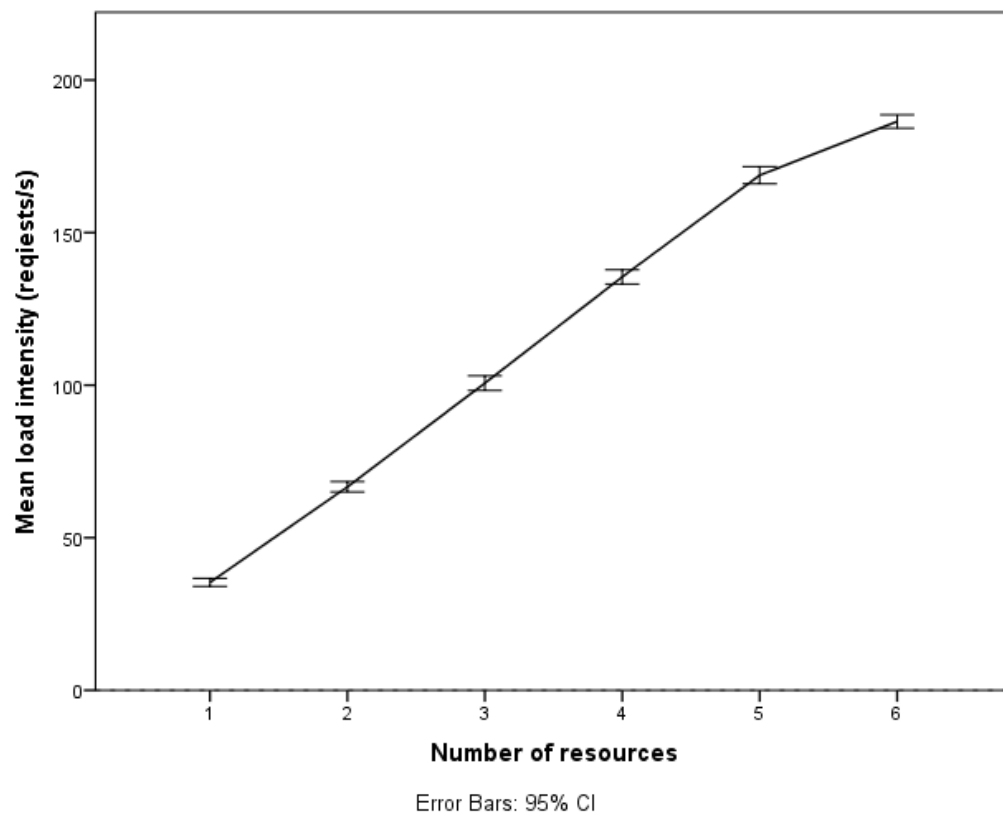


Figure 4.7: Mean load intensity (RPS) handled during each system analysis \pm standard error (31 df, $p < 0.05$)

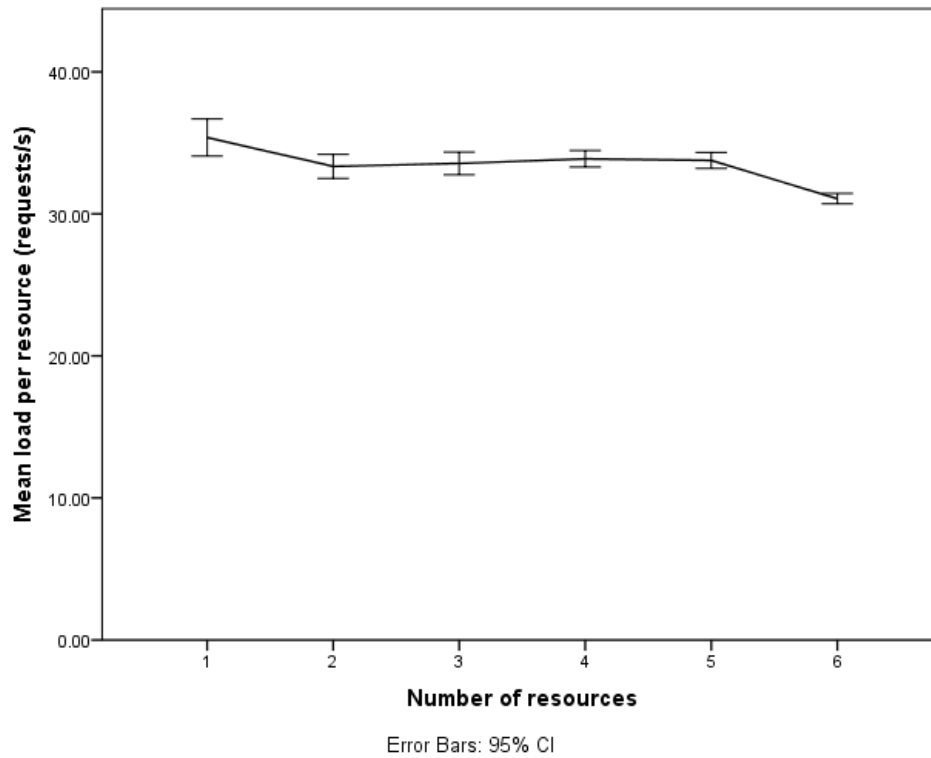


Figure 4.8: Mean load (requests/s*resources) handled per resource \pm standard error (31 df, $p < 0.05$)

Table 4.1: Descriptive statistics

	N	Range	Minimum	Maximum	Mean	Std. Deviation	Relative Std. Deviation (Variation Coefficient)
1 resource	32	15	27	42	35.38	3.64	10.30
2 resources	32	23	53	76	66.69	4.68	7.02
3 resources	32	28	84	112	100.66	6.68	6.64
4 resources	32	29	114	143	135.50	6.51	4.80
5 resources	32	29	154	183	168.81	7.75	4.59
6 resources	32	24	171	195	186.41	6.09	3.27

4.2.2 Container placement on virtual machines

The allocation of containers to virtual machines is determined by the placement strategy. The only way to determine the placement strategy is at creation time of the service. It cannot be modified after the creation of the service.

There is a choice of three placement strategies³⁴:

- Binpack – When a new task is spawned, it will be placed on the VM with least CPU or memory.
- Random – When a new task is spawned, it will be placed randomly on any available VM with sufficient resources.
- Spread – When a new task is spawned, it will be placed with the intention that tasks are spread evenly across available VMs.

The placement strategy is not displayed in the AWS web interface or accessible through the SDK, so once the service has been created, it is not possible to find out which placement strategy was applied. For the experiment conducted in this research, no placement strategy was specifically defined on creation of the service through the AWS SDK.

