Flexible Framework for Elasticity in Cloud Computing

Yahya Al-Dhuraibi

▶ To cite this version:

Yahya Al-Dhuraibi. Flexible Framework for Elasticity in Cloud Computing. Computer Science [cs]. Université lille1, 2018. English. NNT : . tel-02011337

HAL Id: tel-02011337
https://theses.hal.science/tel-02011337v1

Submitted on 7 Feb 2019

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Flexible Framework for Elasticity in Cloud Computing

Yahya Al-Dhuraibi

Supervisor: Dr. Philippe Merle

University of Lille
Prepared at Inria Lille - Nord Europe

This dissertation is submitted for the degree of
Doctor of Philosophy in Computer Science

Thesis Committee:
Dr. Fabienne BOYER, Dr. Thomas LEDOUX (referees)
Prof. Nouredine MELAB, Prof. Sara BOUCHENAK, Prof. Walid GAALOUL, Dr. Nabil DJARALLAH, Dr. Faiez ZALILA (examiners)

University of Lille — Monday 10th December, 2018

Abstract

Recently, cloud computing has been gaining more popularity and has received a great deal of attention from both industrial and academic worlds. Industries and application providers have moved or plan to move to clouds in order to focus on their core business. This frees them from the burden and cost of managing their physical servers in local data center infrastructures. However, the main factor motivating the use of cloud computing is its ability to provide resources according to the customer's needs or what is referred to as elastic provisioning and de-provisioning. Therefore, elasticity is one of the key features in cloud computing that dynamically adjusts the amount of allocated resources to meet changes in workload demands.

The workload of cloud applications usually varies drastically over time and hence maintaining sufficient resources to meet peak requirements can be costly, and will increase the application provider's functional cost. Conversely, if providers cut the costs by maintaining only a minimum computing resources, there will not be sufficient resources to meet peak requirements and cause bad performance, violating Service Level Agreement (SLA). Therefore, adapting cloud applications during their execution according to demand variation is a challenging task. In addition, cloud elasticity is diverse and heterogeneous because it encompasses different approaches, policies, purposes, and applications. Furthermore, elasticity can be applied at the infrastructure level or application level. The infrastructure is powered by a certain virtualization technology such as VMware, Xen, containers or a provider-specific virtualization platform. We are interested in investigating: How to overcome the problem of over-provisioning and under-provisioning? How to guaranty the resource availability? How to overcome the problems of heterogeneity and resource granularity? How to standardize, unify elasticity solutions and model its diversity at a high level of abstraction to manage its different aspects?

In this thesis, we solved such challenges and we investigated all the aspects of elasticity to manage efficiently the resources provisioning and de-provisioning in cloud computing.

It extended the state-of-the-art by making the following three contributions. Firstly, an up-to-date state-of-the-art of the cloud elasticity which reviews different works related to elasticity for both Virtual Machines (VMs) and containers. Secondly, ElasticDocker, an approach to manage container elasticity including vertical elasticity, live migration, and elasticity combination between different virtualization techniques. Thirdly, Model-Driven Elasticity Management with OCCI (MoDEMO), a new unified, standard-based, model-driven, highly extensible, highly reconfigurable elasticity management framework that supports multiple elasticity policies, both vertical and horizontal elasticities, different virtualization techniques and multiple cloud providers.

Declaration

This is to certify that the thesis comprises only my original work towards the PhD. A reference has been made in the text to all other material used.

Yahya Al-Dhuraibi
Monday 10th December, 2018

Acknowledgements

First, I would like to thank Prof. Nouredine MELAB, Dr. Fabienne BOYER, Dr. Thomas LEDOUX, Prof. Sara BOUCHENAK, Prof. Walid GAALOUL, Dr. Nabil DJARALLAH and Dr. Faiez ZALILA for offering their valuable time and serving as my committee members, and for their encouragement and insightful comments.

I would like also to express my sincere gratitude to my supervisor, Dr. Philippe Merle, for guiding me tirelessly, providing me valuable research insights and continuous support throughout my PhD journey. I would also like to thank Dr. Nabil Djarallah, my supervisor in the company, for his guidance and support. He facilitated the access to all the company means to be under my provision. I am also thankful to post-docs Dr. Faiez Zalila and Dr. Fawaz Paraiso for their insightful discussions and guidance.

I would also like to thank all the members of Spirals project team, Inria. Special thanks to Prof. Lionel Seinturier, head of the team for his support. I am fortunate to collaborate with Prof. Romain Rouvoy and Dr. Bo Zhang, many thanks to them. I would not forget to thank the whole staff of Scalair company who were perfect colleagues.

Finally, countless thanks to my family and friends for being always with me and for their support.

Table of contents

List of figures xiii

List of tables xv

1 Introduction 1

1.1 Scope and Challenges 3

1.2 Research Problems and Objectives 5

1.3 Evaluation Methodology 7

1.4 Contributions 8

1.5 Dissertation Outline 9

1.6 Publications 12

I State of the Art 15

2 Elasticity in Cloud Computing: State of the Art and Research Challenges 17

2.1 Introduction 17

2.2 Elasticity 19

2.2.1 Elasticity definition and its related terms 19

2.2.2 Elasticity taxonomy 22

2.2.3 Elasticity performance evaluation 33

2.3 Containerization 35

2.3.1 Pros and Cons 36

2.3.2 Container technologies 37

2.3.3 Container orchestration and management tools 39

2.3.4 Elasticity of containers 40

2.4 Open Issues and Research Challenges 42

2.5 Related Work . . . 47
2.6 Conclusion . . . 48

II Elastic Docker 49

3 ElasticDocker Principles 51
3.1 Introduction . . . 52
3.2 Motivation . . . 53
3.2.1 Resource over-provisioning and de-provisioning . . . 53
3.2.2 Vertical elasticity . . . 54
3.2.3 Containers vs VMs . . . 55
3.2.4 Elasticity coordination . . . 55
3.3 Background . . . 56
3.3.1 Docker technology . . . 56
3.3.2 Checkpoint/Restore In Userspace (CRIU) . . . 59
3.4 ElasticDocker Approach . . . 60
3.4.1 System design . . . 60
3.4.2 Monitoring system . . . 60
3.4.3 Elasticity controller . . . 61
3.5 Coordinated Controller . . . 63
3.5.1 General Design . . . 63
3.5.2 Components of the System . . . 64
3.6 Container live migration . . . 66
3.7 Discussions . . . 67
3.8 Related Work . . . 68
3.9 Conclusion . . . 69

4 ElasticDocker Evaluation 71
4.1 Introduction . . . 71
4.2 ElasticDocker Experiments and Evaluation . . . 72
4.2.1 Experimental setup . . . 72
4.2.2 Evaluation and results . . . 73
4.3 Coordinated controller validation . . . 76
4.3.1 Experimental Setup . . . 76
4.3.2 Research Questions . . . 77
4.3.3 Evaluation Results . . . 77

4.4 Docker live migration efficiency 82
4.5 Conclusion 83

III Model-Driven Elasticity Management with OCCI (MoDEMO) 85

5 MoDEMO Principles 87
5.1 Introduction 88
5.2 Motivation 90
5.2.1 Motivation for MoDEMO 90
5.2.2 Motivation for Docker Model 91
5.3 Background 93
5.4 Docker Model 95
5.4.1 Architecture overview 95
5.4.2 Modeling Docker Containers 96
5.5 MoDEMO Approach 98
5.5.1 MoDEMO Model 98
5.5.2 MoDEMO elasticity policies 105
5.5.3 MoDEMO design tool 110
5.6 Discussion 111
5.7 Related Work 112
5.8 Conclusion 113

6 MoDEMO Evaluation 115
6.1 Introduction 115
6.2 High Level Architecture Overview 116
6.3 Validation 117
6.3.1 MoDEMO policies 118
6.3.2 Runtime settings 119
6.3.3 MoDEMO overhead 121
6.3.4 Docker model evaluation 123
6.4 Conclusion 125

IV Conclusion and Final Remarks 127

7 Conclusions and Future Directions 129

7.1 Summary and Conclusions . . . 129
7.2 Future Directions . . . 131
7.2.1 Proactive approach . . . 131
7.2.2 Application-based elasticity . . . 132
7.2.3 Dynamic thresholds . . . 132
7.2.4 Elasticity coordination . . . 132

References **133**

List of figures

1.1 Resource over-/under-provisioning 4
1.2 Thesis organization 11

2.1 Horizontal vs vertical elasticity 20
2.2 Classification of the elasticity mechanisms 23
2.3 Performance evaluation tools 33
2.4 Container-based virtualization vs. traditional virtualization 36
2.5 Docker and rocket Technologies 38
2.6 Container orchestration engines 40

3.1 High-level overview of Docker architecture. 56
3.2 ElasticDocker architecture 61
3.3 Coordinated elastic controllers between VMs and containers 63
3.4 Migration procedure based on CRIU 66

4.1 Graylog components 73
4.2 Graylog response time with Docker vs ElasticDocker 74
4.3 CPU and memory consumption of Graylog, Elasticsearch and MongoDB containers 75
4.4 Resource utilization 76
4.5 Architecture of Experiment 1 78
4.6 Architecture of Experiment 3 80
4.7 Workloads execution time 80
4.8 workloads total execution time 81

5.1 OCCIware metamodel 93
5.2 Architecture overview. 95
5.3 Docker model. 96
5.4 MoDEMO model 99

5.5 Cloud providers, allocation policies and load balancing algorithms 100
5.6 Compute instances 103
5.7 Scaling and scheduling policies 105
5.8 Migration 105
5.9 MoDEMO design configuration 110

6.1 Architecture overview 117
6.2 MoDEMO runtime configuration 120

List of tables

2.1 Classification of elasticity solutions 24

3.1 ElasticDocker parameters 62
3.2 System control parameters 64

4.1 vCPUs vs. time for Docker container 1 76
4.2 Elastic actions execution performance 79
4.3 Migration performance indicators 82

6.1 Elasticity policies supported by public cloud providers and MoDEMO . . 119
6.2 Configurable parameters at runtime 121
6.3 Vertical elasticity overhead 122
6.4 Horizontal elasticity overhead 123
6.5 Container creating time and overhead. 124
6.6 Container starting time and overhead 124
6.7 Container stopping time and overhead 124

Chapter 1

Introduction

Since 2006, IT witnesses the revolution of cloud computing with the introduction of AWS EC2. According to the National Institute of Standards and Technology (NIST), cloud computing is defined as “a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction” [1]. Cloud computing has attracted attention from the industry and academic worlds because it permits the possibility of acquiring resources in a dynamic and elastic way. In fact, elasticity is a fundamental property in the cloud computing, and perhaps what distinguishes this computing paradigm from the other paradigms, such as grid computing. Elasticity is defined as the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible [2]. According to the NIST definition, cloud computing is classified into three categories:

- **Infrastructure as a Service (IaaS)**: This model allows to lease resources from the cloud such as computing, network, storage resources.
- **Platform as a Service (PaaS)**: This model allows platform providers to deliver a computing platform to application developers such as AWS Elastic Beanstalk, Windows Azure, Heroku, Google App Engine, etc.
- **Software as a Service (SaaS)**: SaaS model aims to provide the applications installed on Cloud to the end-users such as Google Apps, Dropbox, Microsoft Office 365, etc.

This thesis mainly focuses on elasticity aspects related to IaaS. However, it also handles some aspects of elasticity related to platforms and applications.

Most clouds are built on virtualized infrastructure technologies, and since the cloud determines how the virtualized resources are allocated, delivered, and presented, elasticity is a fundamental feature in cloud computing but it also needs to take into consideration the underling enabling virtualization technologies and the main units such as Virtual Machines (VMs) and container computing units. Elasticity is a wide concept and it is usually confused with other terms such as scalability and efficiency. However, compared to the scalability, elasticity is more wider as it covers the spontaneity, efficacity, and velocity. Cloud computing is become an excellent target for business investment but due to the heated marketplace competition in this domain, providers have been under pressure to produce attractive services that satisfy customers by maintaining applications performance and respecting the Service Level Agreement (SLA) with optimal costs. Therefore, elasticity is always a vital component in cloud computing.

However, even though the elasticity has many advantages and has attracted lots of attention from research and industrial communities, it remains a challenging problem. Elasticity has different mechanisms, modes, policies, architectures, scopes, configurations, etc. In addition, there are many trade-offs, which need consideration such as SLA, cost, user Quality of Experience (QoE) and provider profit, etc.

In this thesis, we have extensively explored the concept of elasticity from diverse perspectives. We have proposed new models that manage elasticity aspects using single/multi-strategy with single/multi-technology in the context of mono/multi-cloud. This thesis has been carried out in the Spirals [1] joint project-team between Inria and the University of Lille, and Scalair[2] company. Scalair is a cloud provider company located in France. IT was run in the context of OCCIware[3] research and development project. OCCIware project is funded by French Programme d'Investissements d'Avenir (PIA). This project aims at building a comprehensive, yet modular software engineering toolchain dedicated to service-oriented applications to offer a unified interface for the different clouds. The contributions reported in this thesis have been integrated on the top of this project to enable elasticity for different cloud extensions.

[1] https://team.inria.fr/spirals/
[2] https://www.scalair.fr
[3] www.occiware.org

1.1 Scope and Challenges

Elasticity is one of the main characteristics of cloud computing as it permits the system to automatically adjust resources to the workloads. Elasticity has an ambiguity with scalability an efficiency concepts. Scalability is the ability of the system to sustain increasing workloads by making use of additional resources while elasticity is related to how well the actual resource demands are matched by the provisioned resources at any point in time. Scalability is prerequisite for elasticity, but it does not consider temporal aspects of how fast, how often, and at what granularity scaling actions can be performed as elasticity does. The elasticity in cloud computing can be applied on all cloud service models: IaaS, PaaS, and SaaS. It can be implemented manually or automatically. The automatic method can be further divided into reactive and proactive strategies. Reactive strategies mean the elasticity actions are triggered based on certain thresholds or rules, the system reacts to the load (workload or resource utilization) and triggers actions to adapt changes accordingly. On the other hand, a predictive strategy tries to predict in advance the changes in the system workload and then reconfigures the system accordingly. Many elasticity strategies have been proposed in the literature. However, these strategies are specific to a given system, there is not common strategy that can be applicable to all systems.

There are three main methods of resource management: replication (or horizontal scalability), resizing (or vertical scalability) and migration. Replication involves duplicating a resource in multiple copies. A load balancer is then placed in front of the different replicas to distribute the load among them. Resizing is the process of changing the characteristics of resources (CPU, memory, storage, etc.) for an instance at runtime. Migration involves moving a resource (e.g., migration of a virtual machine) to a more adequate physical location. These last two methods are difficult to implement at runtime in some cases. The state of the art identifies some challenges and open issues on elasticity in cloud computing such as availability of resources, interoperability between clouds, resources granularity, provisioning and de-provisioning time. We discuss some challenges here, while we give a thorough description for many challenges in the state of art in Chapter 2.

Over-provisioning and under-provisioning

Over-provisioning means surplus resources are allocated compared to the demand. Under-provisioning occurs when workload is assigned with fewer numbers/less amount of resources than the demand. As shown in Figure 1.1, over-provisioning guarantees

performance but it lead to the cost increase (e.g., physical machines, energy consumption) for both cloud providers and customers. While under-provisioning reduces costs, it can leads to performance degradation and therefore SLO violation. Maintaining sufficient resources in a real time is one of the challenges that will be addressed by elasticity in this thesis.

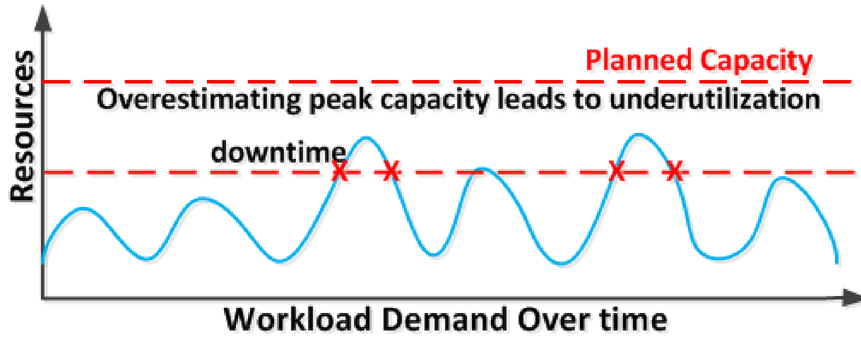


Figure 1.1 Resource over-/under-provisioning

Multi-cloud heterogeneity and interoperability

With a single cloud provider, when an outage takes in all data centers or when there is no enough resources (resource availability challenge), resource provisioning is not possible. Cloud elasticity is essential to accommodate the scale (up/out and down/in) on demand. A solution is to automate the cloud elasticity across different cloud providers. However, managing elasticity across multiple cloud providers is challenging task, according to which criteria the cloud provider is chosen. In addition, each cloud provider has his own Application Programming Interface (API) to manage elasticity, the aim is to provide an abstraction support to hide such complexity and provision resources transparently.

Heterogeneity of the virtualization technologies

Cloud computing relies on virtualization technologies to merge or split computing resources to give one or more execution environments. VMs are the traditional computing units. Nowadays, containers, e.g., Docker, has appeared as a system-level virtualization. We need to manage elasticity across the different virtualization technologies. Are the different elasticity strategies and policies can be applied on the different technologies?

Diversity of mechanisms and strategies

Elasticity has different types, modes, strategies, policies, etc, and different purposes such as increasing performance, reducing cost or power consumption, etc. Having a system that supports multiple policies and strategies, and manages the different trade-offs is very

challenging. Therefore, a unified elasticity framework is needed to handle and manage the different aspects of elasticity.

Monitoring
Monitoring is an essential component of any elasticity system. To guide the elasticity system management, many factors have to be taken into consideration such as metrics. A well planified elastic system will use low performance metrics (e.g., CPU utilization), high level performance metrics stipulated as SLA or SLO (e.g., response time) or a mix of both. Monitoring frequency and interval have also a great impact on the elasticity system. Different monitoring mechanisms may be used according to the virtualization technology (e.g., VMs or Docker containers).

Provisioning delay (startup time)
When provisioning a resource, it takes a time for the acquired resources to be ready, this time period is called startup time. For example, provision a VM with horizontal elasticity could take several minutes, this time will have great impact on the performance of the system. High startup time could break the great advantage of the elasticity that is the ability to dynamically provision resources in response to application demand.

In addition, there are other challenges such as resource granularity, and the runtime settings of the elasticity controller, etc. In our contributions, we have tried to handle such challenges and contradictions. Our state of art highlights in details these and other challenges.

1.2 Research Problems and Objectives

The aim of this thesis is to explore cloud elasticity and investigate its different mechanisms and policies in heterogeneous environments. It can be summarized by the following research question:

How to efficiently manage the different aspects of elasticity across single or multiple cloud providers by taking into consideration different challenges such as technologies of virtualization, diversity of mechanisms, startup time, over-provisioning and under-provisioning, etc.

In order to ease the understanding of the research questions, we formalize this section into three main parts.

Part 1: Cloud Elasticity State of the Art
To handle elasticity, we need to explore this term, therefore several questions arise:

- What is the best definition of elasticity?
- How to measure elasticity systems?
- What are the different elasticity mechanisms and approaches used in the literature?
- What are the major challenges and open issues related to the elasticity?

Part 2: Containers Elasticity
Secondly, since VMs and containers are the main computing units driving cloud providers, we start investigating elasticity around containers. This leads to the following main research question: How to mange elasticity in containers?

Based on this question, we need to address the following sub-questions and challenges:

- How containers can dynamically and timely adapt resources according to the application workload?
- How to avoid rapid scaling oscillation?
- How to reduce system instability due to the scaling actions?
- How to migrate containers lively?
- How to coordinate vertical elasticity of both VMs and containers?
- What is the impact of container elasticity on performance, cost and resource consumption?
- What is the impact of coordinating elasticity of both VMs and containers?
- How containers can automatically detect the hot added resources (on the fly)?
- How to efficiently monitor elasticity compute units (VMs and containers)?

Part 3: Model Driven Elasticity
Thirdly, as we have started investigating elasticity around containers, we then moved towards a unified modular approach to manage elasticity for both VMs and containers. This leads to the following question.

How to model the elasticity concept and all its related aspects at high level of abstraction? Based on this question, we need to address the following sub-questions and challenges:

- How to model the different components of elasticity?

- Is it possible to use Model-Driven Engineering (MDE) in elasticity abstraction without introducing overhead?
- How to modify elasticity configurations and parameters at runtime?
- How to model Docker containers and expose features that allows elasticity and monitoring aspects?
- How to modify load balancer algorithms and parameters at runtime?
- How to overcome the vendor lock-in problem and heterogeneity across cloud providers offerings, and seamlessly provision resources on multiple clouds?
- How to select a target cloud provider in the context of multi-cloud elasticity?
- Is it possible to unify and improve all the elasticity policies provided by the worldwide cloud providers such as Amazon Web Services (AWS), Microsoft Azure, Google Cloud Platform (GCP) in a single modular framework?

Objective
The main objective of this thesis is to propose a flexible framework for elasticity in cloud computing. This framework should cover different kinds of resources such as IaaS and PaaS, support multi-clouds and variety of virtualization techniques, by using a composition of elasticity strategies and policies.

1.3 Evaluation Methodology

Throught this thesis, we have developed several approaches to manage the different aspects of elasticity. These approaches were evaluated using a variety of experimental platforms and different hardware specifications. All of our experiments are done in the production infrastructure of Scalair data centers. Many experiments are executed on different platforms to illustrate the efficacy and feasibility of our proposed system in Chapter 4 and 6. Many scripts are developed to repeat the experimentation and to set some special application settings. We list here the different experimental platforms, and hardware specifications used during the evaluation of our approaches.

- **Hardware specifications**: 2 HP ProLiant DL380 G7, 2 HP Proliant XL170r Gen9.
- **Cloud orchestrators**: vSphere vCenter, OVirt, OpenStack

- **Hypervisors**: VMware, KVM
- **Docker technologies**: Docker engine, Docker-compose, Kubernetes
- **Applications**: Graylog, RUBiS, httpd, nginx
- **Monitoring**: Zabbix, Docker stats, cAdvisor
- **Workload generators**: httperf, ab
- **OS**: Centos, Ubuntu, MacOS

1.4 Contributions

As presented in Section 1.2, the goal of this thesis is to provide a flexible framework that manages the cloud elasticity. In particular, we are interested in the challenges related to the elasticity concept, challenges related to the diversity of mechanisms and policies, heterogeneity of clouds, and technological challenges. Thus, the contributions of this thesis are aimed to solve the challenges and research questions identified in 1.2. These contributions are aimed at supporting the management of every aspect of elasticity. Here is a summary of the main contributions:

1. A complete up-to-date state-of-the-art of elasticity in cloud computing.
    - Revision of the elasticity definitions, proposing a new precise definition and highlighting the related concepts to elasticity such as scalability and efficiency.
    - Revision of the elasticity measurement approaches.
    - Illustration of the elasticity related terms such as elasticity mechanism, policy, mode, scope, configuration, etc.
    - A thorough classification and taxonomy of elasticity solutions applied for both VMs and containers.
    - Discussion of the existing container technologies and their relation to the elasticity.
    - Identification of some open issues and research challenges.
2. An approach to manage vertical elasticity of containers, coordinate the vertical elasticity of both VMs and containers and effectuate live migration of containers.

- Definition of the architecture and its components.
- A prototype implementation of the proposed approach.
- Comprehensive evaluation of the approach with respect to performance, cost, resource optimization, vertical elasticity of containers only, vertical elasticity of VMs only.

3. A unified model-driven approach to handle many aspects of elasticity.
    - Definition of the approach and its components:
        - This approach supports both vertical and horizontal elasticities, both VM and container, multiple cloud providers, and various elasticity policies.
        - This approach is highly extensible, highly reconfigurable with a negligible overhead.
        - This approach supports transparent configuration, it hides complexity and manages the resources transparently such load balancer configuration and algorithms, monitoring system, cloud deployment infrastructure, etc.
    - Definition of a model to manage Docker containers, this model is then used as a one of the infrastructure compute management extensions in our approach for elasticity management.
    - An implementation of the proposed approach.
    - Comprehensive evaluation of the approach with respect to other approaches provided by the worldwide cloud providers, with respect also to the overhead and runtime settings .

1.5 Dissertation Outline

As shown in Figure 1.2, the remainder of this dissertation is composed of 6 chapters that are organized into four parts as follows:

Part I: State of the Art

Chapter 2: Elasticity in Cloud Computing: State of the Art and Research Challenges. This chapter introduces the cloud elasticity and its related terms and concepts. It provides a survey of the most relevant works on elasticity for both VMs and containers.

We have classified the elasticity strategies based on the existing academic and commercial solutions. The proposed taxonomy covers many features and aspects of elasticity based on the analysis of diverse proposals. This chapter also discusses the open issues and research challenges related to the elasticity. This chapter is a based on the following article:

Article 1: *Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in Cloud Computing: State of the Art and Research Challenges.* IEEE Transactions on Services Computing*, 11(2):430–447, March 2018. ISSN 1939-1374. doi: 10.1109/TSC.2017.2711009*

Part II: Elastic Docker

Chapter 3: ElasticDocker Principles. This chapter presents ElasticDocker, the first system powering vertical elasticity of Docker containers autonomously. This approach scales up and down both CPU and memory assigned to each container according to the application workload. A complementary, yet a new controller is proposed to coordinate the vertical elasticity of both containers and VMs. As vertical elasticity is limited to the host machine capacity, ElasticDocker does container live migration when there is no enough resources on the hosting machine. This chapter is a revised version of the following papers:

Paper 2: Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic Vertical Elasticity of Docker Containers with ElasticDocker. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 472–479, June 2017. doi: 10.1109/CLOUD.2017.67

Paper 3: Yahya Al-Dhuraibi, Faiez Zalila, Nabil Djarallah, and Philippe Merle. Coordinating Vertical Elasticity of both Containers and Virtual Machines. In *International Conference on Cloud Computing and Services Science - CLOSER 2018*, Mars 2018

Chapter 4: ElasticDocker Evaluation. In this chapter, we evaluate the approach proposed in Chapter 3 with respect to performance, cost and resource optimization. The coordinating controller is then evaluated with respect to four research questions. This chapter is based on **Paper 2** and **Paper 3**.

Part III: Model-Driven Elasticity Management with OCCI (MoDEMO)

Chapter 5: MoDEMO Principles. This chapter presents a unified model-driven elasticity management approach. This approach covers almost all the elasticity aspects including

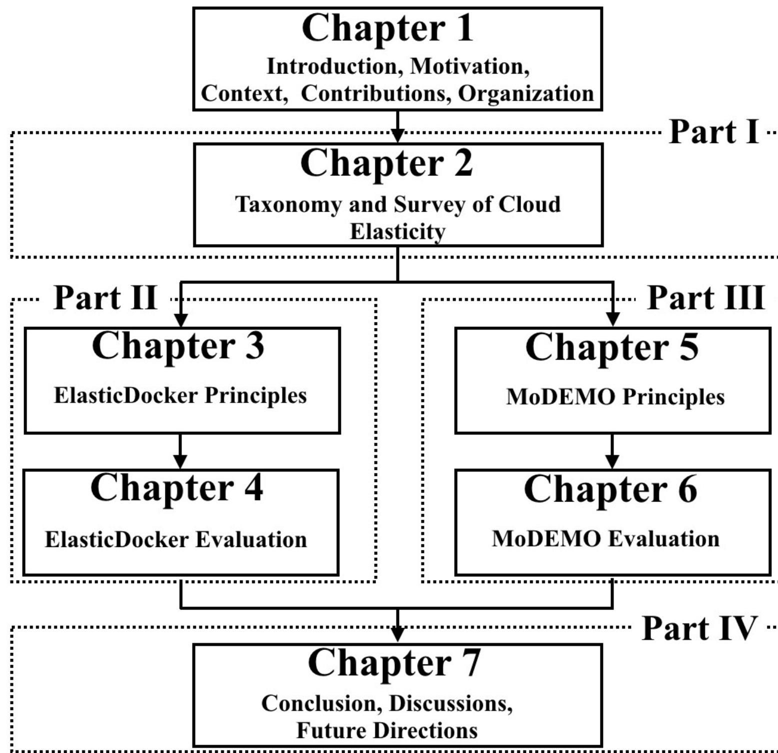


Figure 1.2 Thesis organization

the concepts of multi-clouds, multi-strategies and multi-technologies. All algorithms of elasticity policies are controlled via a model at runtime. It presents the main elements of the proposed approach and how they deal with the issues and challenges identified in Section 1.2. Generally, Chapter 3, Chapter 4, Chapter 5, and Chapter 6 overcome the problems of resource over-provisioning and under-provisioning, heterogeneity, and technological complexities. This chapter is revised version of the following papers:

Paper 4: Yahya Al-Dhuraibi, Faiez Zalila, Nabil Djarallah, and Philippe Merle. Model Driven Elasticity Management with OCCI. *submitted to IEEE Transactions on Cloud Computing, June 15*, 2018. first feedback major revision

Paper 5: Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. Model-Driven Management of Docker Containers. In *9th IEEE International Conference on Cloud Computing (CLOUD)*, pages 718–725, San Francisco, United States, June 2016

Chapter 6: MoDEMO Evaluation. In this chapter, we provide an extensive evaluation for the approach presented in Chapter 5. The proposed solution is compared with the elastic systems provided by AWS, MS Azure and GCP. In addition, the model overhead is evaluated. This chapter is derived from **Paper 4** and **Paper 5**.

Part IV: Conclusion and Final Remarks

Chapter 7: Conclusions and Future Directions

This chapter summarizes this thesis and discusses the contributions, impacts, limitations and potential future perspectives.

1.6 Publications

The results of this work were published in a number of peer-reviewed conference proceedings and journals. The following is a list of publications (conference papers and articles).

- **Yahya Al-Dhuraibi**, Fawaz Paraiso, Nabil Djarallah and Philippe Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," in IEEE Transactions on Services Computing, vol. 11, no. 2, pp. 430-447, March-April 1 2018. doi: 10.1109/TSC.2017.2711009.
- **Yahya Al-Dhuraibi**, Faiez Zalila, Nabil Djarallah and Philippe Merle, "Model-Driven Elasticity Management with OCCI," submitted to IEEE Transactions on Cloud Computing (TCC), June 15 2018, first feedback major revision.
- **Yahya Al-Dhuraibi**, Faiez Zalila, Nabil Djarallah, Philippe Merle, "Coordinating Vertical Elasticity of both Containers and Virtual Machines," 8th International Conference on Cloud Computing and Services Science - CLOSER 2018, Mar 2018, Funchal, Madeira, Portugal.
- **Yahya Al-Dhuraibi**, Fawaz Paraiso, Nabil Djarallah and Philippe Merle, "Autonomic Vertical Elasticity of Docker Containers with ElasticDocker," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, CA, 2017, pp. 472-479. doi: 10.1109/CLOUD.2017.67.
- BO Zhang, **Yahya Al-Dhuraibi**, Romain Rouvoy, Fawaz Paraiso and Lionel Seinturier, "CloudGC: Recycling Idle Virtual Machines in the Cloud," 2017 IEEE International Conference on Cloud Engineering (IC2E), Vancouver, BC, 2017, pp. 105-115. doi: 10.1109/IC2E.2017.26.
- Fawaz Paraiso, Stéphanie Challita, **Yahya Al-Dhuraibi** and Philippe Merle, "Model-Driven Management of Docker Containers," 2016 IEEE 9th International

Conference on Cloud Computing (CLOUD), San Francisco, CA, 2016, pp. 718-725. doi: 10.1109/CLOUD.2016.0100.

Part I

State of the Art

Chapter 2

Elasticity in Cloud Computing: State of the Art and Research Challenges

Elasticity is a fundamental property in cloud computing that has recently witnessed major developments. This chapter reviews both classical and recent elasticity solutions and provides an overview of containerization, a new technological trend in lightweight virtualization. It also discusses major issues and research challenges related to elasticity in cloud computing. We comprehensively review and analyze the proposals developed in this field. We provide a taxonomy of elasticity mechanisms according to the identified works and key properties. Compared to other works in literature, this chapter presents a broader and detailed analysis of elasticity approaches and is considered as the first survey addressing the elasticity of containers.

2.1 Introduction

Cloud computing has been gaining more popularity in the last decade and has received a great deal of attention from both industrial and academic worlds. The main factor motivating the use of cloud platforms is their ability to provide resources

This chapter is derived from: ***Yahya Al-Dhuraibi***, *Fawaz Paraiso, Nabil Djarallah and Philippe Merle, "Elasticity in Cloud Computing: State of the Art and Research Challenges," in IEEE Transactions on Services Computing, vol. 11, no. 2, pp. 430-447, March-April 1 2018. doi: 10.1109/TSC.2017.2711009.*

according to the customer's needs or what is referred to as elastic provisioning and de-provisioning. Therefore, elasticity is one of the key features in cloud computing that dynamically adjusts the amount of allocated resources to meet changes in workload demands [1].