Contacting AWS support could not clarify which placement strategy is applied if no placement strategy was specified through the SDK. Different support agents gave different information, indicating that either random allocation or spread allocation is used under such circumstances.

Figure 4.9 shows the mean additional load that was handled at each resource level after adding an additional resource. The additional load was calculated using the below equation with n being the number of resources:

$$meanAdditionalLoad_{n,n+1} = meanLoad_{n+1} - meanLoad_n$$

Figure 4.9 also shows that the mean additional load fluctuates, with the 6th resource handling on average significantly less additional load than the previous containers. The other resource levels did not have significant differences, as shown by their overlapping error bars.

Future research should explore different placement strategies, to investigate the exact placement behaviour. Repeating the experiment with the placement strategy “binpack” may provide a different outcome and would be part of any future work.

³⁴ Amazon Web Services, Inc. Amazon ECS Task Placement Strategies. Retrieved April 13, 2018, from <https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task-placement-strategies.html>

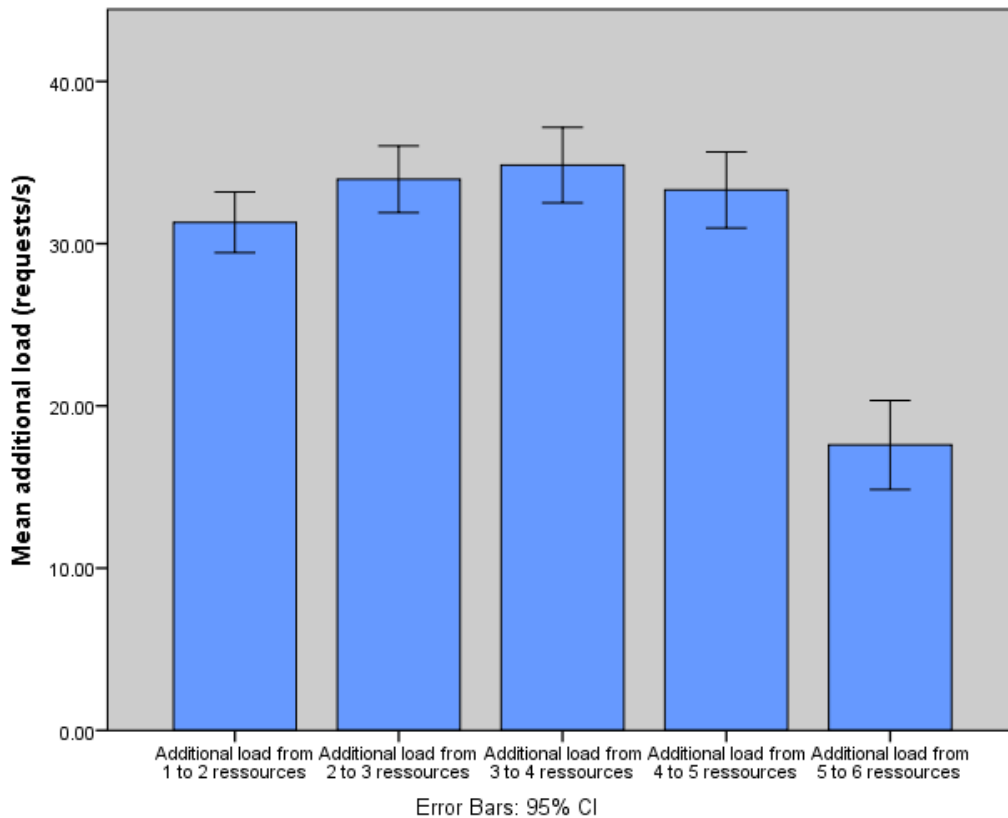


Figure 4.9: Mean additional load handled per new resource \pm standard error (31 df, $p < 0.05$)

4.2.3 CPU utilisation of individual containers

The reduced mean load handled by 6 resources compared to the previous resource levels (Figure 4.9) also raises the question how much CPU units a container can use. Conflicting information was obtained regarding this matter. Clarity was sought from AWS Support, who indicated a task cannot use more CPU units than allocated to it through the “Task Size” setting “CPU units”, which for the experiment was set to 450.

AWS documentation states that a container can use all CPU units of its hosting virtual machine, unless another container reserves them. This means if a VM is running a single container, it can use all its CPU units³⁵. This statement seems to be supported by the behaviour captured in Figure 4.9, where the 6th resource performs significantly less than the previous resources.

³⁵ Amazon Web Services, Inc. Task Definition Parameters. Retrieved April 13, 2018, from https://docs.aws.amazon.com/AmazonECS/latest/developerguide/task_definition_parameters.html

4.2.4 Hypothesis H-A

The following hypotheses will be tested:

H-A₀: With a probability of $\geq 95\%$ there is no statistically significant difference between the resource-load curves produced by running BUNGEE's system-analysis several times on same system.

H-A₁: With a probability of $\geq 95\%$ there is a statistically significant difference between the resource-load curves produced by running BUNGEE's system-analysis several times on same system.

To investigate the above hypotheses, the results were written in a wide format and analysed with a “repeated measures analysis of variance” (ANOVA). The analysis was performed with the software SPSS version 24³⁶. To visualise the wide format, an excerpt of the results in wide format can be seen in

Table 4.2.

The results of the analysis are as follows:

- Sphericity was met, so no correction was applied to the results. The results obtained follow a normal distribution, tested with the Shapiro-Wilk test of normality.
- The ANOVA could not detect a significant difference between the results of the different system analysis runs ($F(31, 124)=1.209$, $p=0.231$, $\eta^2 = 0.232$)³⁷.
- If the test considers the covariate “number of resources” along with the maximum load, the result still cannot show significant differences between the different system analyses ($F(31, 124)=1.541$, $p=0.051$, $\eta^2 = 0.278$).

Given the above, *H-A₀* cannot be rejected. This means it cannot be stated that with a probability of $\geq 95\%$ there is a statistically significant difference between the resource-load curves of the repeated measurement runs. However, the p-value surpasses the

³⁶ IBM SPSS Software. IBM Analytics. Retrieved from <https://www.ibm.com/analytics/data-science/predictive-analytics/spss-statistical-software>

³⁷ Explanation: $F(\text{<degrees of freedom>, <degrees of freedom residual sum of squares>})=\text{<Fvalue>}$, $p=\text{<Significance level>}$, $\eta^2(\text{eta square})=\text{<effect size>}$

acceptable value of 0.05 only by a very marginal 0.001, making the result marginally significant. Rounding the result for the p-value would make the difference significant.

It is worth noting that the observed effect sizes were small (0.232 and 0.278) although the power was high (0.912 and 0.974)³⁸. These results indicate that if there is a difference between the different measurement runs, it is a small difference. To support these results further, Hypothesis B is evaluated.

Table 4.2: Results in wide format for repeated measures ANOVA

Resources	runA	runB	runC	runD	runE	runF	runG	runH	runI	runJ
1	40	29	36	35	38	35	36	35	35	36
2	70	73	76	68	71	65	69	64	65	69
3	95	109	110	101	100	90	102	94	102	96
4	127	140	142	138	140	131	139	129	140	140
5	163	177	167	169	173	154	162	163	175	174
6	187	192	192	184	190	183	185	171	185	185

4.2.5 Hypothesis H-B

The following hypotheses will be tested:

H-B₀: The error of the system analysis is smaller than 5 % on a confidence interval of 95%.

H-B₁: The error of the system analysis is larger than 5% on a confidence interval of 95%.

The statistical test used by Weber (Weber, 2014, p. 76) will be applied to ensure comparability of results.

Let \bar{x} be the mean of several load intensity measurements for a given resource level. Let μ be the true population mean. To reject *H-B₀*, it must be shown μ deviates more than 5% from \bar{x} ($p \leq 0.05$).

³⁸ The probability of committing a type 2 error is 1-power

To maintain $H-B_0$, it must be shown that μ does not deviate more than 5% from \bar{x} ($p \leq 0.05$):

$$P(c_1 > \mu > c_2) \leq 0.05$$

c_1 will be defined as the lower boundary that μ must not fall below. c_2 will be defined as the upper boundary μ must not exceed to be able to state that μ does not deviate more than 5% from \bar{x} .

To calculate c_1 and c_2 :

$$c_1 = \bar{x} * 0.95$$

$$c_2 = \bar{x} * 1.05$$

c_{low} is defined as the lower bound of the 95% confidence interval for the population mean μ , as obtained by a one-sample, two tailed t-test. c_{high} is defined as the higher bound of the 95% confidence interval for the true mean, obtained by the same t-test.

The formula to be used is:

$$CI_{95} = \bar{x} \pm \left(t_{\frac{\alpha}{2}} \right) * \frac{\tilde{s}}{\sqrt{n}} \quad (\text{Sheskin, 2000, p. 81})$$

With \tilde{s} being the unbiased estimate of the population standard deviation (Sheskin, 2000, p. 7)

$$CI_{low\ 95} = \bar{x} - \left(t_{\frac{\alpha}{2}} \right) * \frac{\tilde{s}}{\sqrt{n}}$$

$$CI_{high\ 95} = \bar{x} + \left(t_{\frac{\alpha}{2}} \right) * \frac{\tilde{s}}{\sqrt{n}}$$

To maintain $H-B_0$, the following must be true:

$$c_1 < CI_{low\ 95} \text{ and } c_2 > CI_{high\ 95}$$

Table 4.3 shows the results of the above calculations. The results show that the conditions to maintain $H-B_0$ are true for all resources. Results follow a normal distribution, tested with the Shapiro-Wilk test except for 4 and 6 resources. However, as the t-test is robust to violation of normality at a large enough sample size of $> 30 - 40$ (Ghasemi & Zahediasl, 2012), the results for resource levels 4 and 6 might still be considered relevant.

H-B₀ “The error of the system analysis is smaller than 5 % of on a confidence level of 95%” must therefore be maintained, indicating that the performance fluctuation of the system analysis is within the levels deemed acceptable.

Table 4.3: Mean load, confidence intervals and boundaries for accepted error

					95% Confidence Interval of μ		
	Mean load \bar{x}	t	df	C ₁	CI _{low}	CI _{high}	C ₂
1 resource	35.38	54.92	31	33.60	34.06	36.69	37.14
2 resources	66.69	80.61	31	63.35	65.00	68.37	70.02
3 resources	100.66	85.24	31	95.62	98.25	103.06	105.69
4 resources	135.50	117.82	31	128.73	133.15	137.85	142.28
5 resources	168.81	123.19	31	160.37	166.02	171.61	177.25
6 resources	186.41	173.15	31	177.09	184.21	188.60	195.73

4.2.6 Results in comparison to existing research

This section compares the results obtained from the experiments in this dissertation with results published in the literature (Iosup et al., 2011), (Leitner & Cito, 2016), (Weber, 2014). A review of similar research suggests that this is the first work attempting to apply an elasticity benchmark to a container-based system. However, other works have conducted measurements in public cloud systems and investigated performance variation (Iosup et al., 2011) (Leitner & Cito, 2016). The original work presenting the BUNGEE framework conducted the same experiment as this dissertation, but in a private cloud IaaS system (Weber, 2014) instead of a public cloud container-based system where the experiment conducted for this dissertation was carried out.

Weber (Weber, 2014, p. 77) measured the performance variation of the system analysis phase in a private cloud. They found relative standard deviations between 0 and 1.6%. A private cloud is not affected by the inherent variability of performance in public clouds (see 2.5 “Performance variability in public clouds”). Weber’s work shows that the original BUNGEE benchmark system analysis runs stably in a private cloud environment.

Leitner & Cito (2016, p. 10) indicate the relative standard deviation they found for CPU-heavy workloads as being between 0.16% and 20.28%. For m1.small instances, as used

in this dissertation, they observed relative standard deviations of 12.18%. The results of the experiments conducted for this dissertation have a relative standard deviation between 3.27% and 10.33%. The relative standard deviation allows comparing the variation of two distributions with different means and different units (Sheskin, 2000, p. 10), therefore it is suitable for comparing the results achieved in the two studies.

In conclusion the performance variation in CPU-heavy workloads conducted with AWS m1.small instances encountered in this dissertation's experiment was smaller than the performance variation encountered in Leitner & Cito's (2016) research. This indicates that the performance variation encountered in the present experiments might have been mostly due to the inherent performance variability in public clouds and not due to the changes introduced to adapt BUNGEE to ECS.

An additional experiment conducted during this dissertation supports the assumption that the inherent performance variability of m1.small EC2 instances might be the main reason for the encountered performance variation. The results found are beyond the scope of this dissertation but can be found in Appendix IV and V.

4.2.7 Results of a BUNGEE benchmark run

To verify if the extension created to adapt BUNGEE to the Elastic Container Service works not only for the system analysis phase, but also for the measurement phase, the entire benchmark was conducted once with an SLO of 500ms.

The results show that the system used up to 6 tasks (Figure 4.10, lower graph) and spent a relatively large amount of time in an under- and over-provisioned state (Figure 4.10, upper graph). The results further show that the system scaled up to three virtual machines, which means the system estimated being able to handle all workload with three virtual machines.