Cloud providers generally use virtualization-based approaches to build their stack. Virtualization makes it possible to run multiple operating systems and multiple applications on the same server at the same time. It creates an abstract layer that hides the complexity of both hardware and software working environments. Cloud computing paradigm allows workloads to be deployed and scaled-out quickly through the rapid provisioning of the virtualized resources. This deployment is done through virtual machines (VMs). Virtualization is commonly implemented with hypervisors. A hypervisor is one of the virtualization techniques that allows multiple operating systems to share a single hardware host in a way that each operating system appears to have its own independent resources. VMware ESX, KVM, Xen, and Hyper-V are examples of the worldwide used hypervisors.

Container-based virtualization, called operating system virtualization, is another approach to virtualization in which the virtualization layer runs as an application within the operating system (OS). Containers are a lightweight solution that allows faster start-up time and less overhead [8]. Therefore, since virtualization is a central part of cloud computing that helps to improve elasticity, we discuss cloud elasticity in the context of both VMs and containers. In the literature, there exist various definitions, mechanisms, strategies, methods, and solutions for elasticity in both industrial and research works.

Elasticity has been explored by researchers from academia and industry fields. Many virtualization technologies, on which cloud relies, continue to evolve. Thus, tremendous efforts have been invested to enable cloud systems to behave in an elastic manner and many works continue to appear. Therefore, we are motivated to provide a comprehensive and extended classification for elasticity in cloud computing. This chapter focuses on most aspects of the elasticity and it particularly aims to shed light on the emerging container elasticity as well as the traditional VMs. Although many elasticity mechanisms have been proposed in the literature, our work addressing more broader classification of elasticity taxonomy. It is also the first survey that highlights elasticity of containers. The major contributions of this chapter are summarized as:

- First, we propose a precise definition of elasticity and we highlight related concepts to elasticity such as scalability and efficiency and approaches to measure elastic systems.

- Second, we provide an extended classification for the elasticity mechanisms according to the configuration, the scope of the solution, purpose, mode, method, etc. For example, when discussing the mode of elasticity that can be reactive or proactive to perform elasticity decisions, we discuss in depth each mode by classifying the mode into other subcategories and presenting works that follow the mode as shown in Table 2.1.
- Third, we discuss the existing container technologies and their relation to cloud elasticity. This chapter is the first work that discusses container elasticity in presenting many recent works from the literature.

The remainder of the chapter is organized as follows. Section 2.2 explains the elasticity concept, its related terms, its classical solution classifications and our new extended classification, the tools, and platforms that have been used in the experiments of the existing works in the literature. This section describes cloud elasticity solutions in the VMs. Next, Section 2.3 presents the concept of containerization, and how it could improve elasticity in cloud computing. It discusses the few existing papers on cloud elasticity when containers are used. Then, in Section 2.4, we present the main research challenges in elasticity and also the limits in the new trend of containerization. Section 2.5 discusses previous surveys on cloud elasticity. Finally, Section 6.4 concludes the chapter.

2.2 Elasticity

In order to well understand the elasticity, we describe some related concepts, in addition to a new refined and comprehensive definition for elasticity. We propose a classification and taxonomy for elasticity solutions based on the characteristics: configuration, scope, purpose, mode, method, provider, and architecture. This classification is a result of thorough study and analysis of the different industrial and academic elasticity solutions. This classification provides a comprehensive and clear vision on elasticity in cloud computing. We then review the elasticity evaluation tools and platforms implemented in diverse works.

2.2.1 Elasticity definition and its related terms

There have been many definitions in the literature for elasticity [9], [10], [2], [1]. However, from our point of view, we define elasticity as the ability of a system to add and remove

resources (such as CPU cores, memory, VM and container instances) “on the fly" to adapt to the load variation in real time. Elasticity is a dynamic property for cloud computing. There are two types of elasticity as shown in Figure 2.1: horizontal and vertical. Horizontal elasticity consists in adding or removing instances of computing resources associated with an application. Vertical elasticity consists in increasing or decreasing characteristics of computing resources, such as CPU time, cores, memory, and network bandwidth.

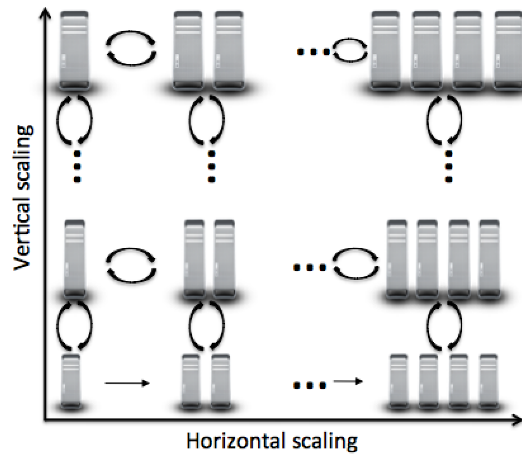


Figure 2.1 Horizontal vs vertical elasticity

There are other terms such as scalability and efficiency, which are associated with elasticity but their meaning is different from elasticity while they are used interchangeably in some cases. Scalability is the ability of the system to sustain increasing workloads by making use of additional resources [2], it is time independent and it is similar to the provisioning state in elasticity but the time has no effect on the system (static property). In order to have a complete understanding, we deduce the following equation that summarizes the elasticity concept in cloud computing.

$$\boxed{Elasticity = \underbrace{scalability + automation}_{\text{auto-scaling}} + optimization}$$

It means that the elasticity is built on top of scalability. It can be considered as an automation of the concept of scalability, however, it aims to optimize at best and as quickly as possible the resources at a given time. Another term associated with elasticity is the efficiency, which characterizes how cloud resource can be efficiently utilized as it scales up or down. It is the amount of resources consumed for processing a given amount of work, the lower this amount is, the higher the efficiency of a system. The amount

of resources can relate to cost, power consumption, etc., depending on the targeted resource [11]. Generally, this is a measure of how well the application is using the provided resources. Higher cloud elastic system results in higher efficiency. The processes of resource provisioning and scheduling (i.e., jobs or customer' requests on instances) are both related to elasticity since they try to provision instances but in response to provider and customer tradeoffs. [12], [13] provision resources according to a utility model to satisfy customers' needs and a certain pricing model to increase service provider profit. The provisioning and scheduling processes may take a certain delay in order to meet SLAs and provider profit conditions.

It is worth noting that scaling up or down the resources can lead to a deviation of the current amount of allocated resources from the actual required resource demand. The accuracy of elasticity systems varies from one system to another. Over-provisioning and under-provisioning are two important factors that characterize an elastic system. The system enters in over-provisioning state once the resources provided (called supply S) are greater than the consumer required resources (called demand D), i.e., $S > D$. Though QoS can be achieved, over-provisioning state leads to extra and unnecessary cost to rent the cloud resources. Under-provisiong takes place once the provided resources are smaller than the required resources, i.e., $S < D$, and this causes performance degradation and violation of service level agreement (SLA). There is no common methodology to measure or determine temporal or quantitative metrics for elasticity. A consumer can measure the delay it takes to provision and deprovision resources, in addition to the sum of delays of over-provisioning and under-provisioning states to quantify different elastic systems [14].

[15] discusses methods to measure scalability and elasticity of a system. According to [15], effects of scalability are visible to the user via observable response times or throughput values at a certain system size. On the other hand, the elasticity, namely the resource resizing actions, may be invisible to the user due to their shortness or due to the transparency and dynamicity of resource provisioning. The effect of reconfiguration on performance metrics (e.g., response time) due to elastic adjustments of resources and the reaction time can quantify the elasticity. It is clear that elasticity is controlled with time. Therefore, the speed is also very important in elasticity. Reaction time is the time interval between the instant when a reconfiguration has been triggered/requested and until the adaptation has been completed.

[16] proposes an approach for elasticity measurements. In addition to the over-provisioning and under-provisioning states, another state called just-in-need is introduced. Just-in-need denotes a balanced state, in which the workload can be properly handled

and quality of service (QoS) can be satisfactorily guaranteed. The approach developed calculation formulas for measuring elasticity values based on the time intervals a system stays in one state. There are three states: over-provisioning, under-provisioning, and just-in-need. A set of rules is used to determine the state of a system based on the workload and computing resources. The equations can be obtained and calculated by directly monitoring the system or by using continuous-time Markov chain (CTMC) model. The drawback of the proposed system is that it assumes the system is in a certain state based on rules. For example, the system is in just-in-need state if the number of requests (j) is greater than the number of VMs (i) and less than 3 multiplied by the number of VMs (i), i.e., ($i < j \leqslant 3i$). We cannot guarantee the certainty for these rules on all elastic systems.

2.2.2 Elasticity taxonomy

Elasticity solutions build their mechanisms on different strategies, methods, and techniques. Therefore, different classifications [9], [10], [17], [18], [19] have been proposed according to the characteristics implemented in the solutions. We have investigated many industrial and academic solutions, in addition to papers in the elasticity literature, and then we propose the classification shown in Figure 2.2. It is an extended and complementary elasticity classification as compared to classification in [9], [10], [17], [18], and [19].

Next subsections explain in details each characteristic and mechanism used. The solutions are classified according to the chosen *configuration*, *scope*, *purpose*, *mode or policy*, *method or action*, *architecture*, and *provider*.

2.2.2.1 Configuration

Generally, configuration represents a specific allocation of CPU, memory, network bandwidth and storage [20]. In the context of our classification (see Figure 2.2), configuration represents the method of the first or initial reservation of resources with a cloud provider. During the first acquisition of resources, the consumer either chooses from a list of offer packs or specifies its needs, i.e., combining different resources. Therefore, the configuration can be either rigid (fixed) or configurable. The rigid mode means that the resources are offered or provisioned in a constant capacity. The virtual machine instances (VMIs) are found with a predefined resource limit (CPU, Mem, etc.) called instances

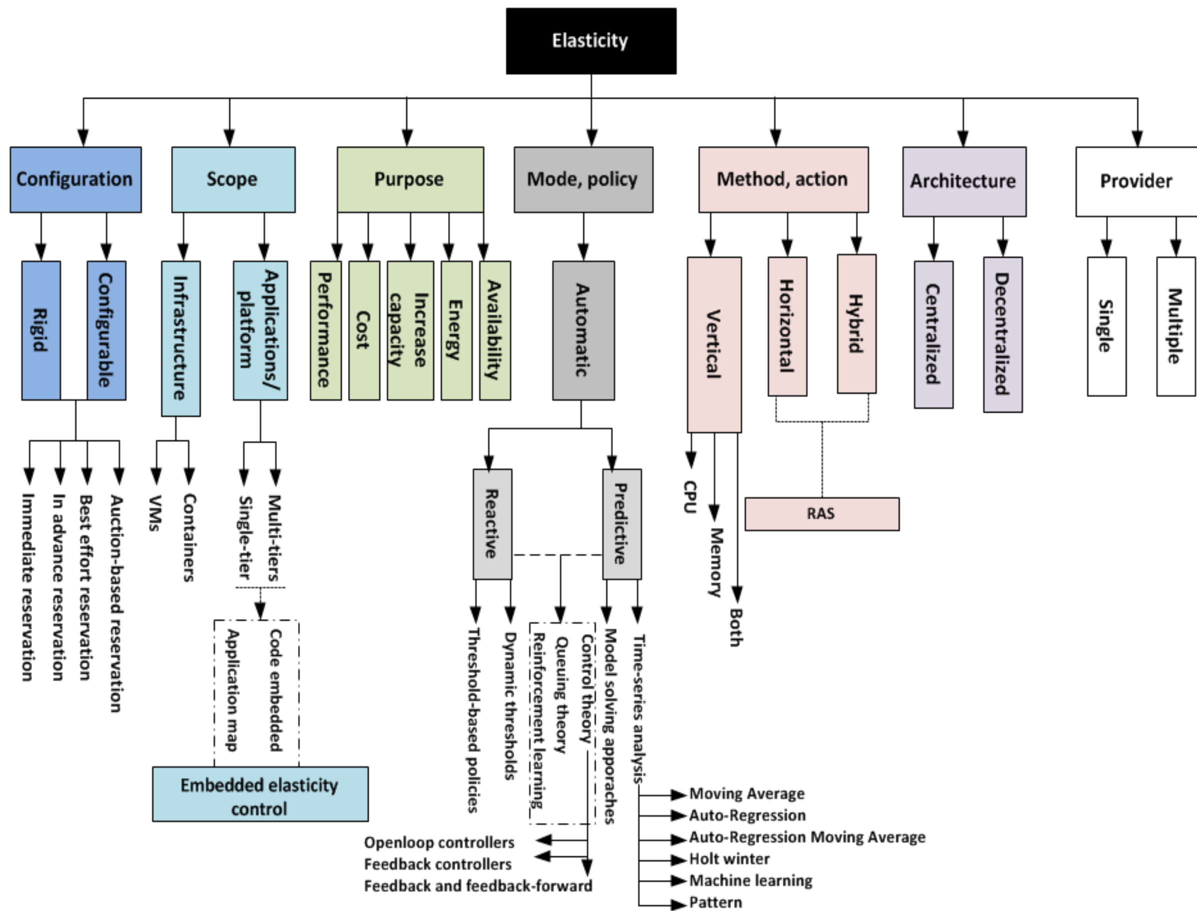


Figure 2.2 Classification of the elasticity mechanisms

such as Amazon EC2 (offering 38 instances), Microsoft Azure (offering many series A, D, DS, G, and GS and each series has different VM sizes). In the cloud market, the VMIs are offered in various configurations. The problem with rigid configuration is that the resource rarely meets the demand, therefore, there is always under-provisioning or over-provisioning. The configurable mode allows the client to choose the resource such as number of CPU cores in the VMs. ProfitBricks [21] and CloudSigma [22] are examples of this type.

The customers can reserve the resources according to the following reservation methods [20]:

- **On-demand reservation**: The resources are reserved immediately or the requests will be rejected if there are no enough available resources.

- **In advance reservation**: The clients send initial requests to reserve resources and a fixed price charge is required to initiate the reservation, the resources must be available at a specific time.
- **Best effort reservation**: Reservation requests are queued and served accordingly such as Haizea, an open-source VM-based lease management architecture used in OpenNebula [23].
- **Auction-based reservation**: Specific resource configurations are reserved dynamically as soon as their prices are less than bid amount offered by the customer [24].
- There are other types of reservation such as Amazon's scheduled reserved instances, Amazon's dedicated instances, Google's preemptible instances, etc.

Table 2.1 Classification of elasticity solutions

Elasticity	**Configuration**	**Rigid**				[25], [26]
		Configurable				[21], [22]
	Scope	**Infrastructure**	**VMs**			[27], [26], [28], [29], [30], [31], [32], [33], [34], [35]
			Containers			[36], [37], [38], [39], [7], [40], [41]
		Application/ Platform	**Single-tier**			[42], [43], [44], [45], [46], [47], [48], [49], [50]
			Multi-tier			[51], [52], [53], [54], [55], [56], [57], [27], [58]
			Application map			[59], [43], [44], [45]
			Code embedded			[17], [60]
	Purpose	**Performance**				[51], [52], [55], [56], [27], [26], [28], [29], [30], [31], [32], [33], [35], [42], [44], [45], [61], [62], [38], [47], [49], [58], [63], [64], [65], [66], [67], [68], [69], [70], [71], [72], [73], [74], [75], [76], [77], [78], [79]
		Cost				[51], [52], [34], [80], [81], [82], [83], [61], [36], [84], [62], [85], [86], [87]
		Capacity				[29], [46]
		Energy				[35], [43], [82], [88], [89]
		Availability				[53], [57], [82], [90], [91]
	Mode	**Automatic**	**Reactive**	**Threshold-based policies**		[53], [56], [57], [29], [83], [61], [84], [91], [38], [65], [67], [70], [71], [78], [92], [93], [94], [95], [96], [97]
				Dynamic thresholds		[62], [88], [98], [99]
				Reinforcement learning		[48], [93], [100]
				Queuing theory		[61], [84], [85], [101]
				Control theory		[76], [98], [99], [102], [103]
			Proactive	**Time series analysis**	**Moving average**	[56], [104], [105]
					Auto regression	[52], [53], [54], [106], [107]
					ARMA	[52], [32], [38]
					Holt winter	[83], [105]
					Machine learning	[27], [80], [66], [69], [70], [71], [74], [77], [108], [109]
					Pattern	[106], [110]
				Model solving approaches		[58], [79], [96]
				Reinforcement learning		[111], [112]
				Queuing theory		[101], [112], [113], [114], [115]
				Control theory		[51], [55], [78], [103], [116]
	Method	**Horizontal scaling**				[52], [53], [57], [27], [26], [29], [30], [31], [32], [33], [46], [61], [91], [47], [66], [68], [96], [117], [118], [119], [120], [121], [122], [41]
		Vertical scaling	**CPU**			[55], [35], [114], [123], [124]
			Memory			[50], [125], [126]
			CPU & Mem.			[28], [83], [127], [128], [129], [130], [131]
		Migration				[35], [44], [89], [49], [73], [77], [132], [133], [134], [135], [136], [137], [138], [139]
		Hybrid				[34], [35], [43], [80], [62], [77], [78], [140], [97]
	Architecture	**Centralized**				Most approaches presented in this table, except the decentralized ones.
		Decentralized				[86], [87], [141], [142], [143], [144]
	Provider	**Single**				[51], [52], [53], [54], [56], [83], [61], [84], [65], [66], [69], [74], [98], [145]
		Multiple				[57], [82], [90], [91], [37], [86], [140], [143], [97]

2.2.2.2 Scope

The elasticity actions can be applied either at the infrastructure or application/platform level. The elasticity actions perform the decisions made by the elasticity strategy or management system to scale the resources. When the elasticity action control is in the application or platform level, it is named embedded elasticity and this will be described below. Google App Engine [146], Azure elastic pool [25] are examples of elastic Platform as a Service (PaaS). The applications can be either one tier or multi-tiers, most of the existing elasticity solutions are dedicated to one-tier applications where elasticity management is performed for one tier only, mostly the business tier. However, there are some recent works that perform elasticity actions on multi-tier applications such as [51], [52], [53], [54], [55], [56], [57], [27].

Beside this, the elasticity actions can be performed at the infrastructure level where the elasticity controller monitors the system and takes decisions. The cloud infrastructures are based on the virtualization technology, which can be VMs or containers. Most of the elasticity solutions [27], [26], [28], [29], [30], [31], [32], [33], [34], [35] are dedicated to the infrastructure level, and these solutions are suitable for client-server applications. However, other elastic solutions exist for the other types of applications. For example, [60] and Amazon EMR are elastic solutions for MapReduce applications, [42] describes an elasticity solution for streaming applications, while [17] discusses approaches for elasticizing scientific applications. Due to the nature of a scientific application such as parallelism, models (e.g., serial, multithread, single program multiple data, master-worker, etc.), an elasticity solution can not be generalized for scientific applications. The elasticity solution must consider the internal structure and behavior of a scientific application, therefore, to have a reliable elastic solution, it should be embedded in the application source code. It is worth mentioning that some elasticity controllers support sticky sessions. The session is the concept of a series of interactions between a client and the application. The stateful nature of some sessions forces the user to be connected to the same server each time he submits a request within the session, if the session data is stored in the server, such sessions are called sticky sessions. Sticky sessions cause issues on efficiently utilizing elastic resources because they limit the ability of the elastic controller to terminate under-utilized instances when there are still unfinished sessions handled by them. Most solutions support stateless applications, while few solutions [147], [148] handle stateful instances or sticky sessions.

Embedded Elasticity

Most of the existing solutions are dedicated to server-based applications. However,

there are many different application modules that have different execution behavior particularities such as scientific applications. Therefore, we named these types of solutions as embedded elasticity controller. In the embedded elasticity, elastic applications are able to adjust their own resources according to runtime requirements or due to changes in the execution flow. There must be a knowledge of the source code of the applications. As seen in Figure 2.2, we classify these solutions into two subcategories.

- **Application Map**: The elasticity controller must have a complete map of the application components and instances. As it is well known that some applications comprise of many components and each component may have many instances. These components are either static or dynamic. Static components must be launched once the application starts, while dynamic components can be started or stopped during the application runtime. In addition, there are interconnections between these instances. Therefore, the elasticity controller must have all the information about the application instances, components, and interconnections that allow it to perform elasticity actions for applications. [59], [43], [44], [45] are examples of such works.
- **Code embedded**: The idea here is that the elasticity controller is embedded in the application source code. The elasticity actions are performed by the application itself. While moving the elasticity controller to the application source code eliminates the use of monitoring systems, there must be a specialized controller for each application. Examples of these solutions are [17] and [60].

2.2.2.3 Purpose

Elasticity has different purposes such as improving performance, increasing resource capacity, saving energy, reducing cost and ensuring availability. Once we look to the elasticity objectives, there are different perspectives. Cloud IaaS providers try to maximize the profit by minimizing the resources while offering a good Quality of Service (QoS), PaaS providers seek to minimize the cost they pay to the cloud. The customers (end-users) search to increase their Quality of Experience (QoE) and to minimize their payments. QoE is the degree of delight or annoyance of the user of an application or service [149]. The goal of QoE management is then to deliver the cloud application to the end user at high quality, at best while minimizing the costs of the different players of the cloud computing stack (IaaS, PaaS, SaaS) [150]. As consequences, there have been many trade-offs. Elasticity solutions cannot fulfill the elasticity purposes from different perspectives at the same time, each solution normally handles one perspective. However, some solutions

try to find an optimal way to balance some of the contradicted objectives. [13] scales resources according to a utility model to reply to customers QoE and a dedicated pricing model to increase service provider gains. [151] presents a survey of how to look for balancing two opposed goals, i.e., maximizing QoS and minimizing costs. As shown in Table 2.1, most proposals improve the performance. However, there are other works that have described the use of elasticity for purposes, such as, increasing the local resources capacity [29], [46], cost reduction [51], [52], [34], [80], [81], [82], [83], [61], [36], [84], [62], [85] and energy savings [35], [43], [82], [88], [89]. Many of the elasticity management solutions as indicated in [18] takes into consideration Quality of Business metrics that are often expressed in monetary units and include service price, revenue per user, revenue per transaction, provisioning cost, and budget. Examples of the solutions that ensure the availability include [53], [57], [82], [90], [91]. [89] takes into consideration both the provider profit and user QoE. In this work, various algorithms have been studied in order to obtain the best trade-off between the user or SLA requirements and provider profit. [152], [153] also propose QoE-aware management elastic approaches that try to maximize users' satisfaction without extra costs. Other examples for improving the performance are found in the research community and commercial clouds such as Rackspace [30], Scalr [33], RightScale [31].

2.2.2.4 Mode or policy

Mode (policy) refers to the needed interactions (or manner) in order to perform elasticity actions. Elasticity actions are performed by an automatic mode. Scaling actions can be achieved by manual intervention from the user. As indicated in [17], there is also another mode, which is called programmable mode. In fact, it is just the same as manual mode because the elasticity actions are performed using API calls. Though a cloud provider offers an interface which enables the user to interact with the cloud system. The manual policy is used in some cloud systems such as Datapipe [154], Rackspace [30], Microsoft Azure [25], and the Elastin framework [44] where the user is responsible for monitoring the virtual environment and applications, and for performing all scaling actions. This mode can not be considered as an elasticity mode since it violates the concept of automation.

Automatic mode: All the actions are done automatically, and this could be classified into reactive and proactive modes.

1. **Reactive mode** means the elasticity actions are triggered based on certain thresholds or rules, the system reacts to the load (workload or resource utilization) and triggers actions to adapt changes accordingly. Most cloud platforms such as Amazon EC2 [26], Scalr [33], Rightscale [31] and other research works such as [61], [120], [155], [156] use this technique.

   - **Static thresholds** or rule-condition-actions: The elasticity actions are fired to scale up or down the resources when the event-condition rule is met. This policy depends on thresholds or SLA requirements, the conditions are based on the measurements of one or a set of metrics such as CPU utilization, memory utilization, response time, etc. Two or more thresholds are used for each performance metric. The measured metrics are compared against fixed thresholds. For example, if CPU utilization is greater than 80%, and this situation lasts 5 minutes, then the resource is scaled up. Amazon EC2, Rightscale and other research works such as [53], [56], [57], [29], [83], [61], [84], [91], [38], [65], [67], [70], [71], [92], [93], [94], [95] use such mechanism.

   - **Dynamic thresholds**: Previous thresholds are static and are fixed user-defined values. On the contrary, dynamic thresholds, called adaptive thresholds, changed dynamically according to the state of the hosted applications. The works in [62], [88], [98], [99] use the adaptive utilization thresholds technique. The thresholds such as CPU utilization are changed dynamically.

2. **Proactive mode**: This approach implements forecasting techniques, anticipates the future needs and triggers actions based on this anticipation. Many academic works such as [28], [32], [34] use this mode as we will see in the following proactive techniques.

   - **Time series analysis**: Time series is a sequence of measurements taken at fixed or uniform intervals [157]. Time series analysis is used to identify repeating patterns in the input workload and to attempt to forecast the future values. In other terms, time series analysis is responsible for making an estimation of the future resource and workload utilization, after this anticipation, the elasticity controller will perform actions based on its policy (e.g., a set of predefined rules). Generally, the time series analysis has two main objectives. Firstly, predicting future values (points) of the time series based on the last observations (recent usage). Secondly, identifying the repeated patterns, if found, then use them to predict future values. The recent history window (resource usage) is used as input to the anticipation technique which in turn generates future values. For achieving the first objective, there are several

techniques such as Moving-Average, Auto-Regression, ARMA, Holt winter and machine learning. For example, [27], [80], [66], [69], [70], [71], [74], [77], [108], [109] use machine learning. [56], [104], [105] use Moving-Average. [52], [53], [54], [106], [107] follow Auto-Regression technique while [52], [32], [38] follow ARMA approach. Holt winter is used by [83], [105]. In order to achieve the second purpose, various techniques are used to inspect the repetitive patterns in time series: pattern matching [106], [110], Fast Fourier Transform (FFT) [106], auto correlation [158], histogram [106].

- **Model solving mechanisms** are approaches based on probabilistic model checking or mathematical modeling frameworks to study the diverse behaviours of the system and anticipate its future states such as Markov Decision Processes (MDPs), probabilistic timed automata (PTAs). [79] and [96] are examples of works that adopt model solving approaches. [58] is a more recent work that uses Alloy models to increase the performance of the model solving (i.e., most of the MDP models and combinations are built offline using a formal specification in Alloy which eliminates the runtime overhead of MDP construction for each adaptation decision).

There are other mechanisms that can be used with both reactive and proactive approaches (when accompanied with other mathematical models such as Markov Decision Process, Q-learning algorithm, Model predictive control (MPC)):

- **Reinforcement Learning (RL)** is a computational approach that depends on learning through interactions between an agent and the system or environment. The agent (decision-maker) is responsible for taking decisions for each state of the environment, trying to maximize the return reward. The agent learns from the feedback of the previous states and rewards of the system, and then it tries to scale up or down the system by choosing the right action. For example, [48], [93], [100] use RL in reactive mode while [111], [112] use RL in proactive mode.

- **Control theory** controls the system functions in reactive mode [76], [98], [99], [102], [103], but there are some cases in which they can work in proactive mode [51], [55], [103], [116]. There exist three types of these controllers: Openloop controllers, Feedback controllers, and Feedback and Feedback-forward controllers. Openloop (non feedback) controllers compute the input to the system, these controllers do not have feedback to decide whether the system is working well or not. Feedback controllers monitor the output of the system and correct the deviation against the desired goal. Feedback-forward controllers predict errors in the output, anticipate

the behavior of the system and react before errors occur. Feedback and feedback-forward controllers are usually combined.

- **Queuing theory** is a mathematical study for queues in the system taking in consideration the waiting time, arrival rate, service time, etc. Queuing theory is intended for systems with a stationary nature. It can be used to model applications (single or multi-tiers). [61], [84], [101], [85] use queuing theory in reactive mode while [101], [112], [113], [114], [115] adhere to the queuing theory principles in predictive mode. For example, [113] proposes a model that estimates the resources required for a given workload λ, the mean response time, and other parameters. [114] uses queue length and inverse model to anticipate capacity requirement taking into consideration also the target response time.

Before finishing this section, it is worth mentioning that many works generally span across different subcategories, use more than one technique and that is why they appear more than once in Table 2.1. Many systems and proposals adhere to use a combination of reactive and proactive policies, e.g., [70], [71] use threshold and machine learning policies. [56] implements threshold-based rules and moving average while [53] uses thresholds and auto-regression. [83] uses thresholds and holt-winter. [61], [84] combine thresholds based rules and queuing theory in reactive mode only. Similarly, [98], [99] use dynamic thresholds and queuing theory while [93] combines thresholds and enforcement learning. [38] uses static thresholds for CPU and memory usage, ARMA to predict the number of requests for Web applications. Other works used more than one technique in proactive mode. For example, [52] implements auto-regression and ARMA. [112] uses reinforcement learning and queuing theory. [106] combines auto-regression and pattern matching.

2.2.2.5 Method

To deploy the elasticity solutions, one or hybrid action of the following methods is implemented: horizontal scaling, vertical scaling. Horizontal elasticity allows adding new instances while vertical elasticity, referred to as fine-grained resource provisioning, allows resizing the resources of the instance itself to cope with the runtime demand. The instances can be VMs, containers, or application modules. Horizontal and vertical techniques have their advantages and shortcomings. Horizontal elasticity is simple to implement and it is supported by hypervisors. It has been widely adopted by many commercial providers. However, horizontal elasticity can lead to inefficient utilization of

the resources due to the fact that it provides fixed or static instances, which sometimes cannot fit exactly with the required demand. On the contrary, vertical elasticity allows resizing the instances but it is not fully supported by all hypervisors, although new hypervisors such as Xen, VMware support it.

- **Horizontal scaling** is the process of adding/removing instances, which may be located at different locations. Load balancers are used to distribute the load among the different instances. It is the most widely implemented method, most cloud providers such as Amazon [26], AzureWatch [47], and many other academic works as shown in Table 2.1 use this method.

- **Vertical scaling** is the process of modifying resources (CPU, memory, storage or both) size for an instance at runtime. It gives more flexibility for the cloud systems to cope with the varying workloads. There are many works [55], [114], [35], [123], [124] that only focus on CPU vertical resizing, other works [50], [125], [126] focus on memory resizing. It is worth noting that, there have been many techniques used in literature for memory resizing such as EMA, page faults, ballooning [130]. While there exist some proposals [28], [127], [128], [129], [131] that control both resources (CPU, memory). [130] is a particular work that not only controls both resources (CPU, memory) but also coordinates the degree of vertical resizing of the CPU in relation to the memory. [83] proposes a mechanism to resize CPU, Disk, and memory. ProfitBricks and RightScale cloud providers offer this feature to their customers.
  Migration can be also considered as a needed action to further allow the vertical scaling when there is no enough resources on the host machine. However, it is also used for other purposes such as migrating a VM to a less loaded physical machine just to guarantee its performance, etc. Several types of migration are deployed such as live migration [35], [44], [73], [77], [137] and no-live migration [159]. Live migration has two main approaches post-copy [139] and pre-copy [132]. Post-copy migration suspends the migrating VM, copies minimal processor state to the target host, resumes the VM and then begins fetching memory pages from the source [160]. In pre-copy approach, the memory pages are copied while the VM is running on the source. If some pages changed (called dirty pages) during the memory copy process, they will be recopied until the number of recopied pages is greater than dirty pages, or the source VM will be stopped, and the remaining dirty pages will be copied to the destination VM.

Before performing migration or replication, a Resource Allocation Strategy (RAS) [161] is used. RAS decides where the destination or new instance will be allocated or created,

on which server, on which cloud data center. RAS is based on cost and speed of VM, the CPU usage of the physical machine, the load conditions specified by the user, the maximum profit [161], etc.

Many works have used a combination of the previously described methods. [34], [43] proposals implement replication and migration methods. [62] proposes an approach that creates new small replicas and then attaches them to load balancer or deploys a new big server and removes the previous server. The application is then reconfigured to use the provided new resources. [80] proposes a framework that uses a combination of vertical resizing (adding resources to existing VM instances) or horizontal scaling (adding new VM instances). A recent work [97] around elasticity proposes a Cloud Resource Description Model (cRDM) based on a state machine for describing the horizontal and vertical elasticity. Authors in [97] defined fixed set of states for the cloud resource or application where the elasticity events and transitions loop inside the defined space. [77] reconfigures CPU and memory, live migration is triggered when there is no sufficient resources. [35] configures CPU voltage and frequency and it also uses live migration.

2.2.2.6 Architecture

The architecture of the elasticity management systems can be either centralized or decentralized. Centralized architecture has only one elasticity controller, i.e., the auto-scaling system that provisions and deprovisions resources. Most solutions presented in the academic literature and business world have a centralized architecture while there are some solutions that are decentralized such as [141] and [142]. In decentralized solutions, the architecture is composed of many elasticity controllers or application managers, which are responsible for provisioning resources for different cloud-hosted platforms. In addition to an arbiter which is the key master component in a decentralized approach because it is charged to allocate resources to the other controllers at the different system components. Multi-Agent Systems (MAS) also represent a distributed computing paradigm based on multiple interacting agents. The interacting agents with cloud shape a new discipline called agent-based cloud computing. Multiple agents allow cloud computing to be more flexible and more autonomous [162]. MAS technologies have been used to decentralize the elasticity management decision [163]. Some examples of existing works using MAS for cloud elasticity, cloud service reservation, and SLA negotiation include [87], [86], [143], [144].

2.2.2.7 Provider

Elastic solutions can be applied to a single or multiple cloud providers. A single cloud provider can be either public or private with one or multiple regions or datacenters. Multiple clouds in this context means more than one cloud provider. It includes hybrid clouds that can be private or public, in addition to the federated clouds and cloud bursting. Most of the elasticity solutions and proposals support only a single cloud provider. However, there are other works [57], [82], [90], [91], [140], [97] that handle elasticity between multiple cloud providers simultaneously.

2.2.3 Elasticity performance evaluation

Experiments are very important for the performance evaluation of elastic cloud systems. However, there is no standard method for evaluating auto-scaling and elasticity techniques due to the uncertainties in the workloads and unexpected behaviors of the system. Therefore, researchers use different testing environments according to their own needs. We introduce the common experimental platforms, workloads, and application benchmarks, as shown in Figure 2.3, that have been used in the literature.

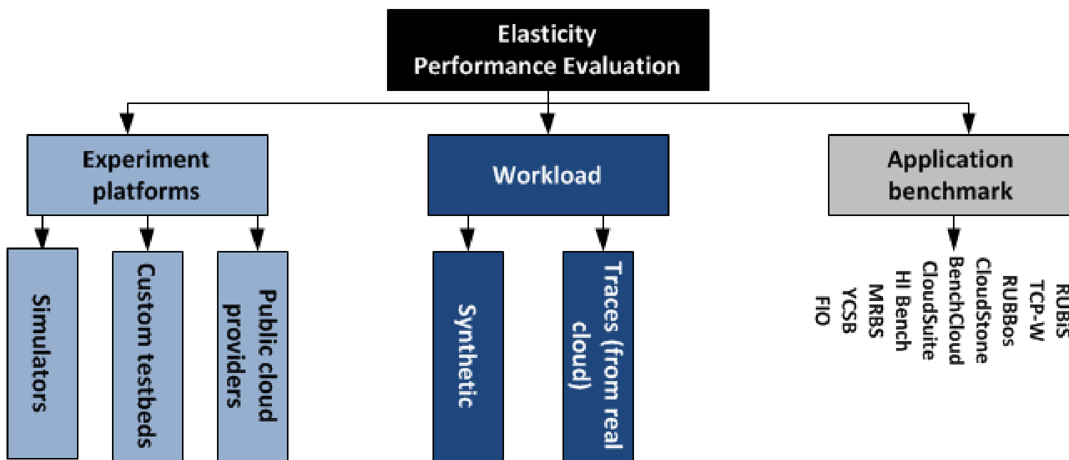


Figure 2.3 Performance evaluation tools

2.2.3.1 Experimental platforms

Experiments can be achieved using simulators, custom testbeds or real cloud providers. **Simulators** are widely used to simulate cloud platforms [158]. Using simulators in

evaluating elasticity systems and application behaviors offer significant benefits, as they allow developers to test the performance of their systems in a repeatable and controllable free of cost environment and they also allow to tune the performance bottlenecks before real-world deployments on commercial clouds. Major cloud simulators are:

- CloudSim [164]: a powerful framework for modeling and simulation of cloud computing infrastructures and services. It is widely used in research works.
- ContainerCloudSim [165] is another simulation tool that integrates most functionalities of CloudSim. It aims to provide support for modeling and simulation of containerized cloud computing environments. It supports modeling and simulation for container resource management, placement, migration on the simulated cloud environment.
- GreenCloud [166]: a framework used to develop novel solutions in monitoring, resource allocation, workload scheduling, as well packet-level simulator for energy-aware cloud computing data centers.
- OMNeT++ [167]: a framework used primarily for building network simulators but it is also used for cloud platforms.
- iCanCloud [168]: targeted to conduct large experiments, provides a flexible and customizable global hypervisor for integrating any cloud brokering policy.
- SimGrid [169]: a simulator for large-scale distributed systems such as clouds.
- EMUSIM [170]: an integrated emulation and simulation environment for modeling, evaluation, and validation of the performance of cloud computing applications.

Custom testbeds offer more control on the platform, but they require extensive efforts for system configuration. For deploying custom testbeds or clouds, many technologies are used such as hypervisors (Xen, VMWare ESXi, KVM, etc.), cloud orchestrators such as OpenStack, CloudStack, OpenNebula, Eucalyptus, and the commercial VCloud. Academic cloud testbeds such as Grid5000 [171], FutureGrid, open research clouds are also widely used.

Public clouds. While achieving experiments on a real cloud reflects the reality, it has a big drawback: there are external factors that cannot be controlled, which could impact negatively the tested system. In addition, a cloud provider offers the infrastructure (on which the experiment will be launched), but monitoring and auto-scaling system, application benchmark, workload generators are still needed.

2.2.3.2 Workloads

User requests or demand together with timestamps are required for the tested platforms to derive the experiments. Workloads can be synthetic or real.

- **Synthetic workloads** are generated with special programs in a form of different patterns. Faban, JMeter, httperf, Rain are examples of workload generators.

- **Real workloads** are obtained from real cloud platforms and stored into trace files. World cup [172], Clark net [173], and Google Cluster trace [174] are examples of real workloads. Different application workloads have different characteristics. Therefore, there exists no single elasticity algorithm which is perfect for the diverse types of workloads. Workload analysis and classification tools [56], [175] are used to analyze workloads and assign them to the most suitable elasticity controller based on the workload characteristics and business objectives.

2.2.3.3 Application benchmark

To test the scale up/down and scale out/in capabilities of a cloud platform, a set of cloud benchmarks are widely used. Benchmarks are commonly used to evaluate the performance and scalability of the servers [158]. Experiments are conducted mainly on all cloud platforms and models including IaaS, PaaS, SaaS, etc. Benchmarks have both applications and generators. RUBBos [176], RUBiS [177], TCP-W [178], CloudStone [179], YCSB [180], MRBS [181] and FIO [182], BenchCloud at USC, CloudSuite [183], and HI Bench [184], are well-known benchmarking platforms.

2.3 Containerization

This section discusses container technologies, their pros and cons. We then present the concepts and surrounding technologies behind containers. Finally, we discuss works from literature related to elasticity of containers.

2.3.1 Pros and Cons

Hypervisors are the most widely used virtualization techniques in cloud computing. However, with the need of more flexibility, scalability, and resource efficiency, cloud providers are tapping hands-on into containers [185]. Containers or what is referred to as operating system-level virtualization have evolved rapidly. Container-based virtualization is much more lightweight and resource efficient than VM-based virtualization. Containers isolate processes on the core-level of the OS. In other words, they share the same OS and they do not need guest OS, which allows to manage resources efficiently and have more instances on the same server. The use of containers eliminates the hypervisor layer, redundant OS kernels, libraries, and binaries, which are needed to run workloads or applications in a virtual machine with the classical hypervisor virtualization. On the contrary, the traditional hypervisor virtualization requires a full OS on the VM, which consumes resources and causes an extra overhead. Figure 2.4 compares application deployment using a hypervisor and a container manager. As shown in Figure 2.4, the hypervisor-based deployment requires different operating systems and adds an extra layer of virtualization compared to containerization.

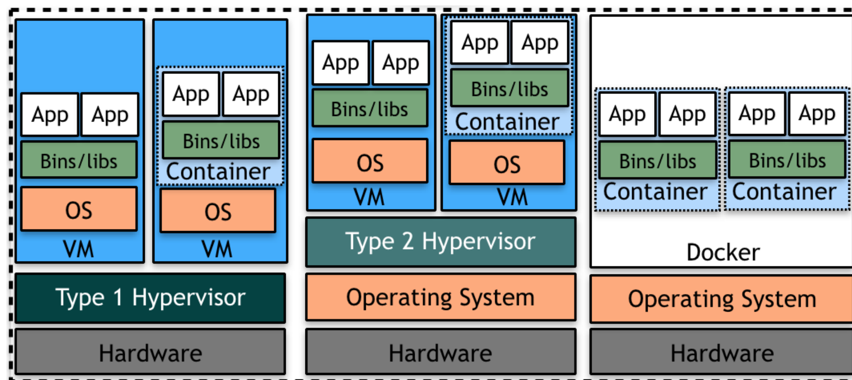


Figure 2.4 Container-based virtualization vs. traditional virtualization

Container technologies provide some advantages such as:

- Containers decrease the start up time, processing and storage overhead when compared to the traditional VMs [186].
- Containers isolate and control processes and resources. Namespaces provide an isolation per process. In Linux OS, cgroups isolate resource usage such as memory, CPU, block I/O and provide resource management. Namespaces and cgroups do not incur overhead or performance penalty.

- Containers solve the issues of portability and consistency between environments [187].

While containerization technology offers many advantages, it has the following shortcomings:

- The use of containers poses security implications. The user processes are isolated on the shared OS but it is hard, at least until now, to provide the same level of isolation between containers as VMs do.
- Since development of new container managers such as Docker is recent, it lacks many functionalities. The development is in progress in this attractive domain.
- New container standards support only 64 bit systems.

2.3.2 Container technologies

The concept of containers has existed for over a decade. Mainstream Unix-based operating systems, such as Solaris, FreeBSD, Linux, had built-in support for containers. The interest in containers led to many actors to develop solutions. There are various implementations of containers such as:

Docker [188] is an open source management tool for containers that automates the deployment of applications. Docker uses a client-server architecture and it consists of three main components: *Docker client*, *Docker host* and *Docker registry*. Docker host represents the hosting machine on which Docker daemon and containers run. Docker daemon is responsible for building, running, and distributing the containers. Docker client is the user interface to Docker.

Rocket (rkt) is an emerging new container technology. With the advent of CoreOS [189], a new container called Rocket is introduced. Besides rkt containers, CoreOS supports Docker. Rocket was designed to be a more secure, interoperable, and open container solution. Rocket is a new competitor for Docker.

LinuX Containers (LXC) [190] is an operating system-level virtualization method for running multiple isolated Linux systems. It uses kernel-level namespaces to isolate the container from the host.

LXD [191] is a lightweight hypervisor, designed by Canonical, for Linux containers built on top of LXC to provide a new and better user experience. LXD and Docker make use of LXC containers.

Others: there are other open source light virtualization technologies such as BSDJail [192] and OpenVZ [193].

Docker and Rocket are the most recent used container technologies due to their enhanced features. We present some of their surrounding technologies [194] in Figure 2.5. Docker uses `runc` and `libcontainer` runtimes that enable interactions with Linux kernel components (cgroups, namespaces) to create and control containers. Rocket uses rkt and CoreOS runtimes. For the management, Docker uses Docker Engine that includes both Docker daemon and Docker client for interacting with Docker daemon. Docker daemon provides an API that abstracts container control functions. Rkt CLI is the container management functionality in Rocket. Docker containers can be defined using Docker images where container instances are created from these images. The images are created with Dockerfiles, text files containing all the commands needed to build Docker images. Rkt supports Docker images, as well as Application Container Images (ACI). Docker registry is the service responsible for storing and distributing images.

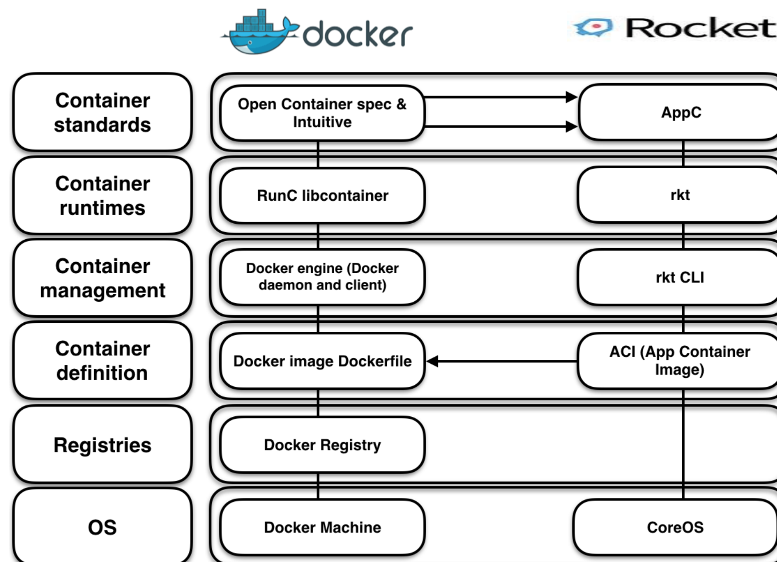


Figure 2.5 Docker and rocket Technologies

2.3.3 Container orchestration and management tools

Container adoption is expected to grow across all application life cycle steps, especially the production phase. However, some applications typically deal with workloads that have dozens of containers running across multiple hosts. This complex architecture dealing with multiple hosts and containers demands a new set of management tools.

Docker Swarm is a well-known clustering management tool for Docker containers. Swarm makes use of the Docker standard interface (API) in order to achieve its tasks such as starting Docker, choosing host to run containers on. Swarm consists of Swarm agents and a Swarm manager. Swarm agents are run on each host, the manager orchestrates, schedules containers on the hosts. Swarm uses discovery process to add hosts to the cluster, and it supports both Rocket and Docker containers. Swarm uses Docker-compose to support horizontal elasticity.

Kubernetes is another powerful container orchestration tool built by Google [195]. Kubernetes has brought new concepts about how containers are organized and networked. Along with managing single containers, it manages pods. Pod is a group of containers that can be created, deployed, scheduled and destroyed together. Kubernetes supports flat networking space, containers in a pod share the same IP, where pods can talk to each other without the need for NAT. In Kubernetes, replication controllers are responsible for controlling, and monitoring the number of running pods (called replicas) for a service [196], when a replica fails, a new one will be launched, and this improves reliability and fault tolerance. Kubernetes supports horizontal elasticity via its internal Horizontal Pod Autoscaling (HPA) system. HPA allows to automatically scale the number of pods based on observed CPU utilization. It uses reactive threshold-based rules for CPU utilization metric [197].

CoreOS Fleet is a cluster management tool that represents the entire cluster as a single init system [198]. Fleet is a low-level cluster management tool that allows a higher-level solution such as Kubernetes to be settled on the top. It provides a flexible management for the containers: fleet can start, stop containers, get information about the running services or containers in the different machines of the cluster, migrate containers from one host to another. It is designed to be fault-tolerant, and it supports both Rocket and Docker containers.

Apache Mesos [199] is an open-source cluster manager designed to manage and deploy application containers in large-scale clustered environments. Mesos, alongside with a

job system like Marathon, takes care of scheduling and running jobs and tasks. It also supports horizontal elasticity.

OpenStack Magnum is a project that facilitates the utilization of container technology in OpenStack. It adds multi-tenant integration of prevailing container orchestration software for use in OpenStack clouds.

Rancher is a unified open source cluster management tool for Docker containers. Rancher is built on the top of other orchestrators such as Kubernetes, Docker Swarm, Apache Mesos, etc. and makes it easy to run different clusters of containers. In this chapter, we have briefly presented some of the container orchestrators, however, there are other orchestrators such as OpenShift, Cattle, etc.

Figure 2.6 shows some of the most used orchestration tools that are used to run applications on a distributed cluster of machines. These tools use service discovery such as etcd, Zookeeper, or Consul to distribute information between services or cluster hosts.

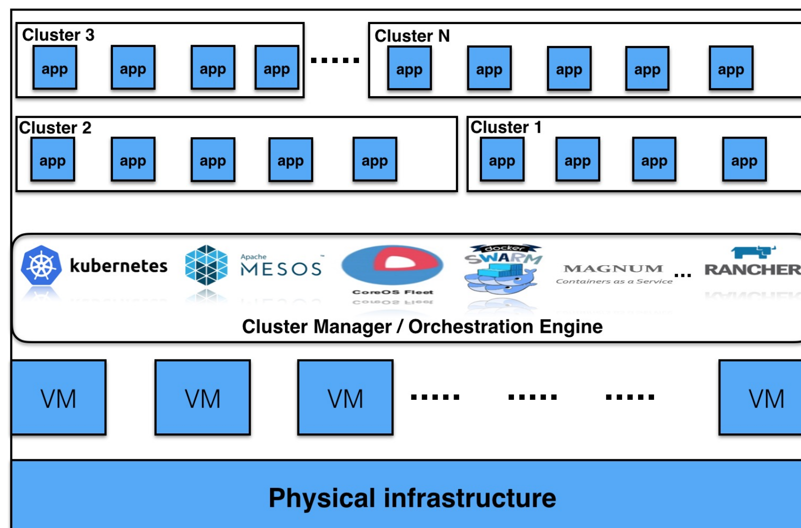


Figure 2.6 Container orchestration engines

2.3.4 Elasticity of containers

Although containers are gaining wide-spread popularity among cloud providers, there are few works addressing elasticity of containers. As shown in Figure 2.2, the elasticity solutions utilize various policies and methods. They have different purposes, configurations, and architectures. These mechanisms applied to the VMs can also be applied to containers as described below.

[39] proposes a design of a system used for developing and automatically deploying micro services. The proposed approach manages more instances of the application when load increases and scales down/in for fewer demands to conserve energy. The requests count and memory load are monitored, when they arrive a certain threshold, containers are scaled out or in. A replication method is used to achieve horizontal elasticity of container instances.

[36] proposes a control architecture that dynamically and elastically adjusts VMs and containers provisioning. In this work, containers and VMs can be adjusted vertically (by varying the computational resources available for instances) or horizontally (change the number of instances) according to an objective function that searches to minimize costs.

[37] proposes a framework called MultiBox. MultiBox is a means for creating and migrating containers among cloud service providers. MultiBox makes use of the Linux cgroups to create and migrate containers that are isolated from the rest of the host OS. MultiBox containers support both stateful and stateless applications.

[200] proposes an approach for the application live migration in Linux container for better resource provisioning and interoperability. This approach uses Checkpoint/Restore In Userspace (CRIU) [201], a Linux functionality that allows container live migration.

Promox VE [40] also permits manual vertical resizing and migration for the LXC [190] and OpenVZ [193] containers. Promox VE is an open source server virtualization management software.

DoCloud [38] is an elastic cloud platform based on Docker. It permits to add or remove Docker containers to adapt Web application resource requirements. In DoCloud, a hybrid elasticity controller is proposed that uses proactive and reactive models to scale out and proactive model to scale in. Since cloud elasticity with containers is in its infancy, almost all the elasticity actions in container elasticity solutions are performed using reactive approach that is based on pre-defined thresholds. However, DoCloud uses dynamic re-dimension method or predictive approaches to trigger elasticity actions. It uses a hybrid reactive, proactive controller that adopts threshold and ARMA approaches.

[7] proposes a model-driven tool to ensure the deployability and the management of Docker containers. It allows synchronization between the designed containers and those deployed. In addition, it allows to manually adjust container vertical elasticity.

[78] proposes a horizontal and vertical auto-scaling technique based on a discrete-time feedback controller for VMs and containers. This novel framework allows coordinating

infrastructure and platform adaptation for web applications. The application requirements and metadata must be precisely defined to enable the system to work. It inserts agents for each container and VM for monitoring and self-adaptation.

[41] introduces a new container-based adaptation method which applies dynamic rules to control the containers horizontal elasticity. This paper presents a set of adaptation rules without learning mechanisms to increase or decrease the right total number of container instances in order to accommodate varied workloads.

As described in the works related to container elasticity, containers can be scaled horizontally and vertically. However, in order to implement the mechanisms used in VMs, some modifications are needed. For example, in reactive approaches, breath duration is a period of time left to give the system a chance to reach a stable state after each scaling decision, since containers adapt very quickly to workload demand, breath duration must be small when compared to VM. To our knowledge, there is no work that adopts proactive approaches to scale containers except [38] which uses ARMA prediction. In addition, container adaptations, its hosted application adaptations and the monitoring system may differ from VM because of the divergence of technology.

Recently, many cloud providers such as Amazon EC2 Container Service, Google Container Engine, Docker Datacenter, Rackspace adopt containers in their cloud infrastructure and offer them to clients.

2.4 Open Issues and Research Challenges

Despite the diverse studies developed about elasticity in cloud computing. There are still many open issues about elasticity in general and research challenges about elasticity in the container emerging technology that the cloud providers and research academy have to deal with.
Open issues about elasticity are:

- **Interoperability**: In order to provide redundancy and ensure reliability, the resources (compute, storage, etc.) should be seamlessly leased from different cloud providers or data centers to the clients. Cloud providers use their own technology and techniques according to their policy, budget, technical skills, etc. Therefore, it is difficult to use multiple clouds to provide resources due to the incompatibilities between them. The combined use of diverse cloud providers remains a challenge because of the lack of standardized APIs, each provider has its own method on how users and applications

interact with the cloud infrastructure. It is not only the job of research to solve this challenge, rather the industry needs to agree on standards. Though there are some academic works that allow allocating resources from different providers or data centers, they are limited to certain criteria, for example, [82] allows to allocate resources according to the price offered or spot that matches the user's bid.

- **Granularity**: As seen in Section 2.2.2, IaaS providers offer a fixed set of resources such as Amazon instances, though some users or applications have different needs, as an example, some applications need more CPU than memory. Generally, there must be a coordination in the resource provisioning or de-provisioning. Most of elasticity strategies are based on the horizontal elasticity. Thus, vertical elasticity is very important to provide a related combination of resources according to the demand. There are many academic works [114], [131], [125] which resize CPU, memory or both but there is no coordination between CPU and memory controllers. They resize CPU and memory without regarding the coordination between them. There are just a few works such as [130] which coordinates the provisioning of both resources. In addition to the resource granularity, billing granularity is another issue. Cloud providers charge clients based on the resource consumption per fixed time unit, almost all cloud providers use hour as a minimal billing time unit. For example, using this billing system, VM is billed for an hour even when used for 5 minutes. Few providers CloudSigma [22], VPS.NET [202] allow to use fine-grained billing system where the client will pay approximately its real consumption of resources. The type of the elasticity method has a great impact on the pricing model. For example, implementing vertical elasticity is accompanied by shifting towards fine-grained pricing policies while using horizontal elasticity leads to extra costs since it uses instances (i.e., VMs) as scaling units (coarse-grained scale) and it also implies running load-balancer (i.e., additional consumption of resources). In addition, container billing is another pricing ambiguity. Since containers are being recently used in production environments, there is no standard pricing model for containers. For example, Amazon charges by VM instance for the Amazon EC2 Container Service. Containers are usually accompanied by orchestrators and cluster of nodes, and the container may settle on VM or on a bare-metal host, therefore, there is still no standard pricing model.
- **Resource availability**: The resource offered by the cloud providers are limited. Therefore, the elasticity of scaling resources is limited by the capacity of the cloud infrastructure. In practice, no cloud provider offers unlimited resources to its clients, but big providers such as Google and Amazon are conceptually unlimited for typical

users. However, temporal network bottlenecks, limited geographical locations, higher latency, etc. may hinder the provisioning of resources.

- **Hybrid solutions**: Reactive and proactive approaches have their advantages and drawbacks. Therefore, a sophisticated solution could combine both reactive and proactive approaches and methods such as horizontal and vertical scaling.

- **Start-up time** or spin-up time is defined as the time needed to allocate resources in response to the client demand. Start-up time can reach several minutes but the worse is that the users (clients/customers) are charged directly once they make their requests to scale-up or scale-down resources before acquiring the resource. Provisioning resources may arrive late, and there are chargeable costs, which are different from the real costs that match the provided resources. Start-up time might be fast or slow, it depends on several factors such as cloud layer (IaaS or PaaS), target operating system, number of requested VMs, VM size, resource availability in the region and elasticity mechanism. The lower the start up time is, the better the elastic solution is. Higher start up time affects the efficiency of elasticity system [203].

- **Thresholds definition**: As we have discussed in Section 2.2.4, threshold-based mechanisms are based on defining thresholds for the measured metrics such as CPU or memory utilization. Choosing suitable thresholds is not an easy task, it is very tricky due to the workload or application behavior changes, that makes the accuracy of the elasticity rules subjective and prone to uncertainty. This can lead to instability of the system. Therefore, it is necessary to have an intelligent self-adaptation system to deal with these uncertainties.

- **Prediction-estimation error**: Proactive techniques anticipate changes in the workload and react in advance to scale-up or scale-down the resources. Herein the start-up time issue is handled using these approaches, however, they could yield errors or what is called prediction-estimation error. Estimation error can lead to resources over-provisioning or under-provisioning. Proactive approaches are characterized as complicated and sophisticated solutions, however, they are not accurate in some cases, and this also depends on the application behavior, unexpected workload changes such as sudden burst or decrease. Some applications are hard to predict, in consequence, predictive techniques can deviate from the intended objectives. Having efficient prediction error handling mechanisms to meet application SLOs with minimum resource cost is worth considering.

- **Optimal trade-off between the user's requirements and provider's interests**: There is a contradiction between provider's profit and user's QoE [89]. Users' QoE is defined as the user satisfaction towards a service. The users search to increase their QoE with the best price and to avoid inadequate provision of resources. While the cloud providers search to increase their profit with providing good QoS services, which means elasticity must ensure better use of computing resources and more energy savings and allows multiple users to be served simultaneously. In addition, due to the market concurrence, cloud providers have to offer cost-effective and QoS-aware services. Therefore, finding an optimal trade-off between user-centric (response time, budget spent, etc.) and provider-centric (reliability, availability, profit) requirements is a big challenge. Offering good QoS will increase customers' satisfaction, this will reflect a good reputation for the provider, and the number of consumers will increase. Hence, the better QoE, the better profits can come from the satisfied customers. Generally, integrating QoE and QoS in the Cloud ecosystem is a promising research domain that is still in its early stages.

- **Unified platforms for elastic applications**: Before discussing elasticity and scalability, the application itself should be elastic. Much of the elasticity solutions implemented by the cloud providers are appropriate for certain types of applications such as server-based applications that depend on the replication of virtual instances and load balancers to distribute the workload among the instances. For that reason, what needed is the development of unified platforms, tools, languages, patterns, abstractions, architectures, etc. to support building and execution of elastic applications. These tools must take into consideration the many application characteristics such as parallelism in order to use elasticity in clouds. Developing such tools, architectures, etc. is a big challenge and worth research, particularly as there is a huge movement towards elasticity and distributed architecture in the computational clouds.

- **Evaluation methodology**: There is no common approach for evaluating elasticity solutions. It is extremely difficult to compare and evaluate different elastic approaches using a formal evaluation technique and a unified testing platform due to the heterogeneity of elastic systems, in addition to the nature of different workload behaviors. In [204], a Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications (PEAS) is proposed, however, the framework cannot be generalized on all elastic solutions and evaluation scenarios.

Research challenges about elasticity of containers are:

- **Monitoring containers**: In order to provide data to be analyzed and to make elasticity decisions or actions, monitoring is an essential part in elasticity solutions. However, it is not an easy task especially with containers. Container holds applications and all of their dependencies and in general many containers may be hosted on the same machine, therefore having stable systems that accurately and rapidly monitor multiple containers is worth searching. In fact, the monitoring challenge is not fully addressed in container technologies.

- **Container-based elasticity**: There are many sophisticated elasticity solutions for the traditional hypervisor-based virtualization. Using these solutions with containers is still an open challenge and research perspective. New container technologies such as Docker use cgroups to limit the resources consumed by a container, such as CPU, memory, disk space and I/O, and also offer metrics about these resources. A container can have static resource limits such as 1 CPU and 2G of RAM or can relativity share resources with other containers on the hosting machine. Using the latter technique, the container will get its resources in function of resource usage for the neighboring containers or applications. For some reasons such as cost and priority, static limits are set on containers. The questions which arise are: i) Can we apply the elasticity solutions used in VM on the containers? ii) How to use proactive approaches to anticipate container resource usage and react in advance to scale up/down resources? In addition, many orchestration tools such as Kubernetes, Rancher, etc. are used to manage and orchestrate clusters of containers, but integrating autonomic vertical and horizontal elasticity in these platforms is important.

- **Combined elasticity between VMs and containers**: Nowadays, cloud providers use containers on the top of virtual machines (see Figure 2.4). This allows to have many instances arranged across levels of hierarchy. Adjusting container resources such as CPU, RAM, etc. to the demand or workload at runtime will lead to efficient resource utilization, and avoid SLA violations. The problem here is that resizing container resources is limited by the resources of the virtual machine in which it is placed. After certain limits, the container cannot gain more resources, fortunately the VM could be resized by its hypervisor, which by its turn will allow to further resize the container. The challenge to coordinate elasticity between the virtual machine and its placed containers remains unaddressed. Achieving elasticity control for VM and containers will allow a great flexibility and would be an efficient elasticity solution.

2.5 Related Work

In this section, some of the related works that are relevant to our work are presented. Being the key property behind cloud computing, several works on elasticity are carried out involving various elasticity approaches that depend on the infrastructure, application or workload behavior. [10] is an old survey, it proposes a basic classification for elasticity solutions based on only four characteristics: scope, policy, purpose and method. In addition, the discussion about these characteristics is limited. New characteristics and even new subcategories have appeared in more recent elastic solutions such as the different techniques in workload anticipation in proactive mode. [18] proposes a classification of the techniques for managing elasticity based on strategy and action. The concentration in this chapter is on the elasticity strategy. The strategy in this context studies elasticity management solutions based on the quality goal. The quality goal can be the Quality of Business or the Quality of Service from the Cloud Provider (CP) and Application Service Provider (ASP) perspectives. Quality of Business refers to the service provider's revenue/profit, satisfaction. Three solutions are evaluated based on this proposition depending on the strategy adopted and whether reactive or proactive action is followed to achieve elasticity. [158] concentrates mainly on the auto-scaling reactive and proactive approaches and elasticity tools. This work is limited to auto-scaling techniques and experimentation tools. [9] addresses the elasticity definition, metrics and tools. It brought many elasticity definitions, in addition to statistical information about elasticity, such as the number of papers published per year, per country. [19] is another work on cloud elasticity. It is a complementary to our work, but we present elasticity strategies and research challenges in more broader fashion. For example, the mechanisms that can be reactive or proactive, we clearly identified solutions that use these mechanisms in each subcategory. [205] provides a survey of auto-scaling techniques for web applications. According to this work, the actions of auto-scaling systems are based on performance indicators that can be high or low level metrics. Low level metrics such as CPU utilization are performance indicators observed at the server layer while high level metrics such as response time are performance indicators observed at the application layer. This survey is limited to one category of applications, i.e., web applications. A more recent survey of control theoretic aspects of cloud elasticity is proposed in [206]. It provides a thorough review and classification of elasticity related works that use control theory. Our work differs from the above works in the following aspects: firstly, a complete overview of the mechanisms implemented in the elasticity solutions is provided, an extended classification is proposed including the embedded elasticity. We have described elasticity based on seven

characteristics: configuration, scope, purpose, mode, method, provider and architecture. We have further classified each approach into sub mechanisms. For example, time series analysis is a proactive approach that anticipates workloads. It uses many mechanisms: moving average, auto regression, ARMA, holt winter and machine learning; we have provided examples for each case. Secondly, contrary to all previous surveys, this chapter is the first that presents works related to container elasticity. Finally, challenges and research perspectives for both VMs and containers are handled in a broader context according to our point of view.

2.6 Conclusion

Cloud computing is becoming increasingly popular; it is being used extensively by many enterprises with a rapid growing. The key feature that makes cloud platforms attractive is elasticity. Elasticity allows providing elastic resources according to the needs in an optimal way. In this chapter, a comprehensive study about elasticity is provided. It started by talking about the elasticity definitions, and its related terms scalability and efficiency. We have suggested an extended classification for the elasticity strategies based on the existing academic and commercial solutions. The proposed classification or taxonomy covers many features and aspects of the cloud elasticity based on the analysis of diverse proposals. Each aspect is then discussed in details providing examples from the proposed proposals that handle cloud elasticity. We have presented the containerization and the orchestration tools where elasticity will be popular in this technology. Many works on the container elasticity are presented. Finally, challenges and new research perspectives are presented.

From this state of art, we notice that there are only a few works related to the elasticity of containers. Therefore, we decided to investigate the elasticity aspects around containers. In addition, there is no work that manages the different aspects of elasticity by using a unified, standardized, modular approach at runtime in a seamless way. Therefore, this thesis concentrates on elasticity management of containers and a model-driven approach to completely handle the elasticity horizon. We present the motivation behind each proposal in the next chapters. Chapter 3, Chapter 4, Chapter 5 and Chapter 6 try to solve some challenges and research questions presented in Chapter 1 and Chapter 2.