The metrics calculated by the benchmark are listed in Table 4.4. It is important to note that these results are not representative of the AWS ECS system, as the benchmark would have to be conducted multiple times, and the mean values employed. Additionally, the thresholds for increasing and decreasing the number of containers and VMs were selected without much investigation into the ideal values. The purpose of running the benchmark was solely to verify that the code extension works.

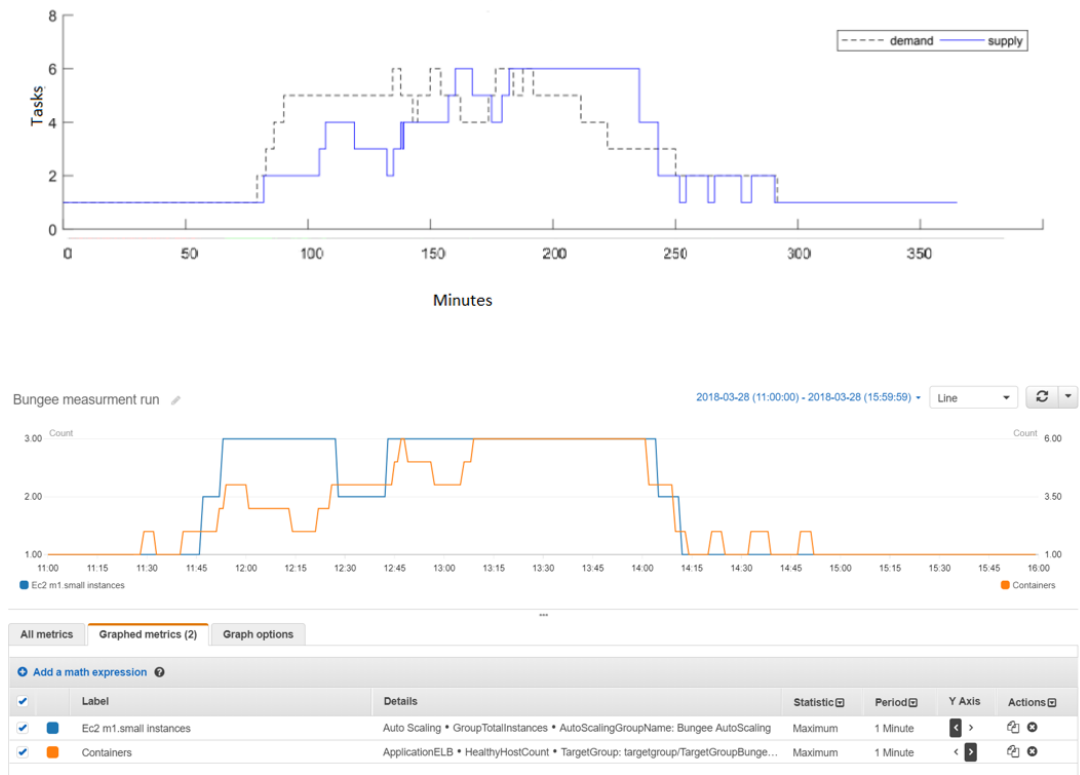


Figure 4.10: Benchmark results ECS

Table 4.4: Benchmark run results

Over-provisioning accuracy	Under-provisioning accuracy	Over-provisioning timeshare	Under-provisioning timeshare	Jitter
0.306	0.467	0.190	0.278	0.017

5 ANALYSIS, EVALUATION AND DISCUSSION

This chapter discusses the results described in 4.2 “Results” and puts them into context with current knowledge. Further, possible explanations for the results will be explored and discussed for future research. Potential adaptations of the BUNGEE framework towards containers will be explored along with the difficulties such an adaptation could pose.

5.1 Discussion of results

The results of the two hypotheses are somewhat conflicting. For hypothesis H-A, when considering the number of containers as covariate, ANOVA determined a marginally significant difference between the different benchmark runs at a 95% confidence interval. When not using the number of containers as a covariate the test did not determine a significant difference. For hypothesis H-B, the t-test found that with a probability of $\geq 95\%$, the true mean load handled by each resource level does not deviate more than 5% from the sample mean, indicating that the adapted BUNGEE benchmark yields acceptably stable results. The above indicates that there is some variance between the different runs of the system analysis, which however does not exceed the level determined as acceptable.

Given that some performance variability was observed, even if not exceeding the acceptable level, it would be appropriate to conduct further research towards the cause of this variability. The variability could be caused by various factors. Three of them were explored in this dissertation but deserve further research. Possible causes explored were:

1. The performance variability inherent to cloud systems
2. Varying allocation of CPU units to individual containers
3. Varying allocation of containers to virtual machines.

To determine the exact impact of each of the above factors on the results, further research is needed.

Given the obtained results in this dissertation, it is likely that BUNGEE is suited to be used for systems using containers, although the above listed factors pose potential problems for using it in this context.

The inherent variability of results in cloud environments as discussed in section 2.5 “Performance variability in public clouds” might require researchers to run several system analyses and several benchmarks to obtain reliable results. Ideally, interleaved random trials, as discussed in section 2.5, should be employed.

The effect of inherent cloud performance variation might be compounded by the uncertainties the container orchestration systems causes. Uncertainties were found in the allocation of containers to virtual or physical machines and the allocation of CPU shares from the host system to the containers.

For the experiment presented in this dissertation, a moderate amount of insight into the underlying allocation of containers to virtual machines was available. The placement strategy of containers to virtual machines can be configured, although it remained unclear which placement strategy is used if no placement strategy was specified when the service was created (see 4.2.2 “Container placement on virtual machines”). The number of CPU shares per container and task can be configured, but the exact number of CPU shares a container can use under the conditions of this experiment remained unclear. Conflicting information was obtained from different sources.

Other CaaS systems (e.g. AWS ECS with Fargate launch type) and PaaS systems which use container orchestration platforms “under the hood” (e.g. AWS Elastic Beanstalk) do not provide granular information or the possibility to configure the container-to-VM allocation or CPU shares of a VM. This could cause performance variability with difficult to identify causes.

Therefore, it seems to be reasonable to assume a certain degree of performance variability will always be present and to structure experiments based on this assumption.

5.2 Suggestions for further research

This dissertation was a first proof of concept to test if the BUNGEE benchmark can be applied to a cloud system based on containers and container orchestration. For logistic reasons, the experiment was only conducted with one set of parameters such as virtual machine type, settings in the ECS task definition and the configuration of the service. For this reason, further research would be of interest such as:

- Researching if the BUNGEE system analysis conducted on Amazon EC2 without ECS fluctuates to the same degree as the research conducted in this

experiment with ECS. An initial test was conducted. Results can be found in Appendix IV and V.