Part II

Elastic Docker

Chapter 3

ElasticDocker Principles

In the literature as well as in industrial products, much attention was given to the elasticity of virtual machines, but much less to the elasticity of containers. However, containers are the new trend for packaging and deploying microservices-based applications. Moreover, most of approaches focus on horizontal elasticity, fewer works address vertical elasticity. In this chapter, we propose ElasticDocker*, the first system powering vertical elasticity of Docker containers autonomously. Based on the well-known IBM's autonomic computing MAPE-K principles,* ElasticDocker *scales up and down both CPU and memory assigned to each container according to the application workload. We propose a new extension, yet a new elasticity controller that coordinates vertical elasticity at both containers and VMs levels. As vertical elasticity is limited to the host machine capacity,* ElasticDocker *does container live migration when there is no enough resources on the hosting machine.*

This chapter is derived from:

- ***Yahya Al-Dhuraibi****, Fawaz Paraiso, Nabil Djarallah and Philippe Merle, "Autonomic Vertical Elasticity of Docker Containers with ElasticDocker," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, CA, 2017, pp. 472-479, doi: 10.1109/CLOUD.2017.67*
- ***Yahya Al-Dhuraibi****, Faiez Zalila, Nabil Djarallah, Philippe Merle. Coordinating Vertical Elasticity of both Containers and Virtual Machines. 8th International Conference on Cloud Computing and Services Science - CLOSER 2018, Mar 2018, Funchal, Madeira, Portugal. 2018.*
- *Fawaz Paraiso, Stéphanie Challita,* ***Yahya Al-Dhuraibi*** *and Philippe Merle, "Model-Driven Management of Docker Containers," 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2016, pp. 718-725. doi: 10.1109/CLOUD.2016.0100*

3.1 Introduction

Elasticity is one of the key characteristics of cloud computing, which leads to its widespread adoption. Elasticity is defined as the ability to adaptively and timely scale computing resources in order to meet varying workload demands [9, 2]. There are two types of elasticity: horizontal and vertical [9], [114]. Horizontal elasticity consists in adding or removing instances of computing resources associated to an application. Horizontal elasticity is also known as replication of resources. Vertical elasticity consists in increasing or decreasing characteristics of computing resources, such as CPU time, cores, memory, and network bandwidth. Vertical elasticity is also known as resizing of resources. Both elasticities are driven by the variation of workload demands, such as the request response time or the number of end-users. In the scientific literature but also in industry practices, most of proposed approaches focus on horizontal elasticity but few addresses vertical elasticity.

Virtualization techniques are the keystone of elasticity in cloud computing and consist to virtualize the actual physical resources – e.g., CPU, storage, network – as virtual resources such as virtual machines (VMs), virtual storages, virtual networks. Numerous works proposed various cloud elasticity handling mechanisms for VMs [114], [57], [130], [127]. However, with the advent of Docker [207], containers are becoming the new trend for packaging and deploying microservices-based applications [208]. Since Docker provides more flexibility, scalability, and resource efficiency than VMs [209], [203], [186], it becomes popular to bundle applications and their libraries in lightweight Linux containers and offers them to the public via the cloud. Then, Docker containers have gained a widespread deployment in cloud infrastructures such as in Amazon EC2 Container Service, Google Container Engine, Docker Datacenter, Rackspace. But compared to VMs, there are only few works that deal with elasticity of Docker containers: [39], [37], [38] focus on automatic horizontal elasticity, [7] address manual vertical elasticity, [36] supports migration. To the best of our knowledge, there is no related work that handles vertical elasticity of containers autonomously.

The main contribution of this chapter is to present ElasticDocker: the first system powering vertical elasticity of Docker containers autonomously. Based on the well-known IBM's autonomic computing MAPE-K principles [210], ElasticDocker scales up and down both CPU and memory assigned to each container when the application workload grows up and down, respectively. This approach modifies resource limits directly in the Linux control groups (cgroups) associated to Docker containers. As indicated before,

many studies address the vertical elasticity of VMs and ElasticDocker handles vertical elasticity of containers, no work manages the coordination between these two vertical elasticities. Towards investigating all the possible approaches of elasticity, we also propose the first approach - coordinated controller - to coordinate vertical elasticity of both VMs and containers. The coordinated controller comprises three components, ElasticDocker, VM elasticity controller and coordination between them. Vertical elasticity is limited to host machine capacity as it cannot provision more resources when all the host machine resources are already allocated to containers. Therefore, in this work, we use live migration to handle this limit. Live migration is the process of moving a container in its executing state from source to target host. Container migration takes place when resizing is no longer possible on the host machine. ElasticDocker uses Checkpoint/Restore In Userspace (CRIU) [201] to implement the concept of container live migration.

This chapter is organized as follows. Section 3.2 describes the motivation for vertical elasticity of Docker containers. Section 3.3 provides the technical background on Docker. Section 3.4 presents our ElasticDocker approach to scale up/down containers. Section 3.5 describes the coordinated controller that manages both the VM and container elasticity. After that, Section 3.6 describes live migration technique. We discuss this approach and its limits in Section 3.7. Before conclusion, related works are presented in Section 3.8.

3.2 Motivation

3.2.1 Resource over-provisioning and de-provisioning

The load of cloud applications varies along with time. Diverse applications have different requirements. Therefore as discussed in Chapter 1, maintaining sufficient resources to meet workload burst and peak requirements can be costly. Conversely, maintaining minimum or medium computing resources can not meet workload's peak requirements, and cause bad performance and Service Level Objective (SLO) violations. Autonomic cloud elasticity permits to adaptively and timely scale up/down resources according to the actual workload.

3.2.2 Vertical elasticity

Elasticity is defined as the ability to adapt resources on the fly to handle load variation. There are two types of elasticity: horizontal elasticity and vertical elasticity. Horizontal elasticity requires more support from the application, so that it can be decomposed into instances. Recently, most attention has been given to horizontal elasticity management. Vertical elasticity is limited due to the fact it can not scale outside the resources provided by a single physical machine and then introduces a single point of failures. However, vertical elasticity is better when there are enough available resources. Vertical elasticity has the following characteristics:

- Vertical elasticity is fine-grained scaling while it permits to add/remove real units of resources.
- Vertical elasticity is applicable to any application, it also eliminates the overhead caused by booting instances in horizontal elasticity while horizontal elasticity is only applicable to applications that can be replicated or decomposed.
- Vertical elasticity does not need running additional machines such as load balancers or replicated instances.
- Vertical elasticity guarantees that the sessions of the application are not interrupted when scaling.
- Some applications such as MATLAB, AutoCAD do not support horizontal elasticity. They are not replicable by design. These applications are composed of components and their interconnections. These components can not be elastic, which means that it is impossible to create several instances of the same component.
- Since horizontal elasticity consists in replicating the application on different machines, some applications such as vSphere and DataCore require additional licenses for each replica. These licenses could be very expensive, while only one license is required with vertical elasticity.

Vertical elasticity maintains better performance, less SLO violations, and higher throughput. Vertical elasticity increases the performance because the elasticity controller just increases the capacity of the same instance. In horizontal scaling, the elasticity controller can add/remove instances, which impacts the application performance. This fact is verified and demonstrated by using the queuing theory in [127]. Horizontal elasticity could result to have many small instances and modeled as M/M/1, while vertical elasticity

controls one instance by varying its capacity and modeled as M/M/c in the queuing theory, where c is the number of CPUs. After solving the corresponding equations for each model, [127] founded that the response and waiting time in vertical elasticity is much less than that of horizontal elasticity for the same workload input. Additionally, [211] proves that vertical elasticity outperforms horizontal scaling in terms of performance, power, cost, and server density in the world of analytics, mainly in Hadoop MapReduce. In the next chapter, we show experimentally that ElasticDocker vertical elasticity outperforms horizontal elasticity.

3.2.3 Containers vs VMs

Docker containers are a lightweight system-level virtualization technology. Docker is used for developing, packaging, shipping, deploying and executing applications into containers, we outline this technology in details in Section 3.3. Docker requires an autonomic elastic system in order to avoid the problems of over-provisioning and under-provisioning. We present here motivations towards Docker vertical elasticity.

- Containers consume low resource because they share resources with the hosting operating system, which makes them more efficient. Therefore, we can deploy more containers than VMs on a physical machine [209].
- Containers result in equal or better performance than VMs [186].
- Containers have small image size, therefore, time of generating, distributing and downloading images is short, in addition they require less storage space [209].

ElasticDocker proposes an approach that manages autonomous vertical provisioning and deprovisioning of Docker containers on the host machine and migrates them if there is no enough resources.

3.2.4 Elasticity coordination

While VMs are ultimately the medium to provision PaaS and application components at the infrastructure layer, containers appear as a more suitable technology for application packaging and management in PaaS clouds [8]. Containers can run on VMs or on bare OS. Running containers or different containerized applications in VM or cluster of VMs is an emerging architecture used by the cloud providers such as AWS EC2 Container Service (ECS), Google Cloud Platform, MS Containers, Rackspace, etc. The VMs are

run by the hypervisors on the host. Many studies address the vertical elasticity of VMs and ElasticDocker handles vertical elasticity of containers, no work manages the coordination between these two vertical elasticities. The coordinated controller proposed in this chapter is the first system that combines and coordinates VM and container elasticity.

3.3 Background

This section gives a brief introduction of Docker technologies in order to facilitate the understanding of our work. It also elaborates Docker image filesystem and Checkpoint/Restore In Userspace (CRIU), the concept behind ElasticDocker container live migration.

3.3.1 Docker technology

3.3.1.1 Docker architecture

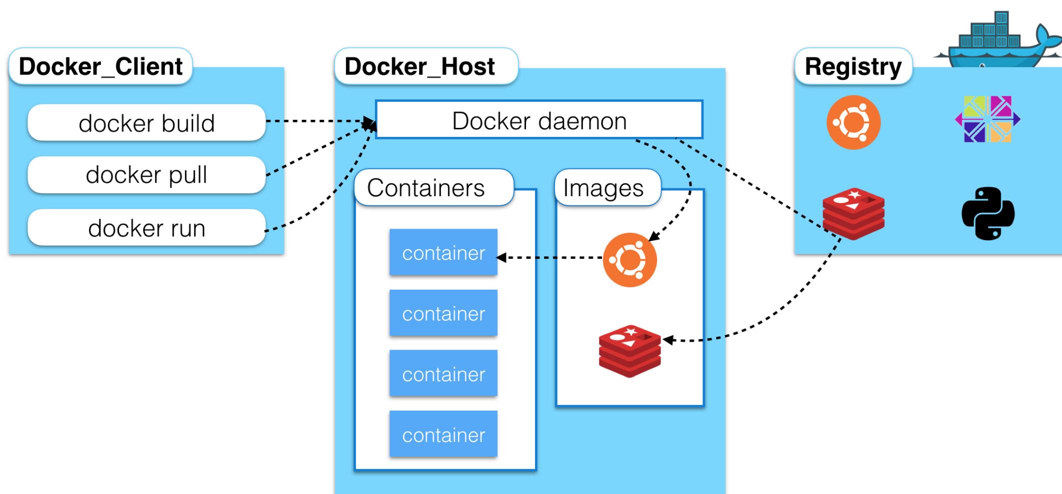


Figure 3.1 High-level overview of Docker architecture.

Docker is a lightweight virtualization technology that allows to package an application with all of its dependencies into one standardized unit for software deployment. Docker uses a client-server architecture. In Figure 3.1, we can see the major components of a Docker installation.

- At the center, the Docker host represents the physical machine or VM in which Docker daemon and containers are deployed (c.f., Figure 3.1). The Docker daemon is responsible for creating, running, and monitoring containers, as well as building and storing images. The launch of Docker daemon is normally taken care of by the host OS.
- The Docker client is on the left-hand side in Figure 3.1. It communicates with the Docker daemon via sockets through a RESTful API. The purpose of Docker client is to control the host, create images, publish, execute and manage containers corresponding to the instantiation of these images. Communication via HTTP makes it easier the remote connections to Docker daemons. The combination of Docker client and Docker daemon is called Docker engine.
- Docker registry is on the right-hand side in Figure 3.1. It stores and distributes images. The default registry is Docker Hub, which hosts thousands of public images. Docker containers are created using base images. A Docker image can include just the OS fundamentals, or it can consist of a sophisticated pre-built application stack ready for launching. To create an image, the most convenient option is to write a script file composed of various commands (instructions) named Dockerfile and then execute it. Many organizations run their own registry that can be used to store private images. Docker daemon will download images from registry in response to requests.

3.3.1.2 Resource management of Docker

Docker containers use *namespaces* to isolate resources, and *cgroups* to manage and monitor resources. Runc is a lightweight tool that runs the containers (container runtime). Runc uses libcontainer and LXC drivers, which are Linux container libraries that enable and abstract interactions with Linux kernel facilities such as cgroups and namespaces to create, control and manage containers.

Control groups (cgroups)
Docker container relies on cgroups to group processes running in the container. Cgroups (also called subsystems) allow to manage the resources of a container such as CPU, memory, and network. Cgroups not only track and manage groups of processes but also expose metrics about CPU, memory and I/O block usage. Cgroups are exposed through pseudo-filesystems. The filesystem can be found under */sys/fs/cgroups* in recent Linux distributions. Under this directory, we can access multiple subsystems in which we can

control and monitor memory, CPU cores, CPU time, I/O, etc. In these files, Docker resources can be configured to have hard or soft limits. When soft limit is configured, the container can use all resources on the host machine. However, there are other parameters that can be controlled here such as *CPU shares* that determine a relative proportional weight that the container can access the CPU. Hard limits are set to give the container a specified amount of resources, ElasticDocker changes these limits dynamically according to the container workload.

By default, Docker sets no limits, then a Docker container can use all the available resources on the host machine. It can use all the CPU cores as well as memory. The CPU access is scheduled either by using Completely Fair Scheduler (CFS) or by using Real-Time Scheduler (RTS) [212]. In CFS, CPU time is divided proportionately between Docker containers. On the other hand, RTS provides a way to specify hard limits on Docker containers or what is referred to as ceiling enforcement. Our elastic approach is integrated with RTS in order to make it elastic. Limits on Docker containers are set, our ElasticDocker scales up or down resources according to demand. Once there is no limit set, it is hard to predict how much CPU time a Docker container will be allowed to utilize. In addition, as indicated, by default Docker can use all resources on the host machine, there is no control how much resources will be used by that container (customer) as many containers (customers) can coexist on the same hosting machine. A customer may not afford to pay for such uncontrolled amount of resource. Moreover, it will be complicated for the provider to manage the customer billing system, e.g., providers usually bill the customer by instance (VM) or according to the number of CPUs, not by a partial usage of many CPUs.

3.3.1.3 Docker image filesystem

Docker builds and stores images, these images are then used to create containers. Docker image consists of a list of read-only layers that represent filesystem differences. The layers are stacked on top of each other to form the base of a container root filesystem. When container is created, a new writable layer called container layer is added on top of the underlying layers or stack. Docker supports many storage drivers such as aufs, btrfs, overlay, etc. Docker daemon can only run one storage driver, and all containers created by that daemon instance use the same storage driver. Docker storage driver and data volumes can operate on the top of the storage provided by shared storage systems. The enterprises generally consume storage from shared storage systems. In our case, AUFS is used. AUFS is a unification filesystem, which means it takes many multiple directories

(image layers), stacks them on top of each other, provides a single unified view through a single mount point. Docker hosts the filesystem as a directory under */var/lib/docker/*. AUFS branches corresponding to Docker image layers are found under this directory depending on the version of Docker. For example with Docker versions 1.9, 1.12, image layers are stored in *aufs/diff/*, the names of the directories (image layers) are the same as the image id. AUFS stores metadata about how image layers are stacked in */aufs/layers*. AUFS union mount point that exposes the container and all underlying image layers as a single unified view exists in */aufs/mnt/(container-id)*. With Docker 1.10 and higher, image layer IDs do not correspond to directory names as in Docker V 1.9. The path for the filesystem depends on Docker version and the OS, for example, AUFS exists in ubuntu 16.04 with Docker 1.10 under the directory */var/lib/docker/0.0/*.

3.3.2 Checkpoint/Restore In Userspace (CRIU)

CRIU is a Linux functionality that allows to checkpoint/restore processes. It has the ability to save the state of a running application so that it can latter resume its execution from the time of the checkpoint. Our migration approach for Docker containers with CRIU can be divided in two steps, checkpoint and restore, in addition to copy process if the dumped files do not reside on a shared file system between the source and target hosts [213].

3.3.2.1 Checkpoint

The checkpoint process relies heavily on */proc* file system. CRIU takes all the information it needs such as files descriptors information, memory maps, etc. from */proc*. CRIU firstly collects process (Docker container) tree and freezes it. It walks through */proc/$pid* collecting recursively threads and tasks. Secondly, it writes all the information about the collected tasks and writes them to dump files. Thirdly, after everything is dumped such as memory pages, ptrace utility is used to drop out all parasite codes that was injected in the tasks during the collection process.

3.3.2.2 Restore

CRIU reads the collection of the dumped files or images, restores all the processes and run them as they were during the time of freeze. We relied on this mechanism to effectuate Docker containers live migration.

3.4 ElasticDocker Approach

3.4.1 System design

We designed ElasticDocker to automatically scale up/down Docker containers to adjust resources to the varying workload. ElasticDocker provisions and deprovisions Docker resources vertically on the host machine. As shown in Figure 3.2, ElasticDocker consists of monitoring system and elasticity controllers. Elasticity controllers can adjust memory, vCPU cores, and CPU fraction according to the workload demand. These components are presented in the next sub sections. ElasticDocker adheres to use the well-known autonomic MAPE-K loop [210]. MAPE-K is an autonomic computing architecture introduced by IBM, it consists of five components: Monitor, Analyze, Plan, Execute, and Knowledge. In our system, firstly, different Docker metrics are continuously monitored. In the second analysis phase, thresholds are calculated based on the monitored metrics, then the decision and plan to scale up/down is taken accordingly. Finally, we implement the decisions to adjust Docker resources according to the need.

3.4.2 Monitoring system

Our monitoring system collects most resource utilization and limits of Docker containers by interrogating directly with Docker cgroup subsystems while it uses Docker RESTful API to check CPU usage. The system continuously monitors the memory subsystem in cgroup to check the memory current size assigned to each Docker container as well the current memory utilization. Similarly, we check the CPU parameters such as the number of vCores and time. In Section 3.4.3, we highlight how to control the CPU time, thus allowing us to control CPU percentage assigned to each Docker container. In the experimentation, we have noticed that the CPU and memory utilization values are sometimes fluctuating rapidly, which could be due to the nature of workload. Therefore, to avoid this oscillation, we measure CPU and memory utilization each 4 seconds on an

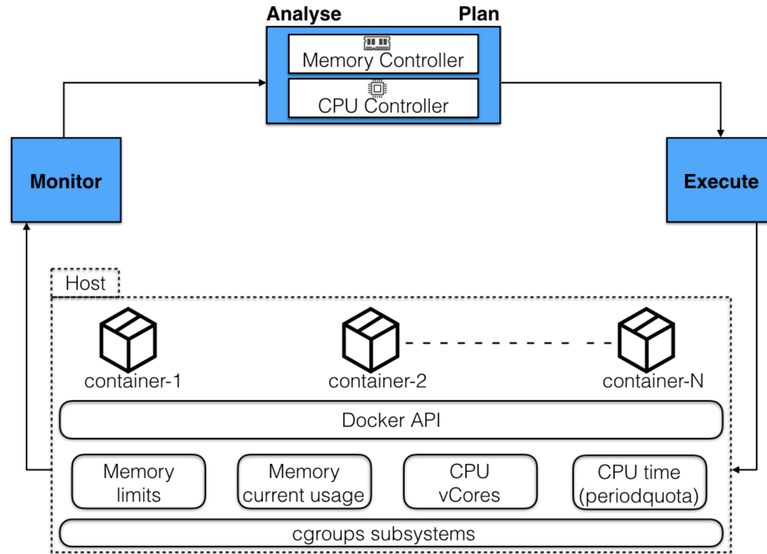


Figure 3.2 ElasticDocker architecture

interval of 16 seconds (as shown in Table 3.1), then we take the average value as the current utilization of CPU and memory.

3.4.3 Elasticity controller

The elastic controller adjusts memory, CPU time, vCPU cores according to workloads. ElasticDocker modifies directly the cgroups filesystem of Docker containers to implement scaling up/down decisions. The memory is monitored and then based on its usage and thresholds, ElasticDocker increases or decreases its size. The upper threshold is set to 90%, and the lower threshold is set to 70%. The values shown in Table 3.1 are chosen following [127], [78] which are based on real-world best practices, in addition we tried different values, and selecting the best values that lead to less response time. Once the memory utilization is greater than the upper threshold, ElasticDocker adds 256MB to its size. In the deprovisioning state, the memory size is decreased by 128MB. We decrease memory size by small amount in the scaling down process because the applications are sensitive to the memory resource, and this could lead to interrupt the functionality of the application. In addition, after each scaling decision, ElasticDocker waits a specific period of time (breath duration). Breath duration is a period of time left to give the system a chance to reach a stable state after each scaling decision. As shown in Table 3.1, we set two breath durations, breath-up and breath-down. Breath-up

is time to wait after each scaling up decision. We chose these small values because the application adapts quickly to the container change, we have noticed that the application functions normally after these time periods. Breath-up is smaller than breath-down to allow the system to scale-up rapidly to cope with burst workload. Breath-down is larger than breath-up duration in order to avoid uncertain scaling down, which could cause degradation in the performance of the system.

Table 3.1 ElasticDocker parameters

Parameter or metric	value
monitored metrics	CPU utilization, CPU time, vCPUs, Memory utilization, Memory limit
measurement period/interval	4 seconds/16 seconds
breath-up/breath-down	10 seconds/20 seconds
upper threshold	90%
lower threshold	70%
CPU increase/decrease ratios	±10% of CPU time or ±1 vCPUs
mem. increase/decrease ratios	+256MB/-128MB

ElasticDocker also controls CPU time (percentage) and number of vCPUs assigned to each Docker container. As we have seen in Section 3.3, we can control CPU time by changing CFS parameters, namely *cpu.cfs_period_us* and *cpu.cfs_quota_us*, we refer to them simply as period and quota. For example if period is set to 100000 and quota set to 10000, Docker can use 10% of CPU percentage (i.e, 0.01 second of each 0.1 second), if a Docker container has two vCPUs and $period = 100000$ and $quota = 200000$, this means the Docker container can completely use the two vCPUs. ElasticDocker increases quota or CPU percentage in function of CPU usage and dynamic thresholds. For example, if a container has 10% of CPU time, the threshold will be 9.5, 20% of CPU time, threshold will be 19% and so on. Once a container has used all the CPU time, new core will be added. Upper threshold to add a vCPU core is 90%. Lower threshold is set to 70%, if CPU usage is less than lower threshold, vcores will be removed. However let's suppose that a Docker container has three vCPUs cores and $quota/period = 250000/100000$ and CPU usage is less than 70%, the scaling down decision is taken according to the following condition: $cpu_usage < 70\%$ and $no_vCPUs > 1$ and $quota < period * (no_vCPUs - 1)$, where no_vCPUs is the number of vCPUs allocated to the container. Similar to memory, breath durations are set for CPU resizing. It is worth noting that ElasticDocker takes in consideration the available resources on the host machine, and the allocated resources of other Docker containers on the host upon each scaling up/down decision.

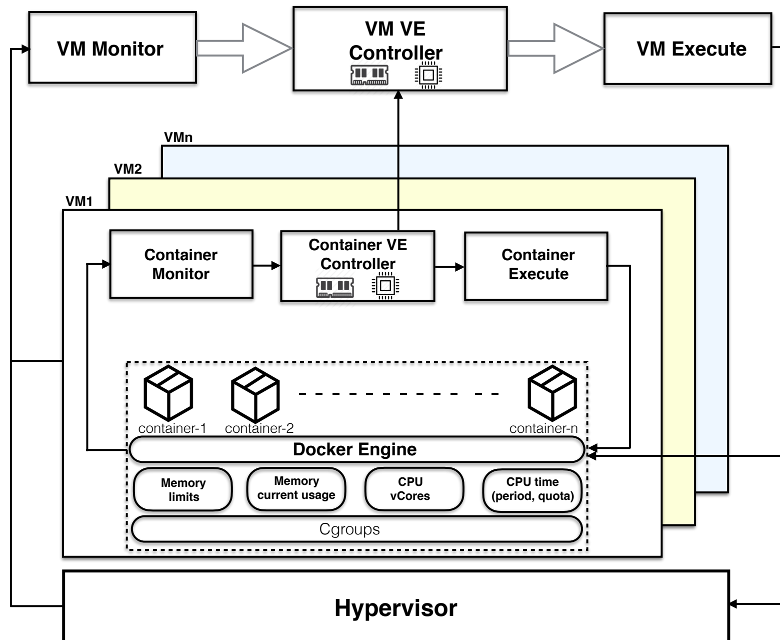


Figure 3.3 Coordinated elastic controllers between VMs and containers

3.5 Coordinated Controller

3.5.1 General Design

Similar to ElasticDocker, the coordinated controller adheres to the control loop principles of the IBM's MAPE-K reference model [210]. The control part of MAPE-K consists of many phases: Monitor, Analyze, Plan, and Execute. The managed components in this context are the infrastructure units VMs and Docker containers, the containerized applications as well. We design elastic controller to automatically adjust resources to the varying workload without violating QoS by growing or shrinking the amount of resources quickly on demand for both containers and their VMs. Figure 3.3 shows the general architecture of our controllers. The architecture design include ElasticDocker (the controller described in Section 3.4) and another one for the hosting machine. The aim for the second controller is to allocate/de-allocate resources if containers residing on a virtual host machine require more/less resources than the amount of resources offered by that VM.

Table 3.2 System control parameters

Parameters	Docker containers	VMs
Upper threshold	90%	90%
Lower threshold	70%	70%
Period	4 sec	1 min
Interval	16 sec	1 min
Breath-up/down	10/20 sec	20/40 sec
CPU adaptation	±1 vCPU	±1 vCPU
Memory adaptation	-128/+256	-256/+512

3.5.2 Components of the System

3.5.2.1 Monitoring Component

The monitoring component consists of two sub monitoring systems (i) the sub monitoring system for Docker containers as explained in Section 3.4 and (ii) sub monitoring system for the VMs. The host machine resource utilization and the amount of acquired resources are monitored periodically as shown in Table 3.2. The elasticity VM controller will use this data to provision/de-provision resources on the host machine. Similarly to ElasticDocker, the CPU and memory utilization values are sometimes fluctuating rapidly, which could be due to the nature of workload. Therefore, to avoid these oscillations, we measure CPU and memory utilization periodically on an interval of 16 seconds for containers and one minute for VMs (as shown in Table 3.2), then we take the average values as the current utilization.

3.5.2.2 Docker Controller

Docker controller is ElasticDocker presented in Section 3.4. The elastic Docker controller manages all the containers residing on a virtual machine taking into consideration the available resources on that machine and the already allocated resources to the containers. It is worth noting that each VM has a Docker engine that manages many containers on that machine. Therefore, there is ElasticDocker controller for each VM.

3.5.2.3 VM Controller

If containers allocate all resources on their hosting VM, they could reach an overload point of 100%. At that time the overload could cause errors in the workload execution since there is no free resources to provision. Therefore, our VM controller should intervene before such situation takes place. Likewise Docker containers, the hosting VMs are monitored constantly and then capacity is increased or decreased in relation to the VM reconfiguration policy involved in our VM controller. The VM controller performs vertical elasticity actions based on rules and real-time data captured by the monitoring system. As shown in Table 3.2, the monitoring component monitors the VM resource usage on an interval of one minute. It uses psutil library to get the resource metrics. The controller analyzes these collected data using its reactive model, it triggers its scaling decisions to increase or decrease VM resources, at the same time, it allows Docker engine to detect the new resources by updating cgroups of that Docker daemon. The values to increase/decrease memory, vCPUs are +512MB/-256MB, +1/-1, respectively.

3.5.2.4 Interactions Between Components

As shown in Figure 3.3, the VM controller can trigger elastic actions based on two cases: (i) when the VM resources utilization reaches certain thresholds, (ii) when it receives a demand from the Docker controller to increase or decrease resources. Here, the VM controller can increase resources without receiving a demand from the Docker controller if we suppose that there are other processes running on the VM alongside with containers. When the VM controller adds more vCPUs to the VM, the Docker engine does not detect these resources whether it uses hard or soft limits. Therefore, upon each scaling decision, the VM controller compares the resources on the VM and Docker engine, it then identifies the ids of the newly added vCPUs, then it updates the cgroups of Docker engine. Now, the Docker engine can allocate these resources to containers. The coordination between the controllers is our major concern, we take the below scenario to illustrate a case of such coordination. Suppose that a VM has 3 vCPUs and three containers are deployed where hard limits are set and each container has 1 vCPU. If the first container usage is 100%, and the other two containers are idle 1 vCPU is 100%, 2 vCPUs are idle), the VM controller will try to decrease the vCPUs, but if it decreases the vCPUs, this will lead to destroy the container whose vCPU is withdrawn. Therefore, the coordination will prevent the VM controller to scale down, and the Docker controller will demand the VM controller to allocate more resources in order to give the first container more resources.

3.6 Container live migration

Many containers generally reside on the same host machine. Therefore, when one Docker container continues to ask for more resources, if there is no more resources on the host, live migration will take place for that Docker container. The container will be migrated to another host machine. The process of live migration consists of four main steps as shown in Figure 3.4. Firstly, the filesystem differences of the container image layers in */aufs/diff/* will be transferred. There are many directories in */aufs/diff/* representing image layers, so we tar and send these layers to the destination host. Secondly, the container process will be pre-dumped. The container is still running after the pre-dump. The objective of the pre-dump is to minimize the migration downtime. The pre-dumped images are compressed using LZ4 compression in a TAR file and sent to the destination. We perform several pre-dump iterations, each pre-dump generates a set of pre-dump images, which contain memory changes after previous pre-dump. This reduces the amount of data dumped on the dump stage. Thirdly, we proceed to dump the container state, the dump process will be rapid because it only takes the memory that has changed after the last pre-dump. On the destination host, we will restore the container to the same memory state on the source host thank to CRIU. Before the restore mechanism, the destination will untar and decompress the images received in order to prepare them for the restoring process.

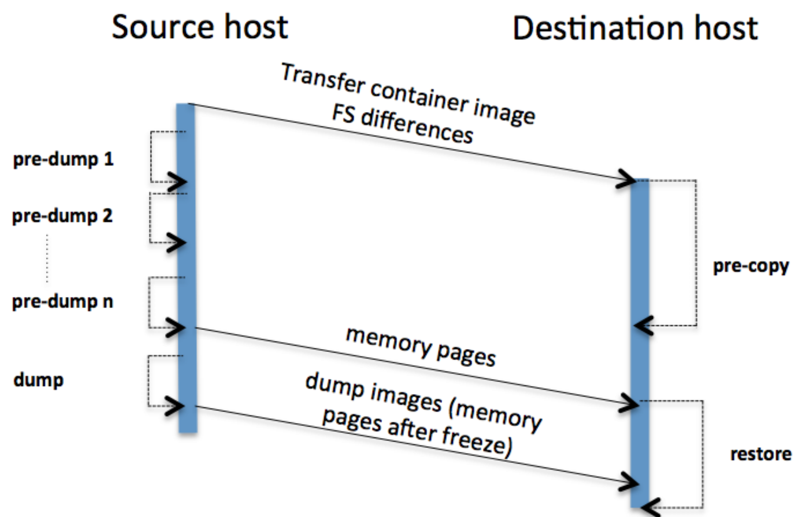


Figure 3.4 Migration procedure based on CRIU

3.7 Discussions

In this section, we discuss our approach and areas of improvements. This chapter has described ElasticDocker, a system that permits to automatically scale up/down resources of Docker containers according to workload changes. The coordinating controller manages the vertical elasticity between containers and their hosting machine. The system also uses live migration to migrate containers if there is no more resources on the actual host. Our system uses reactive approach based on the threshold-based rules to perform elastic actions. While threshold based rules are the most popular auto-scaling technique, even in commercial systems, setting-up suitable thresholds is very tricky, and may lead to instability of the system. Therefore, in the next chapter of our experimentation, we have tried different thresholds, i.e., 90, 85, 80, 70, 60 and different breath or cooling durations. After that, we have chosen the best values as shown in Table 3.1, which yields to best performance (lower response time). The ideal would be to use machine learning to discover the appropriate values. The improvement of QoE by ElasticDocker is not without cost. In fact, ElasticDocker allocates more resources to overcome the workload burst. The system proposed allows to control CPU and memory. Docker allows to control the numbers of operations per second (ops) or amount of data, bits per second (bps) on specific devices connected to Docker container. However, there is no direct method particularly in AUFS filesystem to adjust quota or amount of disk available to a specific container. So, it is difficult to resize the disk storage at runtime. Although it is true that Docker provides the option *–storage-opt* to resize storage, this option is applied to the daemon level not to a single container. For the migration, we simply migrate the container which requires more resources when there is no sufficient resources to reply its demand. The idea is to improve this mechanism in order to have more intelligent system that decides which container to migrate: the one which currently requires more resources, or the one which has less activity, etc. In this chapter, we handled the problem of provisioning and de-provisioning by allocating/deallocating the required resources in a time. The rapid scaling oscillation and system instability of the system are avoided by setting up different breath durations after elasticity decisions. With the adoption of Docker containers and the vertical elasticity, the elasticity actions are performed very quickly and thus eliminates the burden of provisioning or start-up time. As illustrated in this chapter, we investigated many elasticity methods including vertical elasticity, live migration and combined elasticity using different virtualization technologies, VMs and containers.

3.8 Related Work

Elasticity is a major research challenge in the field of cloud computing. Several different approaches have been proposed for the elasticity at VMs level such as [114], [130], [127]. However, with the appearance of Docker containers and their widespread popularity among cloud providers, some researches are dedicated to this field. Kukade et al. [39] proposed a design for developing and deploying microservices-based applications into Docker containers. The elasticity is achieved by constantly monitoring memory load and number of requests by an external master. Once certain thresholds are reached, the master node invokes scaling agent. The scaling agent permits to horizontally spin in or out the container instances. Haldy et al. [37] worked on container live migration technique and proposed a framework called MultiBox. MultiBox is a mean for creating and migrating containers among different cloud providers. It makes use of Linux cgroups to create containers and migrate the source containers to those newly created ones. Hoenisch et al. [36] proposed a control architecture that adjusts VMs and containers provisioning. DoCloud [38] is a horizontal elastic cloud platform based on Docker. It permits to add or remove Docker containers to adapt Web application resource requirements. In DoCloud, a hybrid elasticity controller is proposed that uses proactive and reactive model to scale out and proactive model to scale down. Monsalve et al. [214] proposed an approach that controls CPU shares of a container, this approach uses CFS scheduling mode. Nowadays, Docker can use all the CPU shares if there is not concurrency by other containers. Paraiso et al. [7] proposed a tool to ensure the deployability and the management of Docker containers. It allows synchronization between the designed containers and those deployed. In addition, it allows to manually decrease and increase the size of container resource. These works either handle horizontal elasticity or manual vertical elasticity. [78] proposed horizontal and vertical autoscaling technique based on a discrete-time feedback controller for VMs and containers. This approach is limited to Web applications. In addition, the application requirements and metadata must be precisely defined to enable the system to work. It also adds overhead by inserting agents for each container and VM. Kubernetes and Docker Swarm are orchestration tools that permit container horizontal elasticity, they allow also to set limit on containers during their initial creation. Our proposed approach supports automatic vertical elasticity for Docker containers and live migration if there is no enough resources.

3.9 Conclusion

ElasticDocker is an elasticity controller to dynamically grow or shrink Docker resources according to workloads. An extension, yet a novel coordinated vertical elasticity controller for both VMs and containers is then proposed. It allows fine-grained adaptation and coordination of resources for both containers and their hosting VMs. If there is no more resources on the host machine, we migrate the container to another host. This migration technique is based on CRIU functionality in Linux systems.

The next chapter will extensively evaluate the Elastic Docker approach.

Chapter 4

ElasticDocker Evaluation

In this chapter, we present a wide range of experimentations to evaluate ElasticDocker*.* ElasticDocker *is evaluated with respect to performance, cost and resource utilization. We then evaluate the coordinating controller, the new elasticity controller that coordinates vertical elasticity at both containers and VMs levels. The proposed controller is evaluated with respect to four research questions. After that, we verify the efficiency of our proposed live migration technique.*

4.1 Introduction

Elasticity has different purposes such as improving performance, increasing capacity, saving energy, reducing costs, etc. In this chapter we evaluated ElasticDocker with respect to response time, cost and resource utilization. The results show that ElasticDocker helps to reduce expenses for container customers, make better resource utilization for container providers, and improve Quality of Experience for application end-users.

This chapter is derived from:

- ***Yahya Al-Dhuraibi****, Fawaz Paraiso, Nabil Djarallah and Philippe Merle, "Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER," 2017 IEEE 10th International Conference on Cloud Computing (CLOUD), Honolulu, CA, 2017, pp. 472-479, doi: 10.1109/CLOUD.2017.67*
- ***Yahya Al-Dhuraibi****, Faiez Zalila, Nabil Djarallah, Philippe Merle. Coordinating Vertical Elasticity of both Containers and Virtual Machines. 8th International Conference on Cloud Computing and Services Science - CLOSER 2018, Mar 2018, Funchal, Madeira, Portugal. 2018.*

Towards investigating all the possible approaches of elasticity, we proposed the first approach - coordinated controller - to coordinate vertical elasticity of both VMs and containers. We evaluate the new proposed extension and the results show that our approach reacts quickly and improves application performance. Our coordinated elastic controller outperforms container vertical elasticity controller by 18.34% and VM vertical elasticity controller by 70%. It also outperforms container horizontal elasticity by 39.6%.

The live migration technique implemented in our approach is also evaluated. Based on the observed migration performance metrics, the experiments reveal a high efficient live migration technique.

Throughout the evaluation in this chapter, we have used different real applications (Graylog[1], RUBiS [177]), different VM virtualization technologies (VMware, KVM), different workload generators (httperf, ab) and different container technologies (Docker, Kubernetes). The application components and settings will be described in the next sections.

This chapter is organized around two main sections: Section 4.2 presents intensively the experiments and evaluation of ElasticDocker described in Chapter 3. Section 4.3 presents the experiments and evaluation for the coordinated controller. Section 6.4 concludes the chapter and highlights the final results.

.

4.2 ElasticDocker Experiments and Evaluation

4.2.1 Experimental setup

ElasticDocker approach is evaluated with respect to the performance and end-user QoE, customer cost, resource utilization, migration efficiency. We performed all our experiments on Scalair[2] infrastructure inside VMs. Scalair is a private cloud provider company. The VM on which containers run has 7vCPUs with 5GB RAM and Centos 7.2 OS. We use Graylog, a powerful log management and analysis platform. We chose this application because it consumes a lot of resources. Since Graylog centralizes and aggregates all log files from different sources, it can suddenly get overloaded, and that

[1] https://www.graylog.org
[2] http://www.scalair.fr

requires a lot of attention from the providers to adjust resources according to the need particularly at peak times. Graylog is widely implemented in the industry and it is based on four main components Graylog Server, Elasticsearch, MongoDB and Web Interface. Graylog Server is a worker that receives and processes messages, and communicates with all other components. Its performance is CPU dependent. Elasticsearch is a search server to store all of the logs/messages. It depends on the RAM, disk and also CPU. MongoDB stores read-only configuration and metadata and does not experience much load. Web Interface is the user interface. We run three containers, the first one is for the Graylog server version 2.0.0 and Web interface 2.0.0 while the second and third are for Elasticsearch version 2.3.3 and MongoDB version 3.2.6 respectively as shown in Figure 4.1. We use Docker version 1.9.1 and Docker Compose version 1.7.1. Docker Compose is used to define and run the different components of Graylog in the containers. We also installed Ubuntu 14.04 and Httperf[3] version 0.9.0-2build1 on the second VM. It has 2vCPUs with 4GB RAM. We also set scripts to send log messages and overload Graylog server on this VM. The two VMs are on different VLANs. httperf generates requests to query the Graylog server.

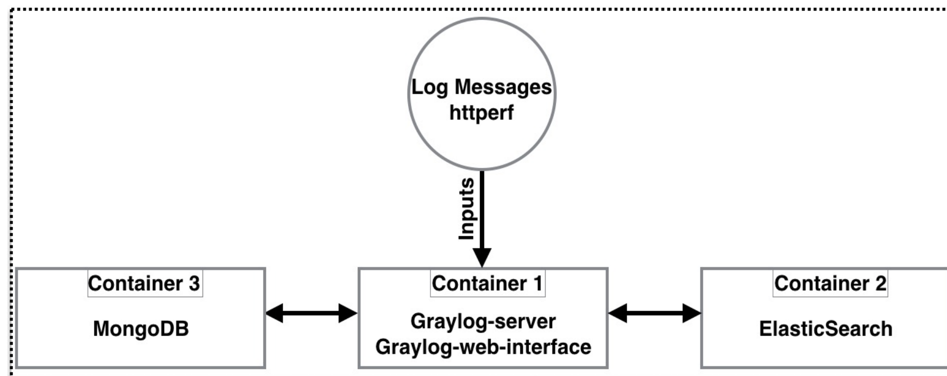


Figure 4.1 Graylog components

4.2.2 Evaluation and results

4.2.2.1 Performance and end-users QoE

First, we investigate the impact of our proposed approach on the performance and end-user QoE and compare the results between Docker and ElasticDocker. Therefore, we run our experiments to evaluate the performance of the Graylog application in two

[3] https://en.wikipedia.org/wiki/Httperf

cases: i) with Docker only, and ii) with ElasticDocker system. We generate different workloads using httperf to query and search information from Graylog via its REST API. Figure 4.2 shows the results of comparison experiments of average response time (RT) between Docker and ElasticDocker. As shown in Figure 4.2, we have different request rates 10 req./sec., 50 req/sec., etc. The more the number of requests are, the more the RT increases. When the number of requests are between 10 and 50 requests per second, there is no significant difference in average RT between the two cases. However, when the number of requests increases and requires more resources, ElasticDocker reacts to provision resources accordingly, therefore the RT decreases and the performance increases. The blue and red bars in Figure 4.2 indicate Docker and ElasticDocker performances, respectively. ElasticDocker increases performance by 74,56%.

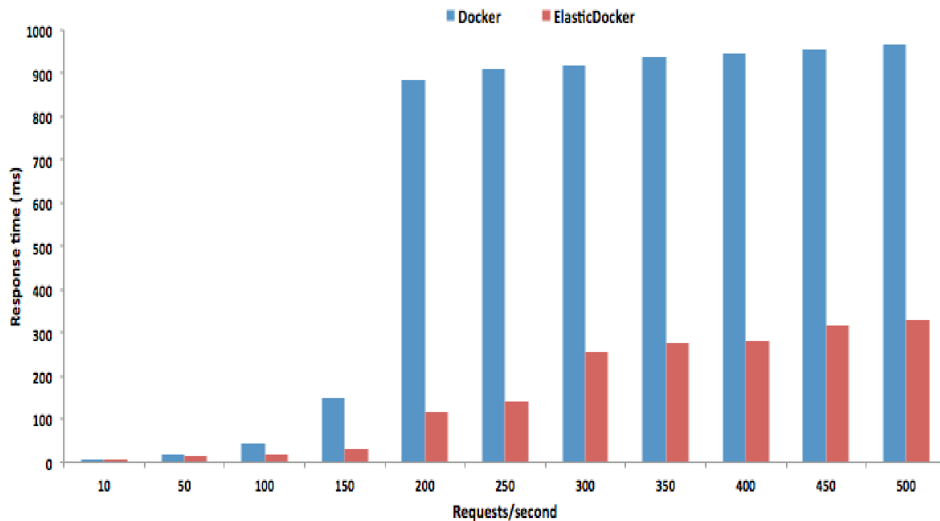


Figure 4.2 Graylog response time with Docker vs ElasticDocker

4.2.2.2 Customers' expenses reduction

In this part of experiment, we study the impact of ElasticDocker on the financial cost. To put stress on Graylog and ElasticDocker containers, we generated workloads with a random rates that arrive to more than 1000 req./sec. by scripts on the second VM. The workloads are syslog and Graylog Extended Log Format (GELF) logs. The logs generated sent to be processed by Graylog server and stored in the Elasticsearch container. At the beginning, each Graylog, Elasticsearch and MongoDB Docker has 1vCPU and 1130MB, 384MB and 128MB respectively. Figure 4.3 shows the vCPU and memory size for each Docker over time. Graylog and Elasticsearch containers consume

CPU and memory while MongoDB has only 1vCPU and 128MB of RAM because it just stores metadata. To facilitate the understanding of this experiment, let us consider the following simplified pricing model used by cloud providers:

$$cost = \sum_{n=1}^{n} cpu(t_n, t_{n-1}) * (t_n, t_{n-1}) * p + mem(t_n, t_{n-1}) * (t_n, t_{n-1}) * p\prime \tag{4.1}$$

where $cpu(t_n, t_{n-1})$ is the number of vCPUs in a time period between (t_n, t_{n-1}), p is the price for each vCPU in time period (t_n, t_{n-1}), $mem(t_n, t_{n-1})$ is the memory size in a time period (t_n, t_{n-1}), $p\prime$ price for the memory. To ease the understanding of the customer costs, let us consider the vCPU consumption of Graylog containers, referring to Figure 4.3 and Table 4.1, the cost according to Equation(1) is $cost = 1*(t1,t0)*p + 2(t2,t1)*p + 3(t3,t2)*p + 4(t4,t3)*p + 3(t5,t4)*p + 2(t6,t5)*p + 1(t7,t6)*p + 2(t7,t8)*p + 1(tx,t8)*p = 28,63p$

Without ElasticDocker, the customer will reserve fixed resources all the time, in our case, it could be 4vCPUs for graylog server and then the cost will be $4*(t8,t0)*p = 66,04p$, (from Figure 4.3, the time periods (t_n, t_{n-1}) are translated against a fixed interval). From these results, ElasticDocker reduces financial cost by 56.65%. It is shown that ElasticDocker significantly decreases the charge for the customers.

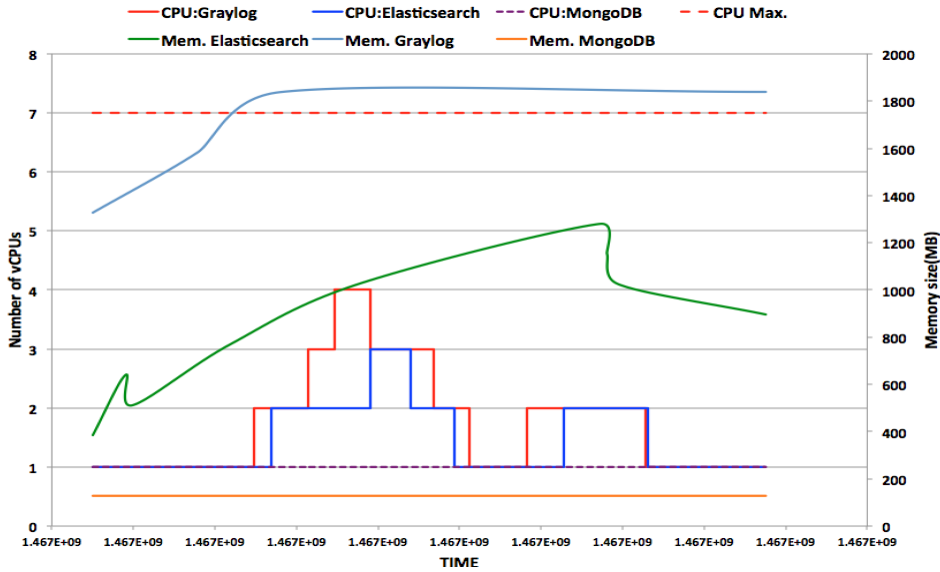


Figure 4.3 CPU and memory consumption of Graylog, Elasticsearch and MongoDB containers

Table 4.1 vCPUs vs. time for Docker container 1

Time	t0	t1	t2	t3	t4	t5	t6	t7	t8
number of vCPUs	1	2	3	4	3	2	1	2	1

4.2.2.3 Optimal utilization for resources

We evaluated the resource utilization in a single host. Figure 4.4 shows that ElasticDocker can maintain a better utilization of resources. Without ElasticDocker, the resources reserved while they are idle. For example in our experiment, to avoid services interruption in Graylog container, 4vCPU must be reserved, however the need for these vcores is for small period only, after that they are idle and would not be possible to run three containers on the same host. ElasticDocker reserves and frees resources according to the charge.

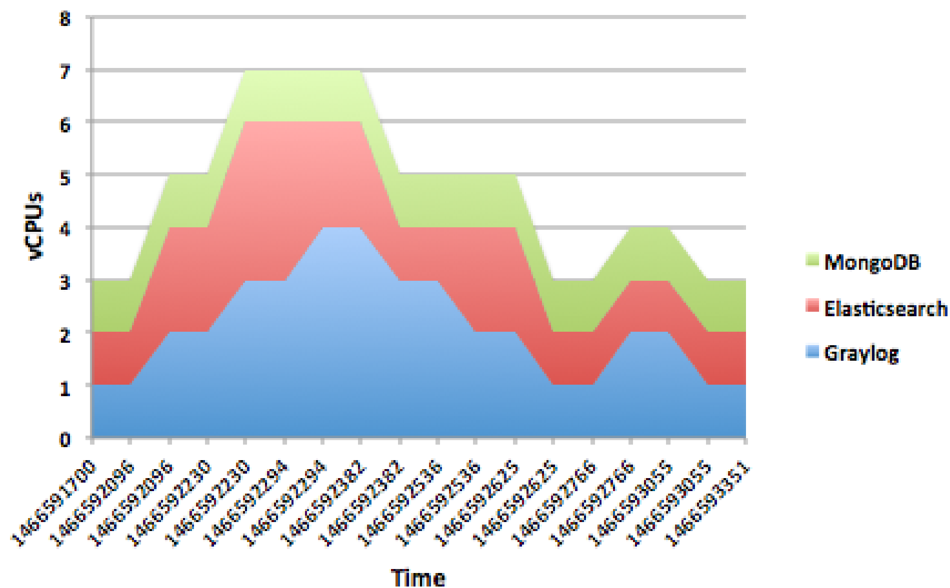


Figure 4.4 Resource utilization

4.3 Coordinated controller validation

4.3.1 Experimental Setup

We evaluated our work using RUBiS [177], a well-known Internet application that has been modeled after the internet auction website eBay. The auction site benchmark

implements the core functionality of an auction site: selling, browsing and bidding. RUBiS is a widely used in a large number of studies and researches to test the scalability and performance of cloud applications and servers. Our deployment of RUBiS uses two tiers: application tier, a scalable pool of JBoss application servers that run Java Servlets, and a MySQL database to store users and their transactions. We performed all our experiments on Scalair infrastructure. We developed the experiments using the following technologies: (a) KVM version 1.5.3-105.el7_2.7 (x86_64), libvirt version 1.2.17, virt-manager 1.2.1, the number of VMs used and their characteristics will be described in the specific experiment subsections because we have used different configurations based on the objective of the experiment. (b) VMWare VCenter version 6.0. (c) Docker engine version 17.04.0-ce. (d) Kubernetes v1.5.2 [215], the Kubernetes cluster consists of 3 machines. (e) ab (Apache HTTP server benchmarking tool) version 2.3 to generate workloads. The hardware specifications consist of 4 powerful servers: 2 HP ProLiant DL380 G7 (Intel(R) Xeon(R) CPU X5650 @ 2.67GHz, 84 GB, 6 NICS), and 2 HP Proliant XL170r Gen9. (Intel(R) Xeon(R) CPU E5-2660 @ 2.60GHz, 100 GB, 8 NICS).

4.3.2 Research Questions

The experiments answer the following research questions (RQ):

- *RQ#1:* how can containers automatically use the hot added resources to their hosting VM?
- *RQ#2:* what is the efficiency of performing scaling decisions made by our coordinated controller?
- *RQ#3:* is our coordinated vertical elasticity of both VMs and containers better than vertical elasticity of VM only or vertical elasticity of containers only (i.e., $V_{cont}.V_{vm} > V_{vm} \oplus V_{cont}$)?
- *RQ#4:* is our coordinated vertical elasticity of both VMs and containers better than horizontal elasticity of containers (i.e., $V_{vm}.V_{cont} > H_{cont}$)?

4.3.3 Evaluation Results

We describe each experiment and analyze the results in response to the RQs.

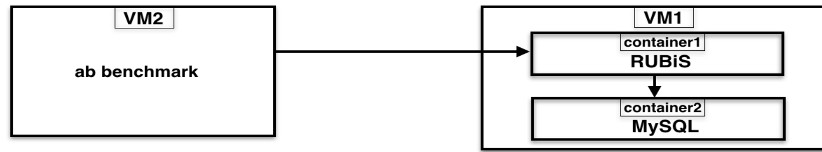


Figure 4.5 Architecture of Experiment 1

RQ#1. In this experiment, we configure two VMs, each with Ubuntu Server 16.04.2 LTS. Initially, VM1 has 2 vCPUs with 2GB of RAM. We deploy RUBiS application inside two containers on VM1 as shown in Figure 4.5. The ab benchmark is installed on VM2, then we generate a workload to the RUBiS application (i.e., 600K requests, concurrency rate 200). The workload requests query RUBiS database to retrieve lists of products, categories, items, etc of the auction website. The difference between workloads is the intensity and concurrency levels. We let the default policy for Docker containers which allow them to use all the available resources. The VM controller is enabled, we register the response time when the workload requests are finished, it was 588.846 seconds. In the second case, we run the same workload, however in this case, we enabled our coordinating controller and we set limits to Docker containers that will be reconfigured by the container controller to accommodate the charge and the response time was 487.4 seconds when the workload is finished. Based on these results, we conclude the following findings:

- The response time is high in the first case because Docker engine does not detect the added resources at the VM level. VM controller has added one vCPU to the VM (the total of CPUs moves to 3 on VM1), however, the two containers used only two CPUs, the third vCPU is idle because containers do not detect automatically the added resources.
- In the second case, the response time becomes smaller, thanks to our coordinated controller which allows containers to demand more resources and subsequently update the Docker engine with the added resources.
- The combined controller augments performance by **20.8%** in this experiment. However if the workload increases, the coordinated controller will accommodate resources in contrary to the first case where the containers can not use more that the initially allocated 2 vCPUs.

RQ#2. In this evaluation, we measure the execution time of elastic actions. Elastic action is the process of adding or removing resources (CPU or memory) to a container, a KVM VM, or a VMware VM. We repeat the experiment eleven times for each resource

Table 4.2 Elastic actions execution performance

Technology	Resource	Action	Repetition	Average execution time (s)	Median (s)	Variance
Containers	CPU	scale up x 15	11	0,003663455	0,00353	2,89474E-07
		scale down x 15	11	0,003192273	0,003345	3,74531E-07
	memory	scale up x 15	11	0,002492073	0,001228	2,95698E-06
		scale down x 15	11	0,001001736	0,0009489	2,88923E-08
KVM VM	CPU	scale up x 15	11	0,282351845	0,2806871	0,003014018
		scale down x 15	11	2,671617281	2,45571805	2,957505433
	memory	scale up x 15	11	0,143514818	0,126101	0,001959036
		scale down x 15	11	0,189079218	0,1609242	0,008245233
VMware VM	CPU	scale up x 15	11	23,9361323	22,599937	27,34976487
		scale down x 15	11	N/A	N/A	N/A
	memory	scale up x 15	11	23,64589671	22,5361431	18,09000088
		scale down x 15	11	N/A	N/A	N/A

(CPU or memory) on each target (i.e., container, KVM VM, VMware VM), and each time the action consists of 15 scaling up or down actions as shown in Table 4.2 (110 experiments in total). During the experiments, the resources experience different stress workloads. We execute elastic actions and we measure the time they take to resize the resource, and then the median and variance is calculated. We take these measures to illustrate the efficiency of our approach to execute auto-scaling actions and to show the differences between the different virtualization units and technologies. We compute the average execution time, median time, and variance for Containers, KVM and VMware VM respetcively as shown Table 4.2.

Based on the these values, we conclude:

- The average execution time is close to median time which indicates that the execution of the elastic actions are stable.
- The elastic actions performed in containers are faster than resizing KVM VM or VMware VM. There is no comparison between containers adaptation and hypervisors, the containers adapt more quickly to the reconfiguration changes while it takes more time to execute scaling actions against hypervisors. The VMware hypervisor managed by VCenter takes more time. High workloads lead to slow execution of elastic actions, particularly in VMware, i.e., the variance is high.

RQ#3. This experiment provides a comparison among vertical elasticity of containers (V_{cont}), vertical elasticity of VMs (V_{vm}), and our proposed approach, coordinating elasticity of both containers and VM ($V_{vm}.V_{cont}$), in terms of performance, i.e., the execution time of workloads and mean response time of concurrent requests. We run three scenarios in this experiment, each scenario has its specific configuration. Five workloads drive each scenario. The workloads consist of a sequence of concurrent requests

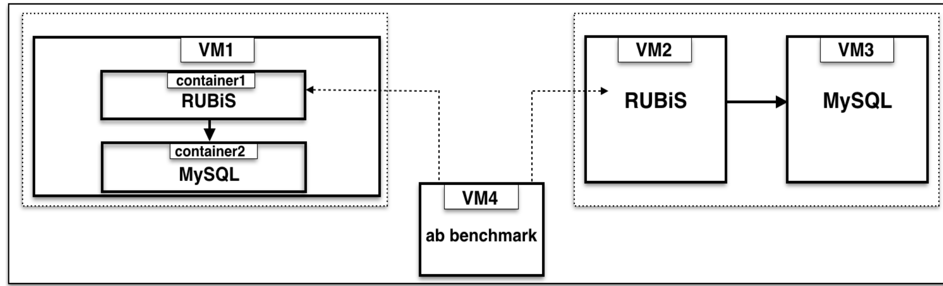


Figure 4.6 Architecture of Experiment 3

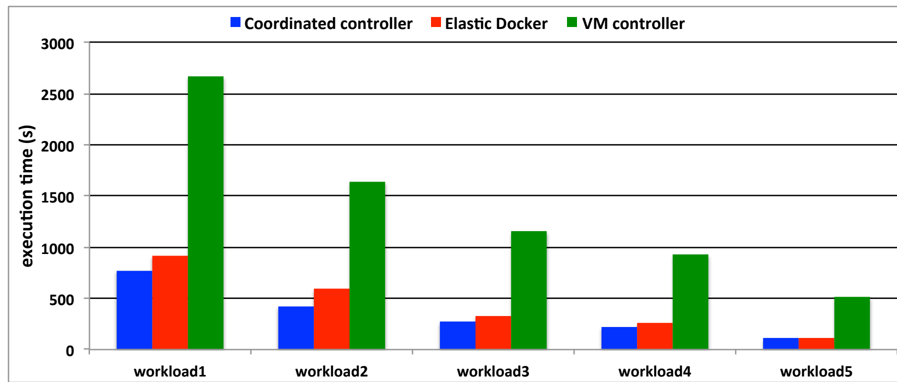


Figure 4.7 Workloads execution time

(the length and concurrency level vary in each workload). In the first scenario (scenario$_1$), as in the topology shown in Figure 4.6, VM1 has 3vCPUs, 2GB of RAM, RUBiS application is deployed on two containers, initially, each Docker container has 1vCPU and 512MB of RAM. We enabled the elastic container controller (which will allow to use the resources available on the hosting VM) and it is named ElasticDocker controller. We measure the total execution time and the mean response time of concurrent requests for each workload as shown in Figure 4.7. In scenario$_2$, we deploy RUBiS application on one VM (VM2) and its database in another VM (VM3). The VMs have 1 vCPU and 1GB of RAM each. We enabled the vertical VM controller to adjust resources according to workload demand and then register the total execution time and the mean response time of concurrent requests for each workload as shown in Figure 4.7. In scenario$_3$, we use the same configuration as in scenario$_1$, except that we enabled our coordinating controller which controls elasticity of containers on the VM, and if there are no enough resources, it will add resources to the VM level.

In Figure 4.7, the red color represents scenario$_1$, the green color represents scenario$_2$ and the blue color represents scenario$_3$. Based on the analysis of this experiment, we concluded the following findings:

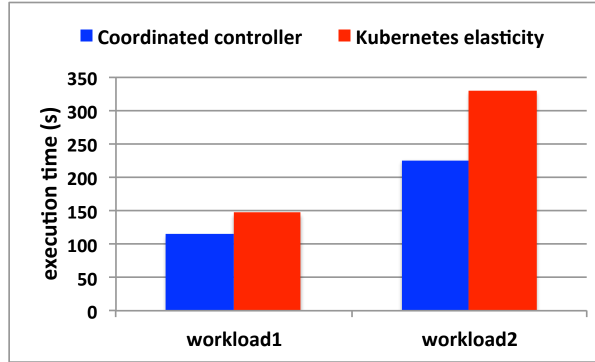


Figure 4.8 workloads total execution time

- In $scenario_1$, the average total execution time, and the mean response time across concurrent requests for the five workloads is **443,7** seconds and **0,91** ms, respectively. Similarly, the average total execution time for the five workloads in $scenario_2$ and $scenario_3$ is **1383,4** seconds and **362,1** seconds, and the mean response time across concurrent requests is **3,1** ms and **0,76** ms, respectively.
- The combined vertical elasticity ($scenario_3$) outperforms the container vertical elasticity ($scenario_1$) by **18.34%** and the VM vertical elasticity ($scenario_2$) by more than **70%**. However, if more workloads are being added to the $scenario_1$, it will not handle them because the available resources will be consumed and performance will be degraded. This demonstrates that the equation $V_{cont}.V_{vm} > V_{vm} \oplus V_{cont}$ is true.

RQ#4. The aim of this experiment is to provide a comparison between horizontal elasticity of containers and our coordinating vertical elasticity of VMs and containers. We use Kubernetes horizontal elasticity. Kubernetes is an open-source system for automating deployment, scaling, and management of containerized applications. To achieve the experiment, we use Kubernetes version v1.5.2. Our deployment of RUBiS on Kubernetes uses three tiers: a loadbalancer (we use Kubernetes service to perform this role), a scalable pool of JBoss application servers, and a MySQL database. Kubernetes platform is deployed on 3 nodes running CentOS Linux 7.2. RUBiS is deployed in two containers, in addition to a load balancer. Then, we set the Kubernetes Horizontal Pod Autoscaling (HPA) to scale RUBiS containers based on rule-based thresholds. We use the same thresholds used in $scenario_3$ in the previous section (in RQ#3). We generate two workloads to both our coordinated controller and Kubernetes cluster. The total execution time across all concurrent requests are measured for each workload as shown in Figure 4.8. According to these results, we conclude the following findings:

- The total execution time for the workloads is **340,66** seconds when our elastic controller is used, while it is **475,558** seconds when Kubernetes HPA is used. The execution time is longer when Kubernetes is used due to the slow Kubernetes integrated monitoring system (Heapster).
- Our combined vertical elasticity outperforms the horizontal elasticity by **39.6%** according to the results of this experiment.
- This proves the equation $V_{vm}.V_{cont} > H_{cont}$ is true.

4.4 Docker live migration efficiency

In this section we migrate different Docker applications from one host to another. We used Docker version 1.9.0-dev and CRIU version 2.2 on both hosts. We evaluate ElasticDocker live migration technique with respect to many parameters such as checkpoint, restore time, etc. We migrate two applications as shown in Table 4.3. The simple workload generator tool (stress) is used. We have checked the application state, for example, we set a counter in the source container and we check the value of the counter once the container is migrated. It shows that the first value on the counter in the migrated container is the value following the last value in the source container. In addition, the container nginx with PHP-FPM pushes incremental counter to a web page, after migration, the operation continues except a suspension for few seconds. Table 4.3 shows different migration indicators. Pre-dump time is the time duration during the

Table 4.3 Migration performance indicators

Application	Image size (MB)	Pre-dump time (s)	Dump time (s)	Restore time (s)	Migration total time (s)	Migration downtime (s)
Nginx	181.5	0.02022	0.2077	3.505	4.28	0.547
Apache	193.3	0.0807	0.196	3.19	5.18	1.712

pre-dump process of the container. Dump time is the time duration during the final dump of the container. Migration and restore times are the periods during the whole process of migration and time to restore the application respectively. Downtime is the time of interruption when the container process is frozen during the final dump process.

Table 4.3 shows the different migration indicators and their values measured during migration of the application containers. The values are small especially the downtime which is the most important in the live migration. Downtime causes a negative impact particularly on stateful applications that are too sensitive for TCP sessions. It is worth noting that there are other factors that could impact the migration such as network

bandwidth. Generally, all migration times in containers only take very short period in comparison to VMs. In this part, we verified this fact by migrating two VMs installed on two different VMware ESXs in the same vCenter cluster using VMware vSphere vMotion. The average time taken during the whole migration process is 14 minutes when migrating both the compute and storage of the VMs while it takes 20 seconds when migrating the compute part only of the VMs. During the migration of these VMs, the hosted applications was operational, however, their performance is impacted.

4.5 Conclusion

This chapter presents evaluation of ElasticDocker approach. Through the experimental evaluations, we have shown that ElasticDocker significantly increases end-user QoE and performance, reduces customer's expenses and makes a better resource utilization. After that, an extensive evaluation of the novel coordinated vertical elasticity controller is performed. The coordinating controller allows fine-grained adaptation and coordination of resources for both containers and their hosting VMs. Experiments demonstrate that: (i) our coordinated vertical elasticity is better than the vertical elasticity of VMs by more than 60% or the vertical elasticity of containers by 18.34%, (ii) our vertical elasticity controller is better than the horizontal elasticity of containers by 39.6%. The experiments demonstrate that fine-grained adaptation capabilities of the proposed approach greatly improve performance when compared to Kubernetes autoscaling. The controller also performs elastic actions efficiently. In addition, the migration downtime is very small.

Part III

Model-Driven Elasticity Management with OCCI (MoDEMO)

Chapter 5

MoDEMO Principles

Elasticity is considered as a fundamental feature of cloud computing where the system capacity can adjust to the current application workloads by provisioning or de-provisioning computing resources automatically and timely. Many studies have been already conducted to elasticity management systems, however, almost all lack to offer a complete modular solution. In this chapter, we propose an approach for modeling Docker containers that provides a tooling to ensure their deployability and management. We then propose MoDEMO*, a new elasticity management system powering both vertical and horizontal elasticities, both VM and container virtualization technologies, multiple cloud providers simultaneously, and various elasticity policies. The proposed Docker model is integrated into* MoDEMO *along with other different infrastructure extensions.* MoDEMO *is characterized by the following features: it represents (i) the first system that manages elasticity using Open Cloud Computing Interface (OCCI) model with respect to the OCCI standard specifications, (ii) the first unified system which combines the functionalities of the worldwide cloud providers: Amazon Web Services (AWS), Microsoft Azure and Google Cloud Platform (GCP), and (iii) allows a dynamic configuration at runtime during the execution of the application.* MoDEMO *permits to timely adapt resource*

This chapter is derived from:

- ***Yahya Al-Dhuraibi***, *Faiez Zalila, Nabil Djarallah and Philippe Merle, "Model-Driven Elasticity Management with OCCI," submitted to IEEE Transactions on Cloud Computing (TCC), June 15 2018, first feedback major revision.*
- *Fawaz Paraiso, Stéphanie Challita,* ***Yahya Al-Dhuraibi*** *and Philippe Merle, "Model-Driven Management of Docker Containers," 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2016, pp. 718-725. doi: 10.1109/CLOUD.2016.0100*

capacity according to the workload intensity and increase application performance without introducing a significant overhead.