- Repeating the experiment conducted in this research with the container placement setting “binpack”.
- Repeating the experiment conducted in this research with a different virtual machine type and observing if the results fluctuate by the same amount.
- Conducting the same experiment on a different CaaS platforms.
- Conducting the same experiment in a private cloud with a setup involving Docker and Kubernetes or Docker Swarm.

To better evaluate the elasticity of systems using containers, some extensions to the BUNGEE framework might be considered.

Through the existing interfaces, BUNGEE makes it very easy to create extensions for further IaaS systems and even CaaS systems through the method applied in this dissertation. The method of extension used in this dissertation unfortunately causes some data to be lost: This loss occurs where BUNGEE monitors the number of containers instead of virtual machines. The number of VMs accommodating the containers then remains unknown or must be extracted separately from the cloud system. Creating interfaces to easily capture custom metrics such as number of containers or container to VM allocation might be a worthwhile extension to the BUNGEE framework. These could be realised by creating interfaces to monitor custom, user defined metrics at user specified intervals. With such a custom metric, the number of containers and virtual machines could be monitored at the same time.

6 CONCLUSION

Application architectures are shifting towards using containerised microservices to host loosely coupled services in the cloud, thereby CaaS systems or self-managed container orchestration systems are growing in popularity.

As containers have only become popular within the last four years, research in this area is still scarce. This experiment, to the researcher's best knowledge, was the first attempt at measuring the elasticity of containerised services.

The goal of this research was to determine if existing tools to measure elasticity of IaaS systems can also measure the elasticity of containerised systems. To achieve this, an existing and proven microbenchmark for measuring elasticity in IaaS systems was adapted to Amazon's Elastic Container System. Funding in the form of AWS credits was secured through an application to the "AWS Cloud Credits for Research Program"³⁹.

To validate that the adapted benchmark yields reproducible results, the system analysis phase was run 32 times. The statistical tests ANOVA and T-Test were performed, testing if the results are reproducible.

The ANOVA test did not find significant differences between the 32 iterations of the benchmark. However, the results when considering the number of resources as covariate were marginally significant. The t-test determined that the confidence interval for the true mean of all resource levels did not deviate more than 5% from the sample mean, meaning the system analysis yields stable results. Despite the level of performance variation encountered in these results being within the levels defined as acceptable, an analysis into the causes of this performance variation was deemed necessary.

An initial investigation was carried out to determine possible causes of performance fluctuations. Three potential causes were identified:

- The system-inherent performance variability that other researchers have encountered in cloud environments.

³⁹ Amazon Web Services, Inc. AWS Cloud Credits for Research FAQ. Retrieved June 10, 2018, from <https://aws.amazon.com/research-credits/faq/>

- The placement of containers on virtual machines, possibly leading to a container having more or less system resources available during different iterations and stages of the analysis.
- The unclear number of CPU units available to a container during various stages of the experiment.

These potential causes of performance variation need to be verified in additional experiments.

A comparison with existing research on performance variability in cloud systems was undertaken. Other researchers found even bigger levels of performance variation, so a likely cause of the encountered fluctuation is the performance variation inherent to cloud systems.

The contributions of this work are:

- An understanding was developed that an existing microbenchmark, the BUNGEE benchmark, is likely suited for measuring elasticity in Container as a Service environments. However, there are some uncertainties that require further investigation.
- Possible causes of these uncertainties were identified, and suggestions were made which experiments could clarify them.
- Suggestions for features to include in the next version of the BUNGEE benchmark were made, to better accommodate measuring the elasticity of systems based on containers.

Recommendations for future research have been listed in 5.2 and consist mainly of repeating the experiment presented in this work with different parameters, conducting it in private and public containerised cloud systems and investigating the performance variability in the cloud further.

BIBLIOGRAPHY

- Abedi, A., & Brecht, T. (2017). Conducting Repeatable Experiments in Highly Variable Cloud Computing Environments. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering* (pp. 287–292). New York, NY, USA: ACM. <https://doi.org/10.1145/3030207.3030229>
- Al-Dhuraibi, Y., Paraiso, F., Djarallah, N., & Merle, P. (2017). Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing*, 1–1. <https://doi.org/10.1109/TSC.2017.2711009>
- Almeida, R. F., Sousa, F. R., Lifschitz, S., & Machado, J. C. (2013). On defining metrics for elasticity of cloud databases. In *SBB D (Short Papers)* (pp. 12–1).
- Anwar, A., Cheng, Y., & Butt, A. R. (2016). Towards Managing Variability in the Cloud. In *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)* (pp. 1081–1084). <https://doi.org/10.1109/IPDPSW.2016.62>
- Armbrust, M., Fox, A., Griffith, R., Joseph, A. D., Katz, R., Konwinski, A., ... Zaharia, M. (2010). A View of Cloud Computing. *Commun. ACM*, 53(4), 50–58. <https://doi.org/10.1145/1721654.1721672>
- Bauer, E., & Adams, R. (2012). *Reliability and availability of cloud computing*. Piscataway, NJ : Hoboken, NJ: IEEE Press ; Wiley.
- Baun, C., Kunze, M., Nimis, J., & Tai, S. (2011). *Cloud Computing: Web-Based Dynamic IT Services*. Berlin Heidelberg: Springer-Verlag. Retrieved from <http://www.springer.com/gp/book/9783642209161>
- Bellenger, D., Bertram, J., Budina, A., Koschel, A., Pfänder, B., Serowy, C., ... Schaaf, M. (2011). Scaling in Cloud Environments. In *Proceedings of the 15th WSEAS International Conference on Computers* (pp. 145–150). Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS). Retrieved from <http://dl.acm.org/citation.cfm?id=2028299.2028329>
- Beltrán, M. (2016). BECloud: A new approach to analyse elasticity enablers of cloud services. *Future Generation Computer Systems*, 64, 39–49. <https://doi.org/10.1016/j.future.2016.05.014>