5.1 Introduction

Cloud computing has become a preferable solution for deploying applications and services. These applications require a variable amount of computing resources depending on the changing workload intensity at runtime. In addition, due to the heated marketplace competition in the cloud domain, providers have been under pressure to produce attractive services that satisfy customers by maintaining applications performance and respecting the Service Level Agreement (SLA) with optimal costs. Therefore, elasticity is a vital asset as it allows to increase or decrease the capacity of virtual resources timely according to the need [3]. There are two main approaches for elasticity: vertical and horizontal. Vertical elasticity consists of increasing or decreasing characteristics of computing resources, such as CPU cores, memory, network bandwidth, etc. Horizontal elasticity is the process of adding/removing resource instances, which may be located at different locations. Load balancers are used to distribute the load among the different instances. Since elasticity is a key feature in cloud computing, it has been widely explored by many works. For example, the works [114], [130] [4], and [5] address the vertical elasticity, while the works [140] and [78] focus on the horizontal elasticity. Cloud elasticity is diverse and heterogeneous because it encompasses different approaches, policies, purposes, and applications as presented in Chapter 2 [3]. There are different policies that the elasticity controller can use to decide when and how to provision or deprovision the resources. Elasticity also has different purposes such as improving performance, increasing resource capacity, saving energy, reducing cost and ensuring availability. In addition, elasticity can be applied at the infrastructure level or application level. The infrastructure is powered by a certain virtualization technology such as VMware, Xen, or a provider-specific virtualization platform. Container, a lightweight virtualization technology, is also increasingly adopted by the cloud providers. To the best of our knowledge, there is no work that proposes an elasticity management system (EMS) supporting both vertical and horizontal elasticities, both VM and container, multiple cloud providers, and various elasticity policies. The main contribution of this chapter is to present a new elasticity management system called MoDEMO (Model-Driven Elasticity Management with OCCI). In addition, we propose a Docker model to abstract

and manage Docker containers. This model is then integrated into MoDEMO approach. The main characteristics of MoDEMO are:

- **Standard-based**: The existing EMS are proprietary systems, MoDEMO is based on Open Cloud Computing Interface (OCCI) standard.
- **Model-Driven**: Most existing EMS are defined by an API while we propose a high-level model that describes all aspects of elasticity.
- **Both vertical and horizontal elasticities**: The majority of EMS support only one type of elasticity. MoDEMO supports both.
- **Both VM and container**: MoDEMO supports both VM and container virtualization technologies.
- **Multi-cloud providers**: Most existing EMS are dedicated to a particular cloud provider. MoDEMO supports different cloud providers simultaneously.
- **Multiple elasticity policies**: MoDEMO supports different categories of elasticity policies such as scaling, scheduling, migration, and swapping policies. MoDEMO is the first EMS that combines all the elasticity policies of the most popular cloud providers including Microsoft Azure, Amazon Web Services (AWS), and Google Cloud Platform (GCP). Based on the chosen policy, MoDEMO permits to adapt resource capacity and increase application performance.
- **Highly extensible**: New elasticity policies, provider allocation policies, load balancer algorithms, monitored metrics, etc. can be added easily. MoDEMO eases the utilization of the elasticity controller. The elasticity mechanisms and metrics are built with plug in/out facilities.
- **Highly reconfigurable**: MoDEMO is an EMS reconfigurable during the execution. MoDEMO allows elasticity controller settings at runtime through a Model@Runtime approach [216]. Consequently, the elasticity controller applies these configurations immediately.
- **Negligible overhead**: MoDEMO introduces a negligible overhead.

The remainder of this chapter is organized as follows. Section 5.2 describes the motivation for this work. Section 5.3 presents the background on OCCI. Section 5.4 presents Docker model. Section 5.5 presents our MoDEMO system and its different components. Section 5.6 provides a short discussion about this work. After that, we discuss related works in

Section 5.7. Section 5.8 concludes the chapter and summarizes the main aspects of the proposed approach.

5.2 Motivation

This section highlights some motivations towards MoDEMO and Docker model.

5.2.1 Motivation for MoDEMO

- **Standard-based**: There is no EMS that manages cloud elasticity based on a standard. Most of the existing EMS are proprietary systems which introduce vendor lock-in problem while MoDEMO is based on OCCI standard. OCCI is an extremely important paradigm that defines open standard API specifications in the cloud space [217].
- **Model-Driven**: Despite the numerous works involved in elasticity and resource management, there is no model-based framework that can manage all aspects of elasticity. Most existing EMS are defined by an API while we propose a high-level model that describes all aspects of elasticity. Our approach employs the Model-Driven Engineering (MDE) approach in order to handle elasticity resources and controller policies at a higher level of abstraction based on the OCCI standard.
- **Vertical and horizontal elasticities**: The majority of elasticity solutions support only one type of elasticity: horizontal xor vertical. Only a few EMS support both of them [140], [78]. MoDEMO supports both types of elasticity.
- **VMs and containers**: Most works concentrates on VM elasticity and few works address the elasticity of containers [3]. MoDEMO supports both virtualization technologies: VMs and containers.
- **Multi-cloud providers**: Most existing EMS are dedicated to a particular provider. Few works [91], [97] support multiple clouds, however, the chosen provider must be defined manually. MoDEMO supports different providers simultaneously. In addition, MoDEMO has an allocation policy that permits to seamlessly and automatically lease the resource from different providers.
- **Multiple elasticity policies**: Since the elasticity is heterogeneous, it has diverse mechanisms and objectives, each work around elasticity concentrates on a single

approach for a certain purpose. MoDEMO is the first modular approach that gives the possibility to use multiple elasticity policies in a single system. MoDEMO provides the elasticity policies supported by the most popular cloud providers and more. These providers include Microsoft Azure, AWS, and GCP. MoDEMO not only supports many elasticity policies including policies supported by these providers but also provides an improvement for the existing ones. For example, simple dynamic scaling policy in AWS only support one metric while MoDEMO permits combining many metrics in the same policy according to a certain logic. We will explain the different policies and the added value or improvement in the elasticity controller policies in Section 4. Generally, the variability and heterogeneity of the elasticity horizon (techniques, approaches and purposes) are very large. MoDEMO is a unified modular solution for these issues.

- **Highly extensible**: MoDEMO provides a separation in the abstraction between resources and policies. Various policies can be added dynamically. The existing EMS are introduced as a single entity or blackbox. The elasticity controller in MoDEMO is loosely coupled where many components support drag-and-drop (plugins) functionality.
- **Highly reconfigurable**: There are many parameters and settings which determine the behavior of an elasticity controller, *e.g.,* the maximum number of instances in a horizontal group, the memory maximum size or vCPUs of a compute instance. Examples of other parameters are: *coolduration*, *upper threshold*, *lower threshold*, etc. MoDEMO allows changing such settings at runtime. Different policies such as *AllocationPolicy*, *LoadBalancerPolicy*, etc. can be changed dynamically. As discussed in Section 4, a rule is used to evaluate a metric (CPU, memory, etc) which can be modified at runtime and, thus, leads to change the monitoring system and the elasticity policy behavior at runtime.

5.2.2 Motivation for Docker Model

- **Containers elasticity**: MoDEMO manages the elasticity of different virtualization techniques (VMs, containers) by importing different OCCI infrastructure extensions. There is no extension that manages Docker containers. Docker model is a vital component that will be integrated in our MoDEMO approach to allow container elasticity.
- **Lack of verification**: Docker provides tools such as Docker Compose or Docker Swarm used to design a set of containers connected together. However, once

designed, the deployment of the containers can face several problems such as misconfiguration of links between containers, lack of resources on the hosts in which the containers are deployed, human errors, etc. Given the executable mechanism the Docker containers are related, the only way to be sure that the containers deployed will run or fail is to deploy them on the target executing environment. Moreover, there is no way to verify that deployed containers are conform with those designed. The lack of verification tool can become quickly painfully and expensive when the deployment task is repeated several times.

- **Resources management at runtime**: when creating containers, Docker gives the possibility to set the resources (CPU, memory, disk, network) limits. In other words, Docker provides the possibility to set container resources at design time. In the cloud environment, the container resources consumption fluctuates according to their embedded application workload. In order to provision the appropriate resources, if the workload grows or shrinks, the container resources should be increased or decreased as required at runtime. Docker does not provide a mechanism to reconfigure the container resources at runtime.

- **Synchronization between design and execution environment**: In Docker context, the execution environment consists of Docker engine and the containers deployed. Conceptually, the deployed containers represent a predefined architecture. Thus, a major challenge is how to synchronize the predefined architecture of containers with the containers deployed in the execution environment. When modifications occur in an existing architecture, the update should be done in the executing environment. Conversely, when changes occur in the executing environment they should affect the existing architecture. A modification can be addition of a new container, the retrieval of an existing container, the addition of a link between a new container with an existing one, etc.

- **Inconsistency use of containers across organization**: Container adoption has not been a carefully planned and executed by companywide understanding and belief in its virtues [218]. Instead, individuals or small teams of developers have started using containers because they are fast and convenient, enabling them to respond to the increased pressure for quick turnaround coming from their business units and thus making their jobs easier. Each user relies on its familiar tools (e.g, Chef, Puppet, and Ansible) which are used for building and deploying containers. In this context, it remains the problem of maintainability due to heterogeneity of tools used by users.

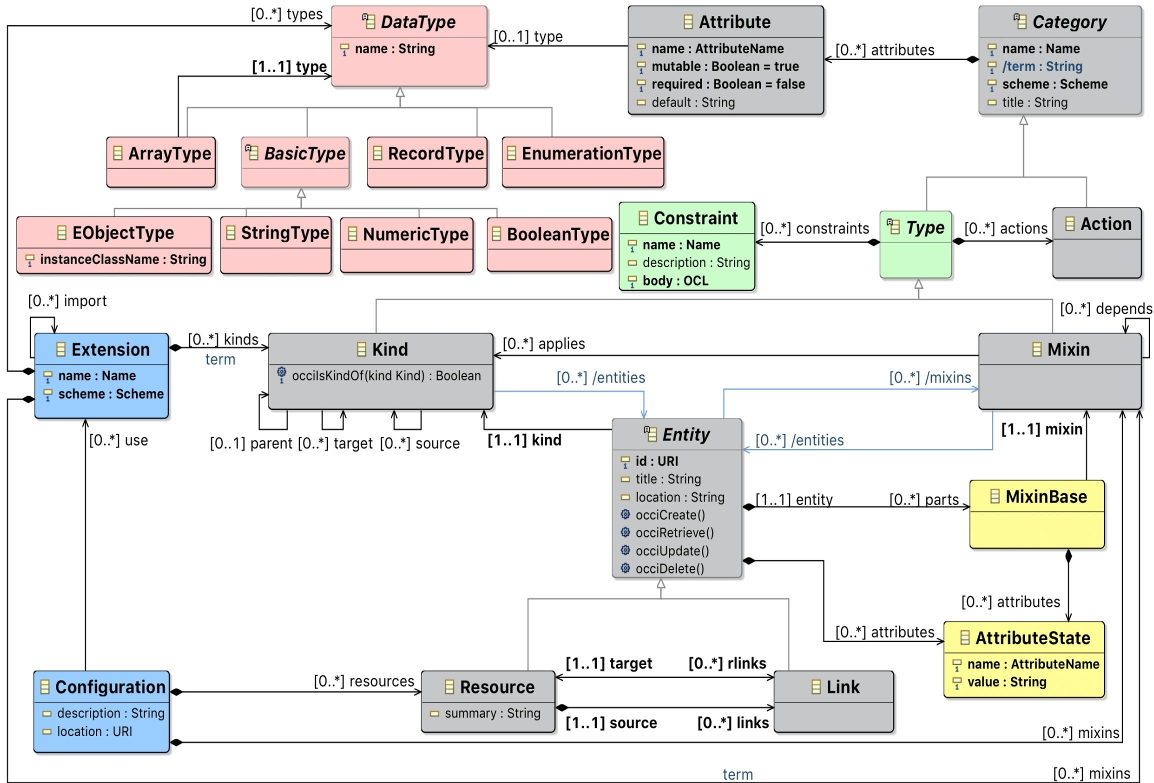


Figure 5.1 OCCIware metamodel

Section 5.4 brings forward a solution to these challenges, using a model-driven approach. This approach will allow Docker technology to have a complementary tool to make it easier of benefiting from containers in production environments.

5.3 Background

A brief description of the OCCI standard and the OCCIware toolchain are provided in this section since MoDEMO and our Docker abstraction are based on OCCI standard and implemented using the OCCIware toolchain. Open Cloud Computing Interface (OCCI) was introduced by Open Grid Forum (OGF) for managing any kind of cloud resources [217]. OCCI is delivered as a set of specification documents divided into the four following categories: OCCI Core Model, OCCI Protocols, OCCI Renderings, and OCCI extensions.

The OCCI core model [219] is composed of eight elements (grey boxes in Figure 5.1): *Resource*, *Link*, *Entity*, *Kind*, *Mixin*, *Action*, *Category*, and *Attribute*. *Resource* is the root abstraction of any cloud resource such as a virtual machine, a database, etc. *Link* represents a relation between two resources, such as a virtual machine connected to a network and an application hosted by a virtual machine. *Entity* is an abstract base class inherited by both *Resource* and *Link*. *Kind* is the concept of type within OCCI such as Compute, Network, and Container. *Mixin* represents a set of attributes and actions that can be dynamically plugged in/out to an OCCI entity. *Mixin* can be applied to zero or more kinds and can depend on zero or more other *Mixin* types. *Action* represents a business-specific management behavior that can be executed on entities such as start/stop a VM. *Category* is an abstract base class inherited by *Kind*, *Mixin*, and *Action*. *Attribute* defines a client-visible property, *e.g.*, the IP address of a network.

During the OCCIware project [220], a precise metamodel of OCCI has been proposed, named OCCIware metamodel [221]. All the concepts of the OCCIware Metamodel are introduced and modeled in [222]. In following, we detail a subset of the OCCIware metamodel required to understand the rest of this chapter. *Extension* represents a cloud domain such as infrastructure, platform, etc. *Extension* has a *name* and a *scheme*. It owns a set of *kinds*, *mixins* and *types*, and can import zero or more *extensions*. *Configuration* represents a running OCCI system. It owns zero or more *resources* (and transitively *links*), and use zero or more *extensions*. *DataType* is the root of a data type system for OCCI as shown at the left part of Figure 5.1 (the red-colored classes). This data type system allows us to define primitive types such as *StringType* to model string types, *NumericType* to model numeric types and *BooleanType* to model boolean types. In addition, it allows us to model Java-based types using *EObjectType* and enumerations using *EnumerationType*. It also provides the capability to model complex types like *ArrayType* to design array types and *RecordType* to design structured types. *Type* represents an abstract type inherited by *Mixin* and *Kind*. *Constraint* represents a business invariant related to a specific cloud domain. For example, all IP addresses of all network resources must be distinct. *MixinBase* represents an instantiation of the *Mixin* concept in an entity. *AttributeState* represents an instantiation of the *Attribute* concept.

Based on the OCCIware metamodel, OCCIware Studio [222], the first model-driven tool chain for OCCI, has been developed. It provides a model-driven tooling to design and verify both OCCI extensions and configurations.

5.4 Docker Model

In this section we present our Docker model. We begin by giving an overview of the solution architecture and then we present how we model Docker containers.

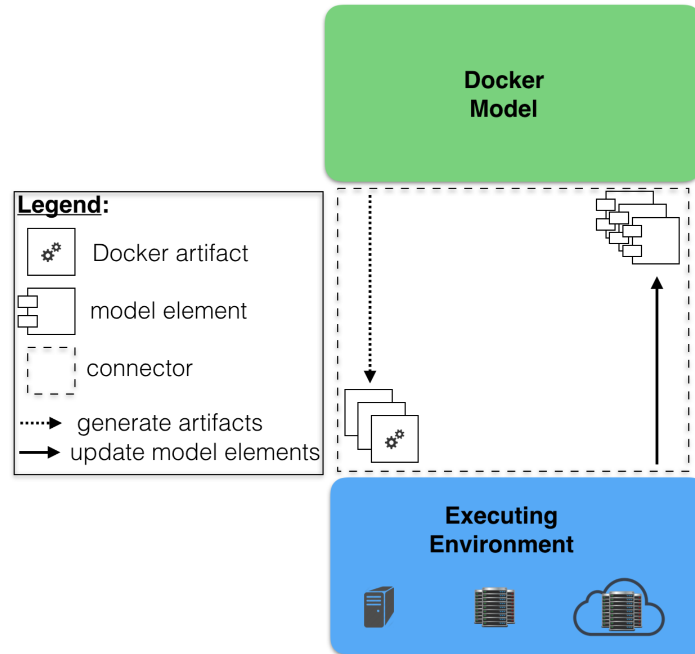


Figure 5.2 Architecture overview.

5.4.1 Architecture overview

To understand the concepts that rely under our architecture, we begin by giving an illustration of it in Figure 5.2. This architecture is composed of three parts: *Docker Model*, *Connector*, and *Executing Environment*. Conceptually, the architecture depicted in Figure 5.2 presents a *Docker Model* which provides expressive model for containers. This model provides an appropriate abstractions of Docker containers (cf. Section 5.4.2 for more details). The *Connector* (cf. Figure 5.2) defines the relationship between the *Docker Model* and *Executing Environment*. This *Connector* provides tools that are used not only to generate necessary Docker artifacts corresponding to the model actions (create, start, stop, restart, pause, unpause, kill), but also to operate efficient and online updating of the *Docker Model* elements according to the changes in *Executing Environment*. Every artifact is handled in a seamless way thanks to the homogeneity provided by modeling principles. Finally, the generated artifacts are executing in the *Executed Environment*.

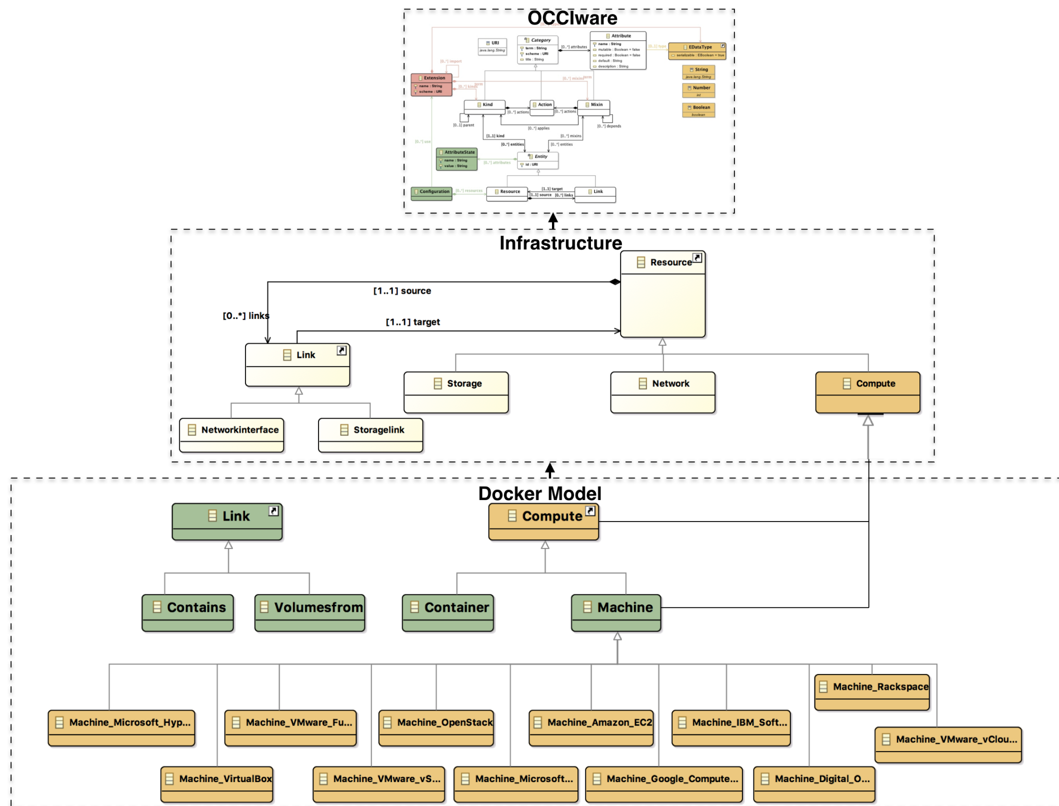


Figure 5.3 Docker model.

Our approach employs Model-Driven Engineering (MDE) techniques [223, 224], in order to handle and analyze Docker Containers at a higher level of abstraction compared to low level the actual Docker solution provides. Using MDE techniques, *Docker Model* describes explicitly certain concerns or certain views on an *Executing Environment* required to face the challenges discussed in Section 5.2.

5.4.2 Modeling Docker Containers

This section describes how the modeling of Docker Containers is achieved. Before we set out to design the *Docker Model*, we investigate to identify the requirements. We begin by examining the Docker Containers with consideration of the main concepts, its structure and the relationship between each other. In this context, our model captures all necessary information related to the characteristics and management of Docker Containers. This model is designed to be compliant with Docker Containers. As depicted by Figure 5.3, our model is conceptually divided into three levels.

The top level represents the **OCCIware**[1] metamodel (see Figure 5.1 for more details). The **OCCIware** metamodel is based on a precise metamodel of Open Cloud Computing Interface (OCCI)[2] [221], an OGF's specification defining an open interface for managing any kind of cloud computing resources (IaaS, PaaS and SaaS). The **OCCIware** metamodel is encoded with the Eclipse Modelling Framework (EMF) [225].

The middle level named **Infrastructure** is based on our **OCCIware** metamodel. The **Infrastructure** model abstracts the Cloud infrastructure resources (i.e, compute, network and storage). The bottom level represents our Docker model, which extends the **Infrastructure** model.

It is worth noting that there are other models such as the VMware model, which extend the **Infrastructure** model. We do not present these models here because they are modeled and developed in the context of the OCCIware project. However, we have imported them into our MoDEMO model (cf. Figure 5.6).

In Figure 5.3, our Docker model is simplified as it does not show, among others, attributes, and enumerations. Based on the **Infrastructure** model, our Docker Model provides a comprehensive view on Docker containers. Building a Docker model means thinking about structure of containers, their relationships with each other, and the hosts inside which there are deployed. In our model, we explicitly provide a rich abstraction for describing, composing, and manipulating structured information related to the containers and the hosts in which there are deployed.

In the following, we present briefly the main concepts of our Docker model:

- *Container* represents a Docker container. *Container* has a set of properties (name, image, command, etc.) related to the Docker container.
- *Link* is a relation between two container instances. *Link* references both source, and target containers, e.g, when containers are linked, information about source container can be sent to target container.
- *Machine* represents any physical or cloud VM that hosts containers. Here, MDE allows to factorize common pattern and reuse them. For instance, the *Machine* class is extended to describe the specificities of targets VM, e.g, *Machine_OpenStack* is an extension of *Machine* used to define the specificities (location, key, type of machine, etc.) of VM belongs to OpenStack.

[1] http://occiware.org
[2] http://occi-wg.org/

- *Volumesfrom* represents a block storage that is attached to one or more container instances to persist data.
- *Contains* is used to define the relationships between a*Machine* and a *Container* instance , e.i, a machine contains zero or more container instances.

The *Docker Model* provides the support for reasoning on architectural constraints of containers. In fact, to analyze architectural constraints, the Object Constraint Language (OCL) and checkers like EMF OCL[3] are used to define and check constraints that are attached to the model elements. For instance, among the constraints defined for the *Docker Model*, one constraint states that a container can not be connected to itself, another one states that bidirectional or closed loop link is not permitted, etc.

Unlike Docker solution, our model uses a constraint validator at **design time** to validate the constraints defined before the deployment. This validation guarantees the coherence of the containers and their relationships with each other.

By allowing the use of constraint validator, our *Docker Model* provides a solution to the **Lack of verification** challenge identified in Section 5.2. On the other hand, the use of an explicit model to represent containers allows us to address the **Inconsistency use of containers across organization** challenge identified in Section 5.2.

5.5 MoDEMO Approach

In this section, we present MoDEMO. We begin by describing the model. We describe the different policies and the model design configuration. We also give an overview of the system architecture. It is worth noting the figures in this section are directly captured from our OCCIware Studio project.

5.5.1 MoDEMO Model

From OCCI perspective, the MoDEMO model defines the following Resource kinds (red colored kinds in Figure 5.4): `HorizontalGroup`, `Provider`,*e.g.*, AmazonProvider, Load Balancer, Compute, Elasticity Controller, Step, Action Trigger, etc. All resource kinds support *CRUD* operations (i.e., `Create`, `Retrieve`, `Update`, and `Delete`). The Compute kind is inherited by different clouds or compute providers. In

[3]https://wiki.eclipse.org/OCL

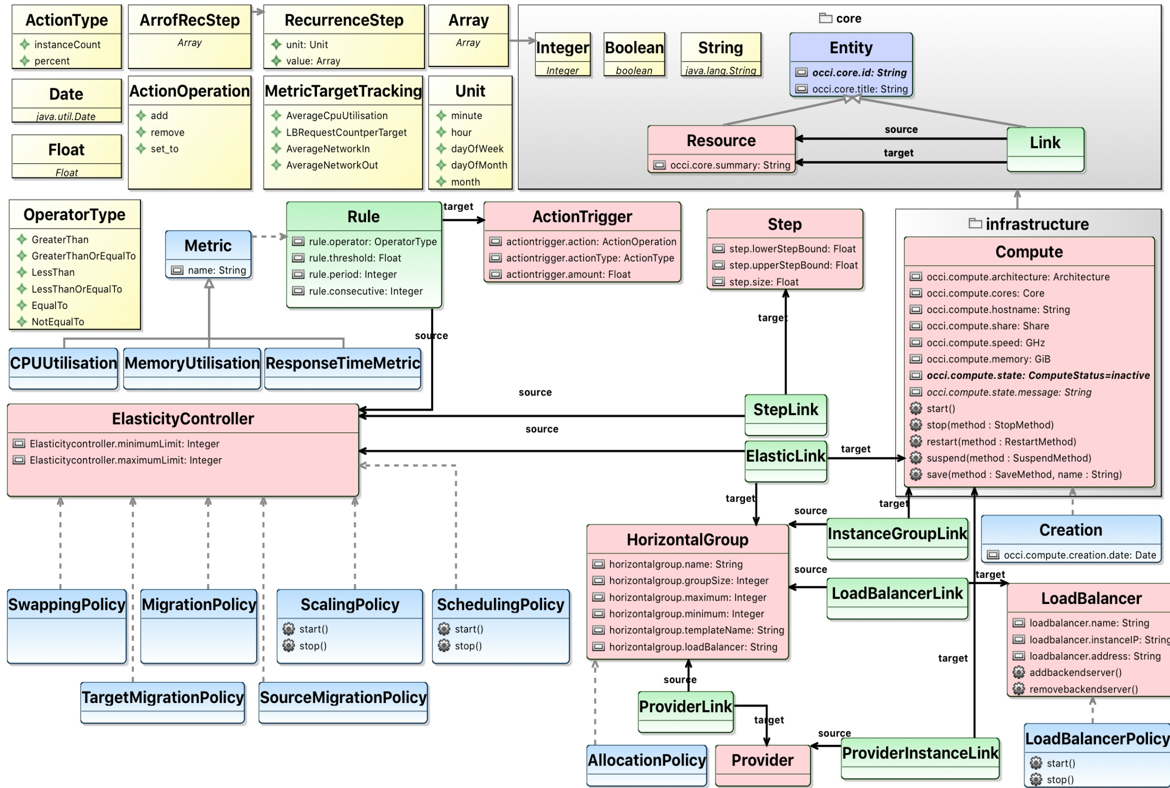


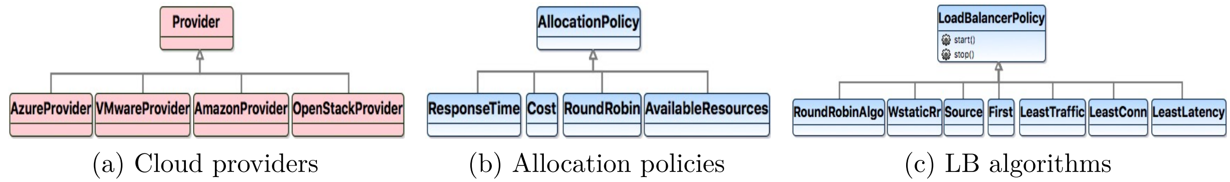
Figure 5.4 MoDEMO model

addition, the model has seven LINK kinds (green colored in Figure 5.4): ELASTICLINK, STEPLINK, RULE, INSTANCEGROUPLINK, LOADBALANCERLINK, PROVIDERLINK, and PROVIDERINSTANCELINK. There are also many *mixins* (blue colored in Figure 5.4) such as METRIC, SCHEDULINGPOLICY, SCALINGPOLICY, ALLOCATIONPOLICY, MIGRATIONTYPE, etc. that can be applied to the other entities (RESOURCE and LINK instances). In the following, we present briefly the main concepts of our MoDEMO model (c.f., Figure 5.4 - all the figures presented in this chapter can be found here[4]).

5.5.1.1 `HorizontalGroup` (HG)

The HG represents a collection of compute instances that share similar characteristics and are treated as a logical unit for the purposes of elasticity and management, *e.g.*, an application that operates across multiple instances. The HG owns a set of `attributes` including a name (`name`), a minimum number of compute instances (`minimum`), a desired

[4]https://github.com/yehia2221/MoDEMO/tree/master/figures



(a) Cloud providers (b) Allocation policies (c) LB algorithms

Figure 5.5 Cloud providers, allocation policies and load balancing algorithms

capacity or number of instances that the HG attempts to maintain (`groupSize`), a maximum size of instances (`maximum`), and a template (`templateName`) or launch configuration which specifies the image or application of the HG compute instances. In addition, the HG is linked to a load balancer (`loadBalancer`) to distribute the traffic among its compute instances.

The instances of a horizontal group can be seamlessly deployed on multiple clouds according to a certain policy. Thanks to the `AllocationPolicy` mixin that can be applied on the horizontal group dynamically.

`AllocationPolicy`
This *mixin* permits to choose the appropriate provider (`Provider`) in order to deploy a new horizontal group compute instance. Our model supports different policies (mixins) as shown in Figure 5.5b:

- `RoundRobin`: This policy selects a provider in turns, starting with the first one from the list of providers in which the horizontal group has links to, until the end of the list is reached at which point the next request (instance provision) will go back to the first provider again.
- `ResponseTime`: The nearest provider will be chosen.
- `Cost`: This mixin will choose the low-cost cloud, the provider which offers the lowest price for deploying the instances.
- `AvailableResources`: This policy chooses the cloud which has sufficient available resources, especially the private cloud based on VMware or OpenStack.

5.5.1.2 `Provider`

The `Provider` kind models a cloud provider entity. The `Provider` provides `Compute` instances, i.e, VMs and containers. As shown in Figure 5.5a, the currently supported providers are:

- `AzureProvider`: Microsoft Azure cloud service.
- `VMwareProvider`: cloud based on VMware technology.
- `AmazonProvider`: Amazon Web Services (AWS).
- `OpenStackProvider`: cloud based on OpenStack.

5.5.1.3 `LoadBalancer` (LB)

The `LoadBalancer` kind represents a LB which is used to distribute traffic among the HG compute instances. The LBs are used for optimizing resource utilization, reducing latency, maximizing throughput and ensuring fault-tolerant configurations. There are some key concepts and terms associated with the LBs such as frontend, backbends, algorithms, etc. The frontend defines where and how the incoming traffic is reached to the machines behind the LB. The backend defines a pool (list) of servers called backend servers that the frontend will forward requests to. LB algorithm is used to determine which server, from the backend list, will be selected when load balancing. As shown in Figure 5.4, the LB has a set of `attributes` and `actions`. The most important actions are `addbackendserver()` and `removebackendserver()` which serve to register/remove the compute instances belonged to a HG in/from the LB. In addition, our model offers a set of mixins (`LoadBalancerPolicy`) as shown in Figure 5.5c, which support the most utilized LB algorithms. Such mixin can be dynamically applied on the LB, each mixin has two operations: `start` the LB algorithm and `stop` the LB algorithm. The supported LB policies are:

- `RoundRobinalgo`: This mixin or Round Robin algorithm [226] selects servers in turns, starting with the first one in a backend until the end of the list is reached at which point the next request will go back to the first server again. The servers list in the backend may be assigned a weight parameter to determine how frequently the server is selected. Therefore, using Round Robin algorithm, each horizontal group compute instance is used in turns according to their weights.
- `WstaticRr`: This mixin policy is a representation of Static Round Robin algorithm [227]. It is similar to the previous algorithm where each server is used in turns, according to their weights. However, it is static, which means that changing a server's weight on the fly will have no effect. Unlike Round Robin algorithm, it has no design limitation on the number of back-end servers. Round Robin is limited by design to certain active servers per backend.

- `Source`: Source policy selects which server to use based on a hash of the source IP. The source IP address is hashed and divided by the total weight of the running servers to designate which server will receive the request. The same client IP always reaches the same server in the backend as long as no server goes down or up. This is useful if it is important that a client should connect to a session that is still active after a disconnection.
- `First`: The first server with available connection slots receives the connection. The main goal of the algorithm is to use the least amount of servers. It allows to turn off additional servers in non-intensive hours [228].
- `LeastConn`: Each compute instance in a given backend is evaluated to determine which one has the least number of active connections. The next request will be forwarded to the server which has the least number of active connections. This algorithm takes into consideration the current load of the server when distributing requests.
- `LeastTraffic`: The request will be forwarded to the server which has the least outgoing traffic. This algorithm takes into consideration the output traffic in the network interfaces.
- `LeastLatency`: The request will be forwarded to the server which has the least latency, therefore the reply is sent quickly.

5.5.1.4 `Compute`

The `Compute` kind abstracts computing resources that can be VMware instance (InstanceVMware), OpenStack (oscore instance), AWS EC2 (instanceEC2), GCP instance or Container instance. As shown in Figure 5.4, this part extends the OCCI **Infrastructure** [229] model which abstracts Cloud infrastructure resources (i.e, Compute, Network and Storage). The Compute has a set of `attributes` and `actions` including *CRUD* operations. These attributes and actions determine the characteristics and behavior of a Compute instance. It is worth noting that there is a specific model for each technology which inherits from the Compute. Regardless of the complexity that hides in the representation of VM and container instances, our model deals with them as Compute instance with generic methods. This will facilitate the addition of a new specific provider's infrastructure. Figure 5.6 is only a simplified version of the model, a more detailed capture figure from Eclipse can be found here[4]. The right part of this figure is the imported, simplified Docker model that have been described in Section 5.4.

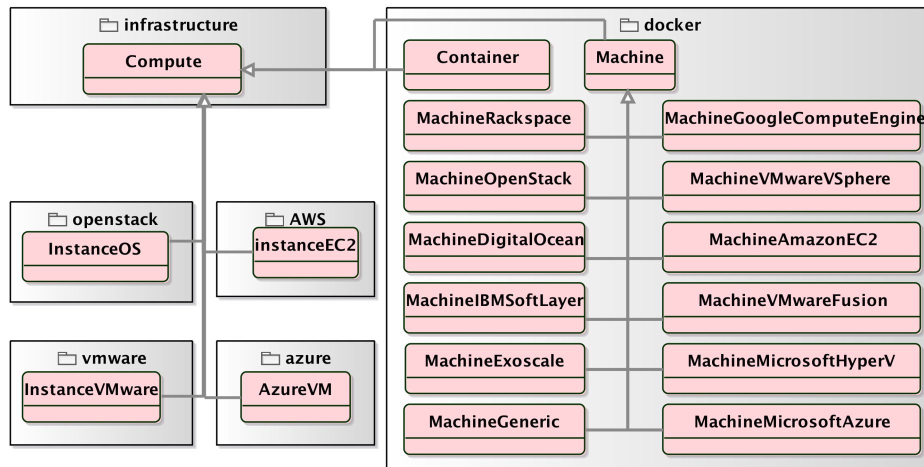


Figure 5.6 Compute instances

5.5.1.5 `ElasticController` (EC)

The `ElasticController` kind represents the elasticity controller. The EC is the core component of this model. As shown in Figure 5.4, many mixins or policies can be applied on the elastic controller. We describe these policies in Section 4.2. In addition, the relationship between the EC and the other components will be highlighted in the coming paragraphs. EC has some attributes such as `minimumLimit`, `maximumLimit` which determine the min/max limit the EC can scale the resource. The resource here is not OCCI resource; it is HG, VM, Memory, vCPUs, etc. EC has also a constraint (stepconstraint) that allows `Steps` to be only used when the EC has a `StepDynamicScalingPolicy`.

5.5.1.6 `Step`

The `Step` kind helps the EC to choose the amount/number of resources/instances to be increased/decreased. Based on the *step bounds* and `size` attributes, EC will decide the amount or size of resources to be increased or decreased. Step works with the `StepDynamicScalingPolicy` that will be explained in Section 4.2.

5.5.1.7 `ActionTrigger` (AT)

AT is used by the EC to perform elasticity actions. AT has three attributes (`action`, `actionType`, `amount`) as shown in Figure 5.4. The `action` specifies the action operation

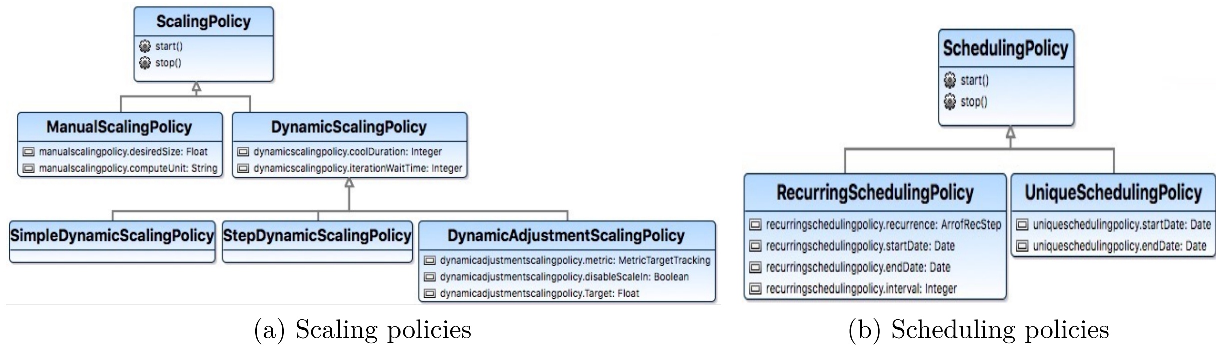
that can be "*add*", "*remove*" or "*set to*". `ActionType` can be "*percentage*" or "*instance*". `amount` determines the amount or size of the added/removed resources, it is based on the `ActionType` attribute to determine its value.

5.5.1.8 `Rule`

The `Rule` kind links an EC to an action trigger, each EC has one or many rules. Each rule link is associated with one `ActionTrigger` (target). A rule is used to evaluate certain metric. The metrics can be CPU, Memory, Response Time that are represented in a form of *mixins* which can be added/removed dynamically to the Rule (`Metric` mixin in Figure 5.4). A Rule owns a set of attributes such as `operator` (equality or relational operator), `threshold`. In addition, `period` and `consecutive` attributes will be passed to the monitoring system to determine the period and interval for the metrics to be monitored. A rule has a constraint (ruleconstraint) to restrict a rule to only have one mixin (metric) at a time.

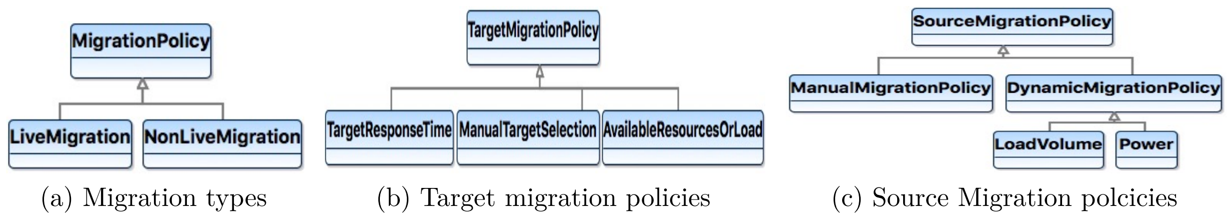
5.5.1.9 MoDEMO link kinds

In OCCI, link is a relation between two resource instances. A link references both source and target resource, *e.g.*, when EC is linked to HG, information about the target (HG) can be sent to the source (EC). OCCI requires *links* to connect the different related resources. As shown in Figure 5.4, our model has the following links. `LoadBalancerLink` links a HG to a LB. Each HG has one LB. `InstanceGroupLink` links a HG to a `Compute` instance. HG has one or many `Compute` instances. `ElasticLink` links an EC to a HG (in case of horizontal elasticity) or a `Compute` instance (in case of vertical elasticity). `StepLink` links an EC to a `Step` instance. EC may have zero or many steps. `ProviderLink` links a HG to a provider. The compute instances of a horizontal group can be deployed on one or many providers. `ProviderInstanceLink` links `Provider` to `Compute`. This link distinguishes which Compute instances belong to which cloud provider. A provider can have zero or many compute instances. `Rule` link is described in the previous subsection to keep the flow of detailing this model.



(a) Scaling policies (b) Scheduling policies

Figure 5.7 Scaling and scheduling policies



(a) Migration types (b) Target migration policies (c) Source Migration polcicies

Figure 5.8 Migration

5.5.2 MoDEMO elasticity policies

As shown in Figure 5.4, an elastic controller is enabled with different types of policies or strategies including scaling and scheduling policies, swapping and migration. There are four different types of scaling policies (Figure 5.7a). In addition, there are two scheduling policies that can be applied to the scaling policies in order to run them in the future (Figure 5.7b). If we look to these different categories of elasticity policies, we realize that the scaling policies launches elasticity policies in the present and may continue to future. While scheduling policies trigger elasticity actions in the future. Migration and swapping are also complementary policies that indirectly enhances the elasticity concept. Therefore, our EC has more than fourteen different modes that enable the adjustment of resources. We introduce these policies in the subsequent subsections:

5.5.2.1 Scaling policies

Scaling policies are elasticity policies which trigger elastic actions from present and may continue to the future. As shown in Figure 5.7a, there are four scaling policies. The difference between these policies hides in how and when to trigger elasticity actions.

A- `ManualScalingPolicy` (MSP)

It means that the user is responsible for performing elasticity actions based on her/his choice. MoDEMO provides an interface to allow performing vertical or horizontal elasticity actions on the HG or `Compute` instances.

B- `SimpleDynamicScalingPolicy` (SimpleDSP)

As other scaling policies, *SimpleDSP* permits to execute elasticity actions dynamically. However, when and how to scale in/out or up/down resources differs from one policy to another depending on the policy type and algorithm used. As shown in Figure 5.4, the EC may have many rules and each rule is associated with its action trigger. As explained before, the rule evaluates a metric, *e.g.*, if the CPU utilization (`Metric`) during an interval of 2 minutes (`periods`) for 3 consecutive periods (`consecutive`) is greater than (`operator`) 90 (`threshold`), then the simple dynamic policy may execute the linked action to that rule. The simple dynamic policy will classify the rules into two categories: ***i)*** rules that have an increase action and ***ii)*** rules that have a decrease action. For the first category, it will evaluate (logical or) between the rules and will execute the action of the first rule that is true. For example, suppose we have two rules that evaluate CPU and memory metrics respectively, if the rule-condition of any of these rules is met, the elasticity action is fired, because the applications may consume one resource (*e.g.*, CPU, memory, etc.) more that the other and that is why the rules are evaluated based on the logical or. For the second category, the rules will be evaluated based on (logical and), if all rules are true in this category, the action which decreases less amount of resources will be chosen from the actions associated with these rules. The reason that the elasticity action is not fired unless all the rules (logical and) are true is simply the applications performance is sensitive to resources, therefore the elasticity controller ,for example, will not decrease the CPU even if its rule is true if there is another rule in the policy that evaluates memory and this rule is not true. The action which decreases less amount of resources will be chosen in order to not have a negative impact on the system due to the scaling in/down actions.

C- `StepDynamicScalingPolicy` (StepDSP)

The main difference between this policy and the previous one is the size or amount of the added/removed resources (VM instances, CPUs, memory) per scaling action. In this policy, the size is determined by the metric usage, threshold and step limits. If the EC has no steps (`Step`) associated with it, StepDSP will simply execute the action associated with the rule when the rule is true. If steps are associated with the EC, the metric usage will be compared with ($threshold \pm lowerStepBound$) and ($threshold \pm UpperStepBound$)

and will take the *size* attribute in Step as the amount or number of resources to be added/removed instead of the amount defined in actiontrigger as shown in Figure 5.4. To sum up, the metric utilization is compared with the ranges specified in the Steps to determine the number/amount of resources to be added or removed.

D- `DynamicAdjustmentScalingPolicy` (DASP)
DASP or target tracking scaling policy [230] performs dynamic scaling based on a target value. A target value or threshold is set for a certain metric. The DASP will keep monitoring the metric and trigger scaling policy actions to increase or decrease resources based on the target value. The scaling policy adds or removes capacity as required to keep the metric at, or close to, the specified target value. However, the scaling down actions are restricted to the formula: $\left(\frac{(count-1)*T_{metric}}{count}\right) - \alpha$, in order to avoid the rapid oscillation of scaling in/out, down/up actions [41]. T_{metric} is the target value (threshold), *count* is the number of current compute instances, vCPUs or memory size depending on the elasticity type, α is a small integer. The metric usage must be less than the value returned by the above formula.

5.5.2.2 Scheduling policies

We present the scheduling policies as shown in Figure 5.7b. Scheduling policies are policies which trigger elasticity actions or other elasticity policies in the future. The objective of scheduling policies is to trigger elastic policies and scale the application in response to predictable load changes.
A- `UniqueSchedulingPolicy` (USP)
An EC can execute elasticity decisions based on a schedule in response to predictable load changes. USP can be applied to the MSP, SimpleDSP, StepDSP, and DASP in order to be fired/stopped at the start/end dates specified. It gives also the possibility to cancel the scheduled future actions.

B- `RecurringSchedulingPolicy` (RSP)
Similar to USP, RSP can be applied to the MSP, SimpleDSP, StepDSP, and DASP in order to be fired in the future. However, recurring scheduler allows the action to be repeated in the future at the time specified. RSP can start/stop the recurrence based on start/end dates specified, however the dynamic policies (DP) can run infinitely, therefore RSP gives a control on DP. For example, RSP can fire the SimpleDSP every weekend for ten hours. RSP accepts all the *cron expressions* [231], which are able to create firing

schedules based on the expression specified. For example, "At 8:00 am every Monday through Friday" or "At 1:30 am every last Friday of the month", etc.

5.5.2.3 `SwappingPolicy`

The above elasticity policies are considered as the de facto solution to support the timely provisioning and de-provisioning of resources, this elasticity can still be broken by budget requirements or physical limitations of a private cloud. SwappingPolicy is an alternative, yet complementary, solution to the problem of resource provisioning by adopting the principles of swapping in the context of cloud computing. In particular, this policy consists in detecting idle virtual machines, containers to recycle their resources when the cloud infrastructure reaches its limits. Cloud providers use the technique of overcomitting (*e.g.*, oversubscription in VMware, overcommitment in OpenStack) in order to virtually increase their capacity. Overcommitting is a technique of allocating more virtualized CPUs or memory than the real physical resources. These techniques allow to deploy more workloads but stability issues can occur and major performance problems can be introduced affecting all of the workloads running on the infrastructure. In [232], we evaluated the impact of resource overcommitment on VM performance, and we found that once the number of allocated physical core is reached on a compute node (P(CPU)), the performance becomes linearly impacted by the provisioning of new VMs. Additionally, in [233], they found that the resource oversubscription with ratios 1:1 to 3:1 has no problem on performance but 3:1 to 5:1 may begin to cause performance degradation. This policy will recycle the unused resources and will automatically restore them on demand via listening through a proxy for their requests.

5.5.2.4 `MigrationPolicy`

Migration can be also considered as a needed action to further allow the elasticity when there is no enough resources on the host machine. However, it is also used for other purposes such as migrating a VM to a less loaded physical machine just to guarantee its performance or to another destination in order to power off the source host and save energy, etc. Migration types are categorized as live or non-live VM migration as shown in Figure 5.8a. A live migration does not suspend application service during VM/container migration, whereas a non-live pattern follows pause, copy, and resume methodology to migrate a VM/container. In our model, the migration policy depends on `TargetMigrationPolicy` to choose the destination to where the migration will be.

5.5.2.5 `TargetMigrationPolicy`

The EC will use this policy to choose the target provider or data center that the VM/container will be migrated to. Our model supports three mixins that choose the target destination for the migrated workloads as shown in Figure 5.8b.

- `ManualTargetSelection`: The user will choose his/her preferred provider/data center manually.
- `TargetResponseTime`: The nearest target will be chosen.
- `AvailableResourcesOrLoad`: The target with most available resources will be chosen.

5.5.2.6 `SourceMigrationPolicy`

Similar to the other policies, MoDEMO offers a mean for migration based on the EC `Rule` and its associated `Metric`. However, since the migration has various objectives such as power management, fault tolerance, load balancing, system maintenance, performance issues, etc., `SourceMigrationPolicy` permits to direct the migration process based on certain objective. If this mixin is not present, the EC will use its rules and metrics to decide when to trigger the migration process. If `SourceMigrationPolicy` is present, it will direct the migration process based on certain objective, *e.g.,* saving energy. It has many mixins where each mixin represents an algorithm based on a single objective as shown in Figure 5.8c.

- `ManualMigrationPolicy`: Based on the user (the user could be the cloud provider, the service provider or the final customer) choice to migrate, *e.g.*, if the user decides to effectuate maintenance, this mixin can be used.
- `Power`: The migration will be triggered in order to save energy.
- `LoadVolume`: The migration will be fired when an overload is detected, *e.g.*, storage is running out of space.

In general, `MigrationPolicy` specifies the migration method, `TargetMigrationPolicy` determines where to migrate, the EC `Rule` and `Metric` or `SourceMigrationPolicy` determines when to migrate. If EC is connected to a single Compute, that Compute will be the entity to be migrated but if the EC is linked to a HG, its instances will be migrated to a higher capacity hosting server or cluster of servers.

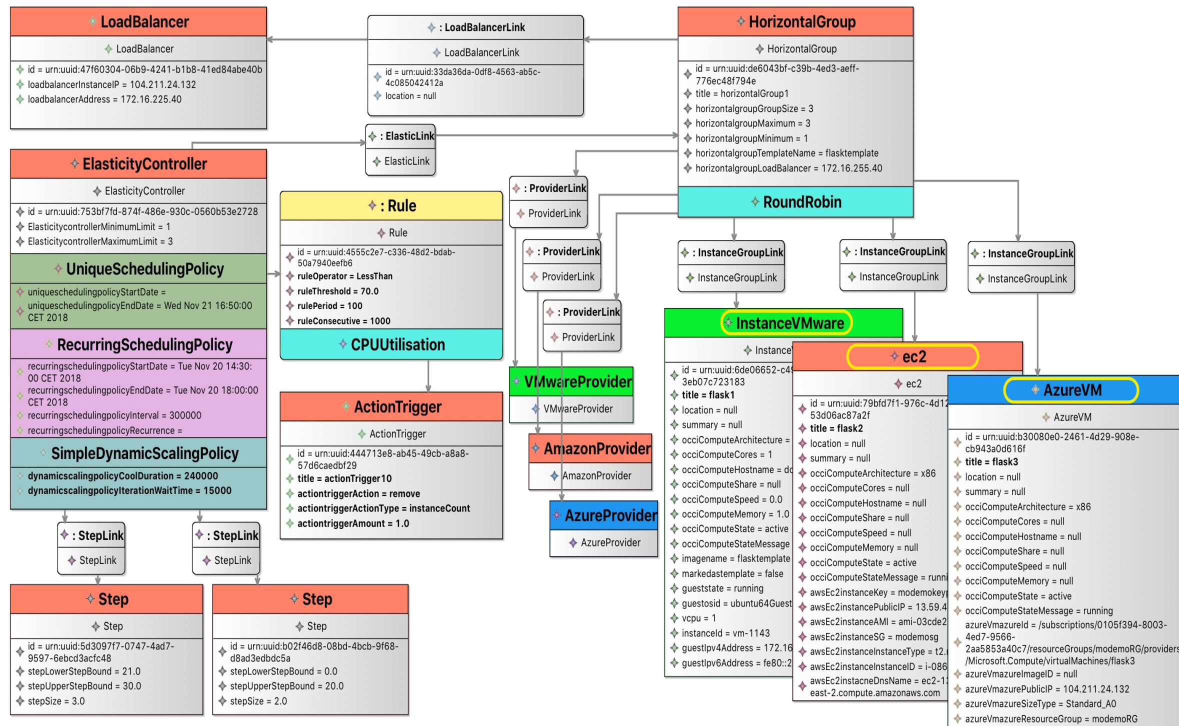


Figure 5.9 MoDEMO design configuration

5.5.3 MoDEMO design tool

MoDEMO is implemented as a set of Eclipse plug-ins on top of OCCIware Studio [222]. Figure 6.2 shows a runtime configuration of our MoDEMO model. All the parameters in the EC and its associated rules, steps, metrics, action triggers can be modified at runtime as shown in Figure 6.2 and indicated in Section 5.2. The list of the configurable settings at runtime will be presented in Chapter 6. In addition, MoDEMO is powered with plugin/plug out functionalities. As shown in Figure 6.2, we drag and drop any policy from the design tool to the EC configuration, which eases the utilization of our framework. The metrics that are evaluated by the rules can be also dragged in/out dynamically in the Rule box as shown in the configuration, thanks to the OCCI *mixin*, which permits adding/removing plugins dynamically to the system.

5.6 Discussion

MoDEMO is a unified system that supports vertical and horizontal elasticity for both VMs and containers along with a seamless and simultaneous use of multiple cloud infrastructures. It is based on a standard representation of the infrastructure entities and elasticity components. This system is demonstrated on real platforms using real applications. MoDEMO handles many challenges, which are presented in Chapter 1. To solve the challenge of resources availability or outage in specific cloud provider, MoDEMO can rent resources from different cloud providers. MoDEMO allocates resources dynamically and seamlessly from different providers, thus it overcomes the problem of heterogeneity of clouds which has different APIs and technologies. In addition, MoDEMO abstracts the computing units, VMs (VMware VM, AWS EC2, GCP instance, etc) or containers as a compute instances, it solves the heterogeneity challenge of the virtualization technologies and techniques. For addressing the granularity of resources, our approach permits to have different instances (VMs, containers), different configurations of vCPUs and memory size. Furthermore, to handle the challenges of vendor-lock-in and proprietary systems, our approach is standard-based, model-driven that presents all aspects of elasticity at a high level of abstraction. MoDEMO is loosely coupled, highly extensible, highly configurable at runtime rather than a single back-box EMS. Since MoDEMO is loosely coupled and extensible, new components such as cloud providers or policies can be easily and savely added to the system without introducing redesign or risk issues.

Although MoDEMO permits to execute several elasticity policies, it has some limits. MoDEMO offers the possibility to enable different policies, however, there is no synchronization between them. It allows launching different strategies but each one is independent of the other. The user must pay attention for the settings, *e.g.*, lower/upper bounds of the step policy. It is the responsibility of the user to not let a gap between the different steps or to not make overlapping of the threshold values. Additionally, the algorithms are rule-based, while they can be adapted and changed at runtime, MoDEMO could be improved to integrate predictive and machine learning approaches along with the reactive strategies.

5.7 Related Work

To the best of our knowledge, no approach has been proposed so far on managing elasticity and unifying its different policies that are driven by the models in the cloud. We, therefore, present and discuss the closest related works that are relevant to our approach. Gandhi et al. [234] proposed cloud service, Dependable Compute Cloud (DC2), that dynamically scales the application deployment based on user-specific performance requirements. This approach makes use of a queuing network model and Kalman filtering technique to determine the necessary scaling actions. CloudMIG [235] is a model-based approach for migrating legacy software systems to scalable and resource-efficient cloud-based applications. The solution model focuses on reengineering activities during the migration of existing software systems to scalable and resource-efficient Platform as a Service (PaaS) and Infrastructure as a Service (IaaS) rather than a representation of elasticity policies or infrastructure entities. Ali-Eldin et al. [113] proposed hybrid horizontal elasticity controller that incorporates both reactive and proactive components to dynamically change the number of VMs allocated to a service running in the cloud based on the current and the predicted future demand. This approach models the state of the system, e.g., infrastructure current servers, number of requests, etc. and the future usage but the elasticity controller is a single entity. [91] describes a Multi-Cloud-PaaS (MCP) architecture to manage elasticity across multiple cloud providers. In this work, they present a non-modular, flat architecture to deploy an application on a list of static catalog of cloud providers. The mentioned proposals, as most of other works around elasticity, represents the elasticity controller as a single black box entity while our work permits to dynamically add/remove different components, thanks to the dynamic modular nature of the design and method used. A recent work [97] around elasticity proposes a Cloud Resource Description Model (cRDM) based on a state machine for describing the cloud elasticity. Authors in [97] defined fixed set of states for the cloud resource or application where the elasticity events and transitions loop inside the limited space. This system requires an intensive manual intervention to define the state of the system and its associated transition and events. cRDM supports multiple clouds in a condition that the provider is manually defined in the state transition. All the works in this field are not based on dynamic modular approaches. Generally, they rely on a single metric and one elasticity policy. MoDEMO is a unified, modular-based framework that covers various approaches for elasticity and it allows the choice according to the need and behavior of the system. MoDEMO allows to seamlessly manage elasticity across multiple clouds and it releases the users to be aware of different low-level cloud service APIs to describe and

control the elasticity of cloud services. MoDEMO supports multiple elasticity policies and actions, different elasticity types (horizontal and vertical), different virtualization techniques (VMs and containers) and multiple clouds.

5.8 Conclusion

This chapter presented our approach named MoDEMO for the Model-Driven Elasticity Management with OCCI. MoDEMO has the following features: *i*) evolutive modular design based on OCCI standard, *ii*) it addresses the heterogeneity of elasticity policies and combines the features of three popular cloud providers (AWS, Azure, GCP), *iii*) it permits reconfiguration and setting at runtime, *iv*) it provides a seamless integration of the infrastructure of different cloud providers for both VMs and containers computing units, *v*) and easy to use with the drag and drop functionalities. MoDEMO is loosely coupled unified modular system which makes it a step forward towards a stand-alone approach of cloud elasticity management. Among its benefits, MoDEMO solves the problem of interoperability by seamlessly leasing resources from multiple clouds and the problem of heterogeneity by using different computation capabilities and various elastic strategies.

The next chapter presents MoDEMO and Docker model evaluation.

Chapter 6

MoDEMO Evaluation

In this chapter, we present a wide range of experimentations to evaluate and validate the approach presented in Chapter 5. In particular, the MoDEMO *approach is evaluated as well as the Docker model which acts as container compute components of* MoDEMO*.* MoDEMO *policies is compared with different cloud providers. Runtime settings are tested while the execution of a running application in the production environment.* MoDEMO *permits to timely adapt resource capacity according to the workload intensity and increase application performance without introducing a significant overhead. In addition, Docker model is illustrated using an event processing application and we show how our solution provides a significantly better compromise between performance and development costs than the basic Docker container solution with a negligible overhead.*

6.1 Introduction

MoDEMO is a unified elastic system that manages in an autonomic way the resources (VMs, containers, CPUs, memory), being adaptive to dynamic workloads, allocating additional resources when workload is increased (on different providers)

This chapter is derived from:

- ***Yahya Al-Dhuraibi**, Faiez Zalila, Nabil Djarallah and Philippe Merle, "Model-Driven Elasticity Management with OCCI," submitted to IEEE Transactions on Cloud Computing (TCC), June 15 2018, first feedback major revision.*
- *Fawaz Paraiso, Stéphanie Challita, **Yahya Al-Dhuraibi** and Philippe Merle, "Model-Driven Management of Docker Containers," 2016 IEEE 9th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2016, pp. 718-725. doi: 10.1109/CLOUD.2016.0100*

and deallocating resources when workload decreases using multiple policies. MoDEMO is a loosely coupled system where it has different components and various policies enabling them that can be plugged in/out dynamically according to the need. In this chapter, we verify the different policies supported by MoDEMO in comparison with other other worldwide cloud providers. In addition, we verify the configurability, extensibility and dynamicity of this approach. In this chapter as well as throughout the whole dissertation, the evaluations are performed using real applications on Scalair company infrastructure. The application descriptions and configurations will be described in the different evaluation subsections. The experiments show that MoDEMO is a unified system where it supports the elasticity policies provided by the worldwide public cloud providers and more, is configurable at runtime, and is with negligible overhead. Moreover, Docker model permits to design, verify, deploy Docker containers with a negligible overhead.

This chapter is organized as follows: Section 6.2 presents a high level architecture of MoDEMO. This architecture is a real use-case implementation of MoDEMO. Section 6.3 describes thoroughly the different evaluation scenarios. Section 6.4 provides a conclusion for this chapter.

6.2 High Level Architecture Overview

Figure 6.1 gives a high-level illustration to understand the concepts behind our architecture. This architecture is composed of the following parts: MoDEMO model manager, Provider connectors (OCCI), Zabbix monitoring system, and the load balancer HAproxy. The provider connectors include VMware connector, OpenStack connector, Amazon EC2 connector, GCP connector and Docker connector. These different components have been described in Chapter 5. Figure 6.1 defines the relationship between the different components of the system. MoDEMO manager communicates with the provider-specific connector which in turn will provision, execute actions on the executing environment via their specific orchestrators, *e.g.*, Vcenter, Docker machine/engine, etc. In addition, MoDEMO must capture the resource utilization in order to allow the elasticity controller to perform scaling actions according to the chosen policy. Therefore, MoDEMO communicates with our monitoring system (Zabbix). Zabbix is an open source monitoring tool that works with a centralized Linux-based Zabbix server [236]. According to the type of elasticity, MoDEMO must register/remove the instance (VMs) in a load balancer. We use HAproxy which is a fast and reliable open source solution offering high availability, load balancing, for implementing complex load balancing in terms of different algorithms

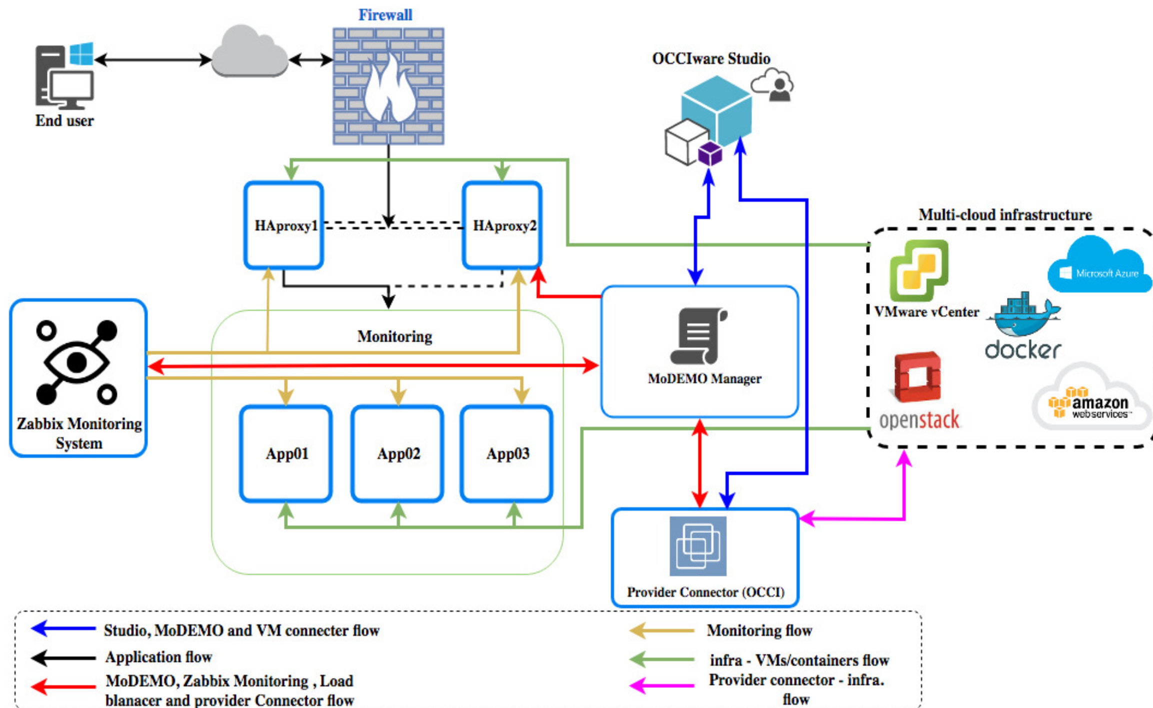


Figure 6.1 Architecture overview

for traffic distribution [237]. In addition, if a compute instance is removed, it will be deregistered from both the monitoring system and the load balancer. It is worth noting that MoDEMO uses generic methods, for example, it deals with a compute instance rather than VM instance directly.

6.3 Validation

MoDEMO has a wide range of components, therefore we evaluated some important aspects of our work by using Flask application version 0.10.1 [238] and Redis database (DB) version 3.2.0 [239]. Flask is a framework for Python based on Werkzeug which computes a calculation (through REST API) and stores the results into the database (Redis). Flask runs an application for Fibonacci numbers, written in Python. The reason we use Fibonacci with flask application is that it intensively consumes resources (CPU, memory). In addition, each operation takes a long time to be calculated. Apache HTTP server benchmarking tool (ab) is used to generate workloads to the Flask application in order to calculate the mathematical operations (Fibonacci sequence). The result of the mathematical operation is stored in the Redis DB. We performed all our experiments on

Scalair[1] infrastructure. Scalair is a private cloud provider company. Scalair infrastructure is managed by VMware vSphere 6.0 technology. The hardware specifications consist of 2 powerful servers: 2 HP ProLiant DL380 G7 (Intel(R) Xeon(R) CPU X5650 @ 2.67GHz, 84 GB, 6 NICS). Our evaluation answers the following questions:

- *Q#1*: Are all the AWS, MS Azure and GCP scaling policies supported by MoDEMO?
- Q#2: What are the possible configurations or settings at runtime?
- Q#3: What is the overhead introduced by MoDEMO model?
- Q#4: What is the overhead introduced by Docker model?