- Bernstein, D. (2014). Containers and Cloud: From LXC to Docker to Kubernetes. *IEEE Cloud Computing*, 1(3), 81–84. <https://doi.org/10.1109/MCC.2014.51>
- Binnig, C., Kossmann, D., Kraska, T., & Loesing, S. (2009). How is the weather tomorrow?: towards a benchmark for the cloud (p. 9). Presented at the Proceedings of the Second International Workshop on Testing Database Systems, ACM. <https://doi.org/10.1145/1594156.1594168>
- Casalicchio, E. (2017). Autonomic Orchestration of Containers: Problem Definition and Research Challenges. In *Proceedings of the 10th EAI International Conference on Performance Evaluation Methodologies and Tools on 10th EAI International Conference on Performance Evaluation Methodologies and Tools* (pp. 287–290). ICST, Brussels, Belgium, Belgium: ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). <https://doi.org/10.4108/eai.25-10-2016.2266649>
- Casalicchio, E., & Perciballi, V. (2017). Measuring Docker Performance: What a Mess!!! In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (pp. 11–16). New York, NY, USA: ACM. <https://doi.org/10.1145/3053600.3053605>
- Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., & Sears, R. (2010). Benchmarking Cloud Serving Systems with YCSB. In *Proceedings of the 1st ACM Symposium on Cloud Computing* (pp. 143–154). New York, NY, USA: ACM. <https://doi.org/10.1145/1807128.1807152>
- Coulouris, G., Dollimore, J., Kindberg, T., & Blair, G. (2011). *Distributed Systems: Concepts and Design* (5 edition). Boston: Pearson.
- Coutinho, E. F., de Carvalho Sousa, F. R., Rego, P. A. L., Gomes, D. G., & de Souza, J. N. (2015). Elasticity in cloud computing: a survey. *Annals of Telecommunications - Annales Des Télécommunications*, 70(7–8), 289–309. <https://doi.org/10.1007/s12243-014-0450-7>
- Coutinho, E. F., Sousa, F. R. de C., Rego, P. A. L., Gomes, D. G., & Souza, J. N. de. (2015). Elasticity in cloud computing: a survey. *Annals of Telecommunications - Annales Des Télécommunications*, 70(7–8), 289–309. <https://doi.org/10.1007/s12243-014-0450-7>

- Dory, T., Mejías, B., Roy, P. V., & Tran, N.-L. (2011). Measuring elasticity for cloud databases. In *Proceedings of the The Second International Conference on Cloud Computing, GRIDs, and Virtualization*.
- Ferdman, M., Adileh, A., Kocberber, O., Volos, S., Alisafae, M., Jevdjic, D., ... Falsafi, B. (2012). 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* (pp. 37–48). New York, NY, USA: ACM. <https://doi.org/10.1145/2150976.2150982>
- Folkerts, E., Alexandrov, A., Sachs, K., Iosup, A., Markl, V., & Tosun, C. (2012). Benchmarking in the Cloud: What It Should, Can, and Cannot Be. In *Selected Topics in Performance Evaluation and Benchmarking* (pp. 173–188). Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-36727-4_12
- Ghasemi, A., & Zahediasl, S. (2012). Normality Tests for Statistical Analysis: A Guide for Non-Statisticians. *International Journal of Endocrinology and Metabolism*, 10(2), 486–489. <https://doi.org/10.5812/ijem.3505>
- Govindan, S., Liu, J., Kansal, A., & Sivasubramaniam, A. (2011). Cuanta: Quantifying Effects of Shared On-chip Resource Interference for Consolidated Virtual Machines. In *Proceedings of the 2Nd ACM Symposium on Cloud Computing* (pp. 22:1–22:14). New York, NY, USA: ACM. <https://doi.org/10.1145/2038916.2038938>
- Herbst, N., Kounev, S., & Reussner, R. (2013). Elasticity in Cloud Computing: What it is, and What it is Not.
- Herbst, N., Kounev, S., Weber, A., & Groenda, H. (2015). BUNGEE: An Elasticity Benchmark for Self-adaptive IaaS Cloud Environments. In *Proceedings of the 10th International Symposium on Software Engineering for Adaptive and Self-Managing Systems* (pp. 46–56). Piscataway, NJ, USA: IEEE Press. Retrieved from <http://dl.acm.org/citation.cfm?id=2821357.2821366>
- Huang, S., Huang, J., Dai, J., Xie, T., & Huang, B. (2010). The HiBench benchmark suite: Characterization of the MapReduce-based data analysis. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)* (pp. 41–51). <https://doi.org/10.1109/ICDEW.2010.5452747>

- Iosup, A., Yigitbasi, N., & Epema, D. (2011). On the Performance Variability of Production Cloud Services. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing* (pp. 104–113). <https://doi.org/10.1109/CCGrid.2011.22>
- Islam, S., Lee, K., Fekete, A., & Liu, A. (2012). How a consumer can measure elasticity for cloud platforms (p. 85). ACM Press. <https://doi.org/10.1145/2188286.2188301>
- Jennings, B., & Stadler, R. (2015). Resource Management in Clouds: Survey and Research Challenges. *Journal of Network and Systems Management*, 23(3), 567–619. <https://doi.org/10.1007/s10922-014-9307-7>
- Khan, A. (2017). Key Characteristics of a Container Orchestration Platform to Enable a Modern Application. *IEEE Cloud Computing*, 4(5), 42–48. <https://doi.org/10.1109/MCC.2017.4250933>
- Kounev, S. (2007). Software Performance Evaluation. *Wiley Encyclopedia of Computer Science and Engineering*. <https://doi.org/10.1002/9780470050118.ecse390>
- Kratzke, N., & Peinl, R. (2016). ClouNS - a Cloud-Native Application Reference Model for Enterprise Architects. In *2016 IEEE 20th International Enterprise Distributed Object Computing Workshop (EDOCW)* (pp. 1–10). <https://doi.org/10.1109/EDOCW.2016.7584353>
- Leitner, P., & Cito, J. (2016). Patterns in the Chaos—A Study of Performance Variation and Predictability in Public IaaS Clouds. *ACM Trans. Internet Technol.*, 16(3), 15:1–15:23. <https://doi.org/10.1145/2885497>
- Li, A., Yang, X., Kandula, S., & Zhang, M. (2010). CloudCmp: Comparing Public Cloud Providers. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement* (pp. 1–14). New York, NY, USA: ACM. <https://doi.org/10.1145/1879141.1879143>
- Li, Z. (Eddie), Zhang, H., O'Brien, L., Cai, R., & Flint, S. (2013). On Evaluating Commercial Cloud Services: A Systematic Review. *Journal of Systems and Software*, 86, 2371–2393. <https://doi.org/10.1016/j.jss.2013.04.021>
- Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. Computer Security Division, Information Technology Laboratory, National Institute of Standards

and Technology Gaithersburg. Retrieved from <http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf>

Moldovan, D., Copil, G., Truong, H. L., & Dustdar, S. (2013). MELA: Monitoring and Analyzing Elasticity of Cloud Services. In *2013 IEEE 5th International Conference on Cloud Computing Technology and Science* (Vol. 1, pp. 80–87). <https://doi.org/10.1109/CloudCom.2013.18>

Muñoz-Escoí, F. D., & Bernabéu-Aubán, J. M. (2017). A survey on elasticity management in PaaS systems. *Computing*, 99(7), 617–656. <https://doi.org/10.1007/s00607-016-0507-8>

Pahl, C. (2015). Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3), 24–31. <https://doi.org/10.1109/MCC.2015.51>

Piraghaj, S. F., Dastjerdi, A. V., Calheiros, R. N., & Buyya, R. (2015). A Framework and Algorithm for Energy Efficient Container Consolidation in Cloud Data Centers. In *2015 IEEE International Conference on Data Science and Data Intensive Systems* (pp. 368–375). <https://doi.org/10.1109/DSDIS.2015.67>

Rauh, F., & Herbst, N. (2015, July 7). BUNGEE Quick Start Guide for AWS EC2 based elastic clouds.

Rosenberg, J. B., & Mateos, A. (2011). *The cloud at your service: the when, how, and why of enterprise cloud computing*. Greenwich, Conn: Manning.

Salkind, N. J. (2017). *Statistics for people who (think they) hate statistics* (6th edition, international student edition). Los Angeles London New Delhi Singapore Washington DC Melbourne: SAGE.

Shawky, D. M., & Ali, A. F. (2012). Defining a measure of cloud computing elasticity. In *2012 1st International Conference on Systems and Computer Science (ICSCS)* (pp. 1–5). <https://doi.org/10.1109/IConSCS.2012.6502449>

Sheskin, D. (2000). *Handbook of parametric and nonparametric statistical procedures* (2nd ed). Boca Raton: Chapman & Hall/CRC.

Sheskin, D. (2007). *Handbook of parametric and nonparametric statistical procedures* (4th ed). Boca Raton: Chapman & Hall/CRC.

- SPEC. (2016, July 28). SPEC CloudTM IaaS 2016 Benchmark Design Overview. Retrieved from https://www.spec.org/cloud_iaas2016/docs/designoverview.pdf
- Suleiman, B. (2012). Elasticity Economics of Cloud-Based Applications. In *2012 IEEE Ninth International Conference on Services Computing* (pp. 694–695). <https://doi.org/10.1109/SCC.2012.65>
- v. Kistowski, J., Arnold, J. A., Huppler, K., Lange, K.-D., Henning, J. L., & Cao, P. (2015). How to Build a Benchmark. In *Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering* (pp. 333–336). New York, NY, USA: ACM. <https://doi.org/10.1145/2668930.2688819>
- Varadarajan, V., Kooburat, T., Farley, B., Ristenpart, T., & Swift, M. M. (2012). Resource-freeing Attacks: Improve Your Cloud Performance (at Your Neighbor's Expense). In *Proceedings of the 2012 ACM Conference on Computer and Communications Security* (pp. 281–292). New York, NY, USA: ACM. <https://doi.org/10.1145/2382196.2382228>
- Varghese, B., Akgun, O., Miguel, I., Thai, L., & Barker, A. (2014). Cloud Benchmarking for Performance. In *2014 IEEE 6th International Conference on Cloud Computing Technology and Science* (pp. 535–540). <https://doi.org/10.1109/CloudCom.2014.28>
- Vazquez, C., Krishnan, R., & John, E. (2014). Cloud Computing Benchmarking: A Survey, 6.
- Weber, A. (2014, July). *Resource Elasticity Benchmarking in Cloud Environments*. Karlsruhe Institute of Technology, Karlsruhe. Retrieved from <https://sdqweb.ipd.kit.edu/publications/pdfs/Weber2014.pdf>
- Zhang, Q., Cheng, L., & Boutaba, R. (2010). Cloud computing: state-of-the-art and research challenges. *Journal of Internet Services and Applications*, 1(1), 7–18. <https://doi.org/10.1007/s13174-010-0007-6>

APPENDIX I – DATA CAPTURED BY THE BUNGEE BENCHMARK

Data captured during BUNGEE system analysis phase

File	Content
timestamps.csv	Contains one column with the times at which a request should be sent out. E.g. at intensity 20 there would be 20 requests/second, meaning requests are scheduled to be sent out n seconds after the start of the experiment, with n= {0.0; 0.100; 0.150; 0.200.....0.850; 0.900; 0.950}
responses.csv	This file contains one row for every single request that was made to the server. This file contains tens of thousands of rows.
	id ID of the request that was made
	startWork Start timestamp of calculation of Fibonacci sequence
	endWork End timestamp of calculation of the Fibonacci sequence
	duration Calculation: (endWork – startWork)
	result result of Fibonacci calculation
	timerstart timestamp when the measurement started
	start timestamp request sent
	end timestamp response received or error code
	responseTime Calculation: (end-start)
	latency response time in seconds if response received. Else 0

	failed	0 if request failed, 1 if it succeeded
	responseCode	HTTP response code
	servierIP	IP of responding instance
mapping.mapping	<p>This file contains the mapping of intensity to resources.</p> <p>Example:</p> <p>maxIntensity;resourceAmount</p> <p>160.0;1</p> <p>180.0;2</p>	

APPENDIX II – AWS CONFIGURATION

EC2 configuration overview

EC2 settings	
Region	EU(Ireland)
Instance Availability Zone	EU-West-1b
Instance Types	m1.small
Instance AMI	ami-64c4871d
EC2 Application Load	<ul style="list-style-type: none">• Idle timeout 60s
Balancer settings	<ul style="list-style-type: none">• Listener: 8080
EC2 Target Group settings	<ul style="list-style-type: none">• Port 8080• Target Type: Instance• Deregistration delay 300s• Stickiness: disabled
Load Balancer security group	Inbound Ports: 8080, 80, 22
Instances security group	Inbound ports: 32768 – 65535, 8080, 22
EC2 instances time synchronization	Chrony ^{40, 41} installed in EC2 instances.
Instance count	5 instances during system analysis

⁴⁰ See 3.2.5 “Load driver machine specifications”.

⁴¹ Churnow, R., & Lichvar, M. (2017). Chrony (Version 2). Retrieved from <https://chrony.tuxfamily.org/>

ECS configuration overview

ECS settings

Task definition	<ul style="list-style-type: none">• Task memory (MiB) 450• Task CPU (unit) 450• Container Definition:<ul style="list-style-type: none">○ Self-created Ubuntu 16.04 Docker image which contains the “SimpleHTTP” example contained in the BUNGEE source code.
-----------------	--

ECS Service settings	<ul style="list-style-type: none">• Min healthy percent: 50• Max percent: 200• Health Check Grace Period 0• Task placement strategy: default = random
----------------------	--

Container definition	<ul style="list-style-type: none">• CPU units: 400• Hard/ Soft memory limits: none• Port Mapping Host:Container 0:8080
----------------------	--

APPENDIX III – RESULTS OF THE SYSTEM ANALYSIS - ECS

Date	Load driver	1_res	2_res	3_res	4_res	5_res	6_res
run_A_20180314	Physical Machine	40	70	95	127	163	187
run_B_20180315	Physical Machine	29	73	109	140	177	192
run_C_20180320	Physical Machine	36	76	110	142	167	192
run_D_20180321	Physical Machine	35	68	101	138	169	184
run_E_20180322	Physical Machine	38	71	100	140	173	190
run_F_20180329	Physical Machine	35	65	90	131	154	183
run_G_20180403	Physical Machine	36	69	102	139	162	185
run_H_20180404	Physical Machine	35	64	94	129	163	171
run_I_20180405	Physical Machine	35	65	102	140	175	185
run_j_20180406	Physical Machine	36	69	96	140	174	185
20180519_00.05	VM	35	53	93	124	155	171
20180519_15.13	VM	31	63	103	141	181	188
20180519_21.18	VM	35	67	100	134	169	179
20180520_02.30	VM	35	69	98	138	173	195
20180520_22.44	VM	38	67	104	139	169	185
20180521_08.45	VM	40	63	90	138	172	190
20180521_15.19	VM	36	70	105	137	162	191
20180522_09.00	VM	34	62	99	128	166	191
20180522_18.19	VM	27	66	104	138	162	193
20180523_00.06	VM	30	63	99	114	162	190
20180523_09.36	VM	28	62	84	130	166	190
20180523_15.58	VM	36	60	109	141	169	185
20180524_00.02	VM	39	73	112	142	182	194
20180524_09.06	VM	42	61	104	134	165	174
20180524_18.43	VM	41	72	107	142	162	187
20180525_17.45	VM	40	72	112	143	183	189
20180526_01.30	VM	35	66	96	128	156	182
20180526_09.07	VM	36	68	104	137	181	191
20180526_15.16	VM	37	70	103	138	170	193
20180526_22.07	VM	31	65	98	138	176	185
20180527_03.33	VM	34	63	93	137	176	184
20180527_10.44	VM	37	69	105	129	168	184

APPENDIX IV – RESULTS OF THE SYSTEM ANALYSIS - EC2

Date_time	1_res	2_res	3_res	4_res	5_res	6_res
20180519_00.05	35	53	93	124	155	171
20180519_15.13	31	63	103	141	181	188
20180519_21.18	35	67	100	134	169	179
20180520_02.30	35	69	98	138	173	195
20180520_22.44	38	67	104	139	169	185
20180521_08.45	40	63	90	138	172	190
20180521_15.19	36	70	105	137	162	191
20180522_09.00	34	62	99	128	166	191
20180522_18.19	27	66	104	138	162	193
20180523_00.06	30	63	99	114	162	190
20180523_09.36	28	62	84	130	166	190
20180523_15.58	36	60	109	141	169	185
20180524_00.02	39	73	112	142	182	194
20180524_09.06	42	61	104	134	165	174
20180524_18.43	41	72	107	142	162	187
20180525_17.45	40	72	112	143	183	189
20180526_01.30	35	66	96	128	156	182
20180526_09.07	36	68	104	137	181	191
20180526_15.16	37	70	103	138	170	193
20180526_22.07	31	65	98	138	176	185
20180527_03.33	34	63	93	137	176	184
20180527_10.44	37	69	105	129	168	184

APPENDIX V – ANALYSIS RESULTS EC2 ONLY

The experiment conducted in this dissertation was repeated without the ECS extension developed for this dissertation to investigate the performance variability inherent to the AWS cloud system. The AWS environment was set up according to BUNGEE quick start guide (Rauh & Herbst, 2015). M1.small instances were used.

Descriptive Statistics

	N	Mean	Std. Deviation	Std. Error Mean	Relative Std deviation
1 resource	20	69.85	13.880	3.104	19.87147
2 resources	20	128.45	25.442	5.689	19.8072
3 resources	20	177.10	24.630	5.507	13.90722
4 resources	20	246.00	26.206	5.860	10.65271
5 resources	20	301.70	33.098	7.401	10.97053
6 resources	20	375.50	19.856	4.440	5.287899

Results of t-test

The distribution of the values is not normal in this case and not sufficient samples were captured to ensure the t-test is robust against violation of normality as described in Ghasemi & Zahediasl (2012). Therefore below results are to be viewed with caution.

	Mean load \bar{x}	t	df	C1	95% Confidence Interval of μ		C2
					CIlow	CIhigh	
1 resource	69.85	22.51	19	66.36	63.35	76.35	80.16
2 resources	128.45	22.58	19	122.03	116.54	140.36	147.38
3 resources	177.10	32.16	19	168.25	165.57	188.63	198.06
4 resources	246.00	41.98	19	233.70	233.74	258.26	271.18
5 resources	301.70	40.76	19	286.62	286.21	317.19	333.05
6 resources	375.50	84.57	19	356.73	366.21	384.79	404.03

Results of the ANOVA

The distribution of the results was sufficiently normal for the purposes of conducting an ANOVA as tested by the Shapiro-Wilk test. The ANOVA detected a significant difference between the results of the different system analysis runs when the experiment is conducted with virtual machines instead of containers as resources ($F(19, 76) = 2.314$, $p = 0.005$, $\eta^2 = 0.367$). Power: 0.983.

If the test considers the covariate “number of resources” along with the maximum load, the result cannot show significant differences between the different system analyses ($F(19, 76) = 1.430$, $p = 0.139$, $\eta^2 = 0.263$). Power: 0.847.

APPENDIX VI – COMPARISON VIRTUAL & PHYSICAL LOAD DRIVER

The difference between the results obtained by running the experiment from a physical load driver (DIT library) versus vs on a virtual load driver (Private cloud Univ. of Würzburg) are explored in this appendix. The results of the different groups were tested for normality and were not normal. Therefore, the non-parametric Mann Whitney U test was employed instead of an independent sample t-test.

Maximum load per resource level did not differ significantly between a virtual machine or a physical machine as a load driver. The test statistic, p value, z-score and mean ranks can be found in the Table “Results Mann-Whitney-U test.

Further, box-plots can be found at the end of this appendix to visualise the similarity of the obtained results. The circle in the box-plot signifies a mild outlier while the asterisk signifies a strong outlier.

Results Mann-Whitney-U test

Statistic	1 res.	2 res.	3 res.	4 res.	5 res.	6 res.
U	109.00	65.00	127.00	89.50	118.00	129.00
z	-0.41	-1.84	0.69	-0.84	0.33	0.78
p	0.984	0.070	0.509	0.411	0.764	0.458
Mean rank VM	16.60	21.00	14.80	18.55	15.70	14.60
Mean rank physical machine	16.45	14.45	17.27	15.57	16.86	17.36