MoDEMO is a unified framework that enables all the elasticity policies supported by the worldwide cloud providers. We have chosen AWS, MS Azure and GCP as indicated in *Q#1* because they are the major actors in the cloud market. According to Synergy Research Group (SRG) [240], AWS, MS Azure and GCP represent the majority of cloud market share in the world, *e.g.,* Amazon maintained its dominance as its market share to 34% in Q2 2018. Secondly, MoDEMO supports runtime configurations, therefore *Q#2* will verify what are the different components that can be configured at runtime. Thirdly, since our system is configurable at runtime, it could impact the performance and that is why we measure the overhead in *Q#3* and *Q#4* for the two models.

6.3.1 MoDEMO policies

In order to answer **Q#1**, we compare MoDEMO elasticity policies and the corresponding AWS, MS Azure, GCP scaling policies in Table 6.1.

Table 6.1 shows that MoDEMO supports all the elasticity policies found in AWS, MS Azure, and GCP. However, MoDEMO not only supports the elasticity policies in the above popular providers, it suggests some improvements. For instance, simple policy in AWS only supports one metric (rule) per policy, MoDEMO permits many rules to be aggregated in the same policy. The schedulers in MS Azure are limited to certain dates and days while MoDEMO supports any combination and recurrence frequency. In addition, MoDEMO avoids the rapid oscillations as explained in the DASP because the scaling down actions are not only limited to the target usage but also to the group size.

[1] http://www.scalair.fr

Table 6.1 Elasticity policies supported by public cloud providers and MoDEMO

MoDEMO	**AWS**	**Azure**	**GCP**
MSP	Manual	Instance account	Manual
MSP with USP	Scheduled	Scheduled	–
MSP with RSP	Scheduled cron	Scheduled	–
SimpleDSP	Simple Policy	Metric based	–
SimpleDSP with USP	–	Scheduled (specific dates)	–
SimpleDSP with RSP	–	Scheduled (specific days)	–
StepDSP	Step Policy	–	–
StepDSP with USP	–	–	–
StepDSP with RSP	–	–	–
DASP	Target tracking	–	autoscaler (single/multiple metrics)
DASP with USP	–	–	–
DASP with RSP	–	–	–
Swapping	–	–	–
Migration	AWS Server Mig. Service	Windows-built mig.	CloudEndure

Finally, these policies are applicable for both vertical and horizontal elasticities, while the supported policies in AWS and GCP can only be applied to the horizontal elasticity.

6.3.2 Runtime settings

In order to answer **Q#2**, we have deployed our application as described before, where MoDEMO used to deploy and control the elasticity. Figure 6.2 is a MoDEMO runtime configuration for the deployed application. It shows some of the components of the model. Frame (a) in Figure 6.2 shows the Eclipse Model Explorer used to navigate through the different project configurations containing our MoDEMO Model. Frame (b) displays the design area that provides a graphical representation of some components of MoDEMO Model. Frame (c) in Figure 6.2 displays the configuration pallet that represents the MoDEMO Model elements such as HG, Link, LB, Compute, etc. Frame (d) in Figure 6.2 gives an outline or a global view of the modeled components (EC, application, infrastructure, LB, etc.). The Eclipse properties editor is used for visualizing and modifying attributes of a selected modeling element, console tab, errors tab, etc.

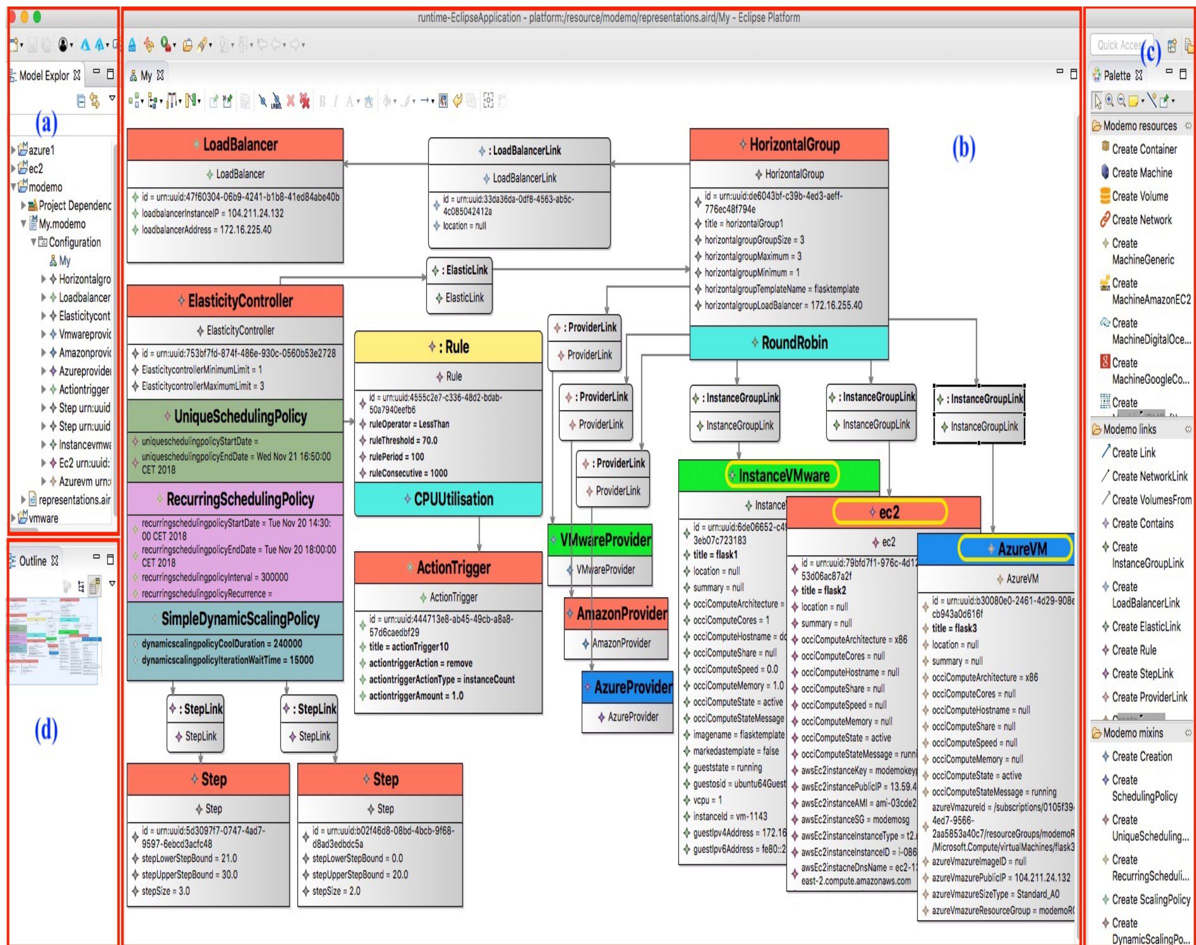


Figure 6.2 MoDEMO runtime configuration

are at the bottom of the figure, we closed them to gain space. At the beginning of the experiment, the HG (Scalair HG in Figure 6.2) has only one VMware VM instance (Flask1). With the increased numbers (Fibonacci numbers) send to the application, other two instances (Flask2 and Flask3) are provisioned on AWS and Microsoft Azure respectively. The instances are provisioned subsequently on the different providers. This is because of the allocation policy (*RoundRobin*) set on the horizontal group which is used to choose the cloud provider. If there is other more instances to be provisioned, the next instance will be deployed on the private VMware provider and etc. as shown in Figure 6.2. During the workload generation, MoDEMO adds more instances, at the same time we also change many parameters at runtime. Table 6.2 shows a list of parameters that

Table 6.2 Configurable parameters at runtime

Elasticitycontroller Maximum Limit	*Rule Operator*
Elasticitycontroller Minimum Limit	*Rule Threshold*
Dynamicscalingpolicy Cool Duration	*Rule Period*
Dynamicscalingpolicy Iteration Wait Time	*Rule Consecutive*
Step Lower Step Bound	*ActionTrigger Action*
Step Upper Step Bound	*ActionTrigger Action Type*
Step Size	*ActionTrigger Amount*
Mixin CPUUtilisation	*Mixin MemoryUtilisation*

can be changed at runtime. During the execution of the experiment, we change the values as shown in Figure 6.2 at runtime. Table 6.2 only lists some of the important parameters that control the behavior of elasticity while there are other parameters and mixins such as *name, title, load balancer algorithm mixins, allocation policy mixins*, etc. can be changed at runtime. In addition, components of the model such as *Rules* and *AT* can be added dynamically to the design at runtime when the *SimpleDSP* is applied. To integrate elasticity controller with a new scaling policy, it is enough to drag it from the design tool. With the possibility to drag/drop policies/metrics to the configuration as shown in Figure 6.2, MoDEMO eases the utilization for users. The Compute instances and the LB are deployed and configured transparently and independently of any deployment tool.

6.3.3 MoDEMO overhead

To determine the overhead introduced by our MoDEMO model (**Q#3**), we evaluate two scenarios where our application described previously in this section is deployed. A certain workload is generated to the application. The application is deployed and the elasticity

is achieved using two scenarios: i) with script which runs directly on the infrastructure, the scripts used in this experiment can be found here[2], and ii) with MoDEMO model. Of course, we take one case of the elasticity which is the simpleDSP. The two scenarios are repeated ten times for both the vertical and horizontal elasticity.

6.3.3.1 Vertical elasticity

We measured the average total time it takes to execute the workload and perform elasticity actions, i.e. reconfiguration of the VM resources. In Table 6.3, we present the results of the average total time for executing the workload using the vertical elasticity for both scenarios (script runs directly on the infrastructure and MoDEMO), as well as the median, minimum and maximum values, and the overhead introduced by the MoDEMO Model. The overhead introduced by our model is **0.8%**. It is worth noting that during the experimentation of the two scenarios, we have set the same values of the elasticity parameters such as *threshold*, *cool duration*, etc.

Table 6.3 Vertical elasticity overhead

Scenario	Avg. total time	Median	Max. value	Min. value	Overhead
Script on the infr.	391,7301 sec	391,8085	396,877	383,427	-
MoDEMO Model	394,8677 sec	394,74	401,964	391,85	0.8%

6.3.3.2 Horizontal elasticity

Similarly, we have generated a workload to our application, we measured the time it takes to add more instances and to execute the workload in the context of the horizontal elasticity. Table 6.4 shows the average total time that it takes to achieve the horizontal elasticity and manipulate the workload (the scientific calculations) in the two scenarios, in addition to the median, maximum and minimum values, and overhead. The overhead introduced by our model is **1.64%**. The overhead with the horizontal elasticity is a little bit bigger than the vertical elasticity, the reason is that simply the horizontal elasticity involved with more model configurations and components. The median for both types of elasticity is very close to the mean total execution time which indicates that there are no outliers that are much greater or smaller than most of the other values of our ten experiments.

[2] https://github.com/yehia2221/MoDEMO

Table 6.4 Horizontal elasticity overhead

Scenario	Avg. total time	Median	Max. value	Min. value	Overhead
Script on the infr.	412,085 sec	412,014	418,661	402,062	-
MoDEMO model	418,855 sec	420,2785	422,23	408,275	1.64%

These experiments show that there is just a very small overhead introduced by adding the MoDEMO model. The overhead is negligible regarding all advantages provided by our approach.

6.3.4 Docker model evaluation

In this part, we evaluate *Docker Model* approach with respect to both performance and online-model-manipulation achievement.

In order to answer **Q#4** and evaluate our *Docker Model* in cloud environments, we run a distributed containerized application composed of 8 containers. This distributed application is a computation system for processing large volume data. To focus on the real performance of our *Docker Model*, all our experiment were performed using a Macbook Pro workstation with 2,2 GHz Intel Core i7 processor, 16 Go 1600 MHz DDR3, OSX version 10.11.2 (15C50), and Oracle Java 1.7 to run our *Docker Designer*. The deployments of the containers are done with Scalair cloud provider, which use VMware to build their cloud infrastructure.

We have decomposed **Q#4** into two sub questions: (i) Does *Docker Model* introduce overhead? (ii) How time is taken to manipulate *Docker Models* online? The following sub sections discuss these questions.

6.3.4.1 Overhead introduced by the Docker Model

To determine the overhead introduced by our *Docker Model*, we evaluate two of the scenarios where our distributed containerized applications were created, started, and stopped: i) natively with Docker, and ii) with Docker integrating our model. The scenario was executed hundred times on each of the two implementations.

In Tables 6.5, 6.6 and 6.7, we present the results of the average time for creating, starting, and stopping containers for each implementation, as well as the mean overhead introduced by the *Docker Model*.

The overhead introduced by our model when creating containers is **1.11%**, this creation phase consists of pulling image once and the creation of the containers. Next, when starting the containers, the overhead introduced by our model is **2.12%**. Then, the overhead introduces by our model when stopping containers is **2.25%**. The small overhead fluctuation of the start and stop actions compared to the creation action is due to the model elements manipulation.

Table 6.5 Container creating time and overhead.

Create action	Avg. create time	Docker Model overhead
Docker	168.509 sec	-
Docker with Model	170.382 sec	1.11%

Table 6.6 Container starting time and overhead

Start action	Avg. start. time	Docker Model overhead
Docker	5.033 sec	-
Docker with Model	5.04 sec	2.12%

Table 6.7 Container stopping time and overhead

Stop action	Avg. stop time	Docker Model overhead
Docker	84.12 sec	-
Docker with Model	86.01 sec	2.25%

This experiment shows that there is an overhead introduced by adding the *Docker Model*. The overhead is negligible regarding all advantages provided by our approach: verification of containers, increasing resource at runtime (cf. Section 5.2).

6.3.4.2 Online model manipulation

When manipulating our model, we have evaluated the time took by our Docker Model to detect and integrate the changes inside the model. In this context, we have modified the model and evaluate the time taken by the Docker Connector to propagate changes in the *Executing Environment*. Conversely, we operate modification in *Executed Environment* and evaluate the time taken by Docker Connector to operate the changes in the model.

When creating a new container from *Executed Environment*, our Docker Connector takes about **12** milliseconds to detect the changes and spends about **290** milliseconds to represent graphically a container into the model. Next, when creating new container in the model, it took about **10** milliseconds for the Docker Connector to detect the changes. This experiments shows that our Docker Connector reacts quickly to change.

6.4 Conclusion

This chapter presents evaluation for MoDEMO approach. Experiments demonstrate that MoDEMO covers the elasticity policies provided by the well-known public providers and more, is configurable at runtime, and is with negligible overhead. We have designed our approach on a graphical model-driven tool chain called Multi-cloud studio to design, reason, and deploy elasticity policies across different providers for both containers and VMs. MoDEMO implement different infrastructure extensions including Docker model that we have proposed in Chapter 5. Docker model permits to design, verify, deploy Docker containers with a negligible overhead.

Part IV

Conclusion and Final Remarks

Chapter 7

Conclusions and Future Directions

This chapter summarizes the research works achieved in this thesis on managing elasticity in cloud computing. In addition, this chapter identifies some future directions to pursue in this area.

7.1 Summary and Conclusions

As cloud computing has gained popularity in both industries and academia, elasticity became a vital feature to provide dynamically and timely the resources according to the demand.

The contributions of this thesis are aimed at providing a flexible framework for managing the different aspects of elasticity in cloud computing. We started by exploring the concept of elasticity and studying different existing works related to it. A precise definition for elasticity is proposed, which gives a clear distinction between elasticity and its related terms, scalability and efficiency. This state of art provides a detailed, comprehensive review and analysis of different works related to elasticity for both VMs and containers. Based on the analysis of diverse works, we have proposed a thorough taxonomy of the elasticity mechanisms. The state of art ends by discussing open issues, research challenges and perspectives.

Afterwards, we proposed different approaches to manage the elasticity, we started from a low-level or system-level until we arrived a complete high level abstraction for all aspects of elasticity. Firstly, ELASTICDOCKER, the first system powering vertical elasticity of Docker containers autonomously is proposed. A further complementary, yet

a new coordinated controller, is proposed. The coordinated controller manages both container and VM vertical elasticity. As vertical elasticity is limited to the host machine capacity,ELASTICDOCKER effectuates container live migration when there is no enough resources on the hosting machine. ELASTICDOCKER reduces expenses for container customers, make better resource utilization for container providers, and improve Quality of Experience for application end-users. In addition, the experiments demonstrated that: (i) the coordinated vertical elasticity is better than the vertical elasticity of VMs by 70% or the vertical elasticity of containers by 18.34%, (ii) our combined vertical elasticity of VMs and containers is better than the horizontal elasticity of containers by 39.6%. In addition, the controller performs elastic actions efficiently on different virtualization technologies. In addition to the elasticity management of containers, these approaches uses a reliable, low-level monitoring system.

After that, we moved toward a more general, higher level of abstraction, standard-based, model-driven approach. This approach handles many challenges including variability of elasticity strategies and policies, resource availability, heterogeneity of different cloud providers, granularity of resources, manageability at runtime of elastic systems, consistency of deployment, etc. Therefore, in this part, we started by introducing Docker model. This approach is model-driven management of Docker containers that ensures deployability verification of Docker containers at design time, provides synchronization between the designed containers and those deployed, and manages resources at runtime. Secondly, we proposed MoDEMO, a model-driven elasticity management with OCCI. MoDEMO is a unified system that supports both vertical and horizontal elasticity for both VMs and containers along with a seamless and simultaneous use of multiple cloud infrastructures. It is based on a standard representation of the infrastructure entities and elasticity components. It also permits setting-up and configurations at runtime. Docker model is integrated with MoDEMO along with other infrastructure extensions of different provider infrastructures. These models introduce negligible overhead. MoDEMO supports all the elasticity policies offered by the worldwide cloud providers and more.

This work is integrated in the OCCIware project, therefore, it has a real implementation rather than a proof of concept or theoretical abstraction.

7.2 Future Directions

In this thesis, we handled different aspects regarding cloud elasticity. Elasticity is a wide concept which has different perspectives and purposes. We solved several research challenges and we included others in our future work.

7.2.1 Proactive approach

We envision extending our elasticity models with features to support predictive approaches in order to anticipate workloads and rapidly scale up resources. When containers are used, there is no any impact of the start-up time. However, when big VMs are used, a proactive approach will decrease the start-up time. Even with the utilization of proactive approaches, a reactive approach (static thresholds) is used at certain point. Therefore, we intend to integrate one or hybrid technique of the following forecasting approaches:

- **Time series analysis**: Elastic systems are driven by the input workloads. Time series approaches analyize these workloads by taken measurements in a uniform intervals. Time series approaches try to construct patterns based on the history of the workloads, these patterns are used by other models to anticipate the future workloads. Time series analysis include many common techniques such as Moving-Average, Auto-Regression, ARMA, Holt winter, machine learning, pattern matching, Fast Fourier Transform, etc.
- **Model solving mechanisms** are probabilistic or mathematical models to study the behavior of the system and predict its future state. Generally, the system state space is fixed or limited. Example of such systems are Markov Decision Processes (MDPs), probabilistic imed automata (PTAs). Specification tools such as Alloy can be used to built offline these mathematical models.
- **Reinforcement Learning (RL)** is an area of machine learning based on interactions between an agent and the environment or target system. The agent takes action, which is in turn interpreted to a reward and a representation of system state, the reward and the state are fed back to the agent which will take or adapt new decisions.

7.2.2 Application-based elasticity

While our proposed approaches can be applied on both infrastructure and application layers, e.g., triggering elasticity policies according to the response time of the application, we plan to propose a dedicated elasticity framework that manages internal application configurations at runtime. The challenge is that each application has its different parameters and configurations, for example, number of connections in web applications. We plan to design approaches that manage application parameters according to the workloads. Applications are installed with pre-configured default values which limits the application capacity to process certain workloads, an intelligent system can modify these parameters to allow applications to adapt to the demand. Regarding the application elasticity, it will be difficult to have one framework to manage different categories of application, however, similar applications can be grouped in dedicated categories.

7.2.3 Dynamic thresholds

The elasticity reactive approach is based on a set of rules and conditions. When a condition is met, a corresponding elasticity action may triggered. Generally, a metric is compared against fixed thresholds, and these thresholds are user-defined values. The questions that arise are these values the appropriate values that meet the elasticity purpose? How to set the correct values based on the application type and state? In our future perspectives, we intend to use adaptive thresholds that change dynamically according to the state of the hosted applications.

7.2.4 Elasticity coordination

Throughout this dissertation, we have proposed different elasticity policies, we plan to propose an intelligent coordination between the different elasticity policies. In our approach MoDEMO, the elasticity controller can be enabled with different combinations of elasticity policies, the objective is to have a smart coordination and synchronization between these policies.

References

[1] Lee Badger, Tim Grance, Robert Patt-Corner, and Jeff Voas. Draft cloud computing synopsis and recommendations. *NIST special publication*, 800:146, 2011.

[2] Nikolas Roman Herbst, Samuel Kounev, and Ralf Reussner. Elasticity in Cloud Computing: What It Is, and What It Is Not. In *Proceedings of the 10th International Conference on Autonomic Computing (ICAC 13)*, pages 23–27, San Jose, CA, 2013. USENIX. ISBN 978-1-931971-02-7.

[3] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing*, 11(2):430–447, March 2018. ISSN 1939-1374. doi: 10.1109/TSC.2017.2711009.

[4] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic Vertical Elasticity of Docker Containers with ElasticDocker. In *2017 IEEE 10th International Conference on Cloud Computing (CLOUD)*, pages 472–479, June 2017. doi: 10.1109/CLOUD.2017.67.

[5] Yahya Al-Dhuraibi, Faiez Zalila, Nabil Djarallah, and Philippe Merle. Coordinating Vertical Elasticity of both Containers and Virtual Machines. In *International Conference on Cloud Computing and Services Science - CLOSER 2018*, Mars 2018.

[6] Yahya Al-Dhuraibi, Faiez Zalila, Nabil Djarallah, and Philippe Merle. Model Driven Elasticity Management with OCCI. *submitted to IEEE Transactions on Cloud Computing, June 15*, 2018. first feedback major revision.

[7] Fawaz Paraiso, Stéphanie Challita, Yahya Al-Dhuraibi, and Philippe Merle. Model-Driven Management of Docker Containers. In *9th IEEE International Conference on Cloud Computing (CLOUD)*, pages 718–725, San Francisco, United States, June 2016.

[8] C. Pahl. Containerization and the PaaS Cloud. *IEEE Cloud Computing*, 2(3): 24–31, May 2015. ISSN 2325-6095. doi: 10.1109/MCC.2015.51.

[9] Emanuel Ferreira Coutinho, Flávio Rubens de Carvalho Sousa, Paulo Antonio Leal Rego, Danielo Gonçalves Gomes, and José Neuman de Souza. Elasticity in cloud computing: a survey. *annals of telecommunications - annales des télécommunications*, 70(7):289–309, 2015. ISSN 1958-9395. doi: 10.1007/s12243-014-0450-7.

[10] Guilherme Galante and Luis Carlos E. de Bona. A Survey on Cloud Computing Elasticity. In *Proceedings of the 2012 IEEE/ACM Fifth International Conference*

on Utility and Cloud Computing, UCC '12, pages 263–270, Washington, DC, USA, 2012. IEEE Computer Society. ISBN 978-0-7695-4862-3. doi: 10.1109/UCC.2012.30.

[11] Sebastian Lehrig, Hendrik Eikerling, and Steffen Becker. Scalability, Elasticity, and Efficiency in Cloud Computing: A Systematic Literature Review of Definitions and Metrics. In *Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, QoSA '15*, pages 83–92, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3470-9. doi: 10.1145/2737182.2737185.

[12] Junliang Chen, Chen Wang, Bing Bing Zhou, Lei Sun, Young Choon Lee, and Albert Y. Zomaya. Tradeoffs Between Profit and Customer Satisfaction for Service Provisioning in the Cloud. In *Proceedings of the 20th International Symposium on High Performance Distributed, HPDC'11*, pages 229–238, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0552-5. doi: 10.1145/1996130.1996161.

[13] Stéphane Genaud and Julien Gossa. Cost-wait Trade-offs in Client-side Resource Provisioning with Elastic Clouds. In *4th IEEE International Conference on Cloud Computing (CLOUD 2011)*, pages 1–8, Washington, United States, July 2011.

[14] Sadeka Islam, Kevin Lee, Alan Fekete, and Anna Liu. How a Consumer Can Measure Elasticity for Cloud Platforms. In *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering, ICPE '12*, pages 85–96, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1202-8. doi: 10.1145/2188286.2188301.

[15] Michael Kuperberg, Nikolas Roman Herbst, Joakim Gunnarson von Kistowski, and Ralf Reussner. Defining and Quantifying Elasticity of Resources in Cloud Computing and Scalable Platforms. Technical report, Karlsruhe Institute of Technology (KIT), Am Fasanengarten 5, 76131 Karlsruhe, Germany, 2011.

[16] Wei Ai, Kenli Li, Shenglin Lan, Fan Zhang, Jing Mei, Keqin Li, and Rajkumar Buyya. On Elasticity Measurement in Cloud Computing. *Scientific Programming*, 2016:13, 2016.

[17] Guilherme Galante and Luis Carlos Erpen De Bona. A programming-level approach for elasticizing parallel scientific applications. *Journal of Systems and Software*, 110: 239 – 252, 2015. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2015.08.051.

[18] Amro Najjar, Xavier Serpaggi, Christophe Gravier, and Olivier Boissier. Survey of Elasticity Management Solutions in Cloud Computing. In *Continued Rise of the Cloud*, pages 235–263. Springer, 2014.

[19] Athanasios Naskos, Anastasios Gounaris, and Spyros Sioutas. Cloud Elasticity: A Survey. In *Algorithmic Aspects of Cloud Computing*, pages 151–167. Springer, 2016.

[20] Hamoun Ghanbari, Bradley Simmons, Marin Litoiu, Cornel Barna, and Gabriel Iszlai. Optimal Autoscaling in a IaaS Cloud. In *Proceedings of the 9th International Conference on Autonomic Computing, ICAC '12*, pages 173–178, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1520-3. doi: 10.1145/2371536.2371567.

[21] ProfitBricks. Website https://www.profitbricks.com/why-profitbricks.

[22] CloudSigma. Website https://www.cloudsigma.com/features/.

[23] Suvendu Chandan Nayak and Chitaranjan Tripathy. Deadline sensitive lease scheduling in cloud computing environment using AHP. *Journal of King Saud University - Computer and Information Sciences*, 2016. ISSN 1319-1578. doi: https://doi.org/10.1016/j.jksuci.2016.05.003.

[24] Hui Wang, Huaglory Tianfield, and Quentin Mair. Auction Based Resource Allocation in Cloud Computing. *Multiagent Grid Syst.*, 10(1):51–66, January 2014. ISSN 1574-1702. doi: 10.3233/MGS-140215.

[25] Microsoft. Website https://azure.microsoft.com/en-us/.

[26] Amazon. Website http://aws.amazon.com.

[27] Nedeljko Vasić, Dejan Novaković, Svetozar Miučin, Dejan Kostić, and Ricardo Bianchini. DejaVu: Accelerating Resource Allocation in Virtualized Environments. In *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS XVII*, pages 423–436, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-0759-8. doi: 10.1145/2150976.2151021.

[28] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic VM for cloud resources provisioning optimization. In *Advances in Computing and Communications*, pages 431–445. Springer, 2011.

[29] P. Marshall, K. Keahey, and T. Freeman. Elastic Site: Using Clouds to Elastically Extend Site Resources. In *2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing (CCGrid)*, pages 43–52, May 2010. doi: 10.1109/CCGRID.2010.80.

[30] Rackpace. Website http://www.rackspace.com.

[31] Rightscale. Website http://www.rightscale.com.

[32] Nilabja Roy, Abhishek Dubey, and Aniruddha Gokhale. Efficient autoscaling in the cloud using predictive models for workload forecasting. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 500–507. IEEE, 2011.

[33] Scalr. Website http://www.scalr.com.

[34] Upendra Sharma, Prashant Shenoy, Sambit Sahu, and Anees Shaikh. A cost-aware elasticity provisioning system for the cloud. In *2011 31st International Conference on Distributed Computing Systems (ICDCS)*, pages 559–570. IEEE, 2011.

[35] Zhiming Shen, Sethuraman Subbiah, Xiaohui Gu, and John Wilkes. CloudScale: Elastic Resource Scaling for Multi-tenant Cloud Systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing, SOCC '11*, pages 5:1–5:14, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0976-9. doi: 10.1145/2038916.2038921.

[36] Philipp Hoenisch, Ingo Weber, Stefan Schulte, Liming Zhu, and Alan Fekete. Four-Fold Auto-Scaling on a Contemporary Deployment Platform Using Docker Containers. In *Service-Oriented Computing*, pages 316–323. Springer, 2015.

[37] James Hadley, Yehia El Khatib, Gordon Blair, and Utz Roedig. *MultiBox: lightweight containers for vendor-independent multi-cloud deployments*, pages 79–90.

Communications in Computer and Information Science. Springer Verlag, 11 2015. ISBN 9783319250427. doi: 10.1007/978-3-319-25043-4_8.

[38] C. Kan. DoCloud: An elastic cloud platform for Web applications based on Docker. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 478–483, Jan 2016. doi: 10.1109/ICACT.2016.7423440.

[39] Priyanka P Kukade and Geetanjali Kale. Auto-Scaling of Micro-Services Using Containerization. *International Journal of Science and Research (IJSR)*, pages 1960–1963, 2013. ISSN 2319-7064.

[40] Proxmox VE. Website https://www.proxmox.com/en/proxmox-ve.

[41] Salman Taherizadeh and Vlado Stankovski. Dynamic Multi-level Auto-scaling Rules for Containerized Applications. *The Computer Journal*, page bxy043, 2018. doi: 10.1093/comjnl/bxy043. URL http://dx.doi.org/10.1093/comjnl/bxy043.

[42] S. Vijayakumar, Qian Zhu, and G. Agrawal. Dynamic Resource Provisioning for Data Streaming Applications in a Cloud Environment. In *2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 441–448, Nov 2010. doi: 10.1109/CloudCom.2010.95.

[43] Xinwen Zhang, Anugeetha Kunjithapatham, Sangoh Jeong, and Simon Gibbs. Towards an Elastic Application Model for Augmenting the Computing Capabilities of Mobile Devices with Cloud Computing. *Mob. Netw. Appl.*, 16(3):270–284, June 2011. ISSN 1383-469X. doi: 10.1007/s11036-011-0305-7.

[44] Iulian Neamtiu. Elastic executions from inelastic programs. In *Proceedings of the 6th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 178–183. ACM, 2011.

[45] D. Rajan, A. Canino, Jesus A. Izaguirre, and D. Thain. Converting a High Performance Application to an Elastic Cloud Application. In *2011 IEEE Third International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 383–390, Nov 2011. doi: 10.1109/CloudCom.2011.58.

[46] Rodrigo N. Calheiros, Christian Vecchiola, Dileban Karunamoorthy, and Rajkumar Buyya. “The Aneka platform and QoS-driven resource provisioning for elastic applications on hybrid Clouds “. *Future Generation Computer Systems*, 28(6):861 – 870, 2012. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2011.07.005.

[47] AzureWatch. Website http://www.paraleap.com/azurewatch/.

[48] Enda Barrett, Enda Howley, and Jim Duggan. Applying reinforcement learning towards automating resource allocation and application scalability in the cloud. *Concurrency and Computation: Practice and Experience*, 25(12):1656–1674, 2013.

[49] T. Knauth and C. Fetzer. Scaling Non-elastic Applications Using Virtual Machines. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 468–475, July 2011. doi: 10.1109/CLOUD.2011.77.

[50] Germán Moltó, Miguel Caballer, Eloy Romero, and Carlos de Alfonso. Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. *Procedia Computer Science*, 18:159–168, 2013.

[51] A. Ashraf, B. Byholm, and I. Porres. CRAMP: Cost-efficient Resource Allocation for Multiple web applications with Proactive scaling. In *2012 IEEE 4th International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 581–586, Dec 2012. doi: 10.1109/CloudCom.2012.6427605.

[52] H. Fernandez, G. Pierre, and T. Kielmann. Autoscaling Web Applications in Heterogeneous Cloud Infrastructures. In *2014 IEEE International Conference on Cloud Engineering (IC2E)*, pages 195–204, March 2014. doi: 10.1109/IC2E.2014.25.

[53] Soguy Mak-Kare Gueye, Noel De Palma, Eric Rutten, Alain Tchana, and Nicolas Berthier. Coordinating self-sizing and self-repair managers for multi-tier systems. *Future Generation Computer Systems*, 35:14 – 26, 2014. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2013.12.037.

[54] Waheed Iqbal, Matthew N. Dailey, David Carrera, and Paul Janecek. Adaptive resource provisioning for read intensive multi-tier applications in the cloud. *Future Generation Computer Systems*, 27(6):871 – 879, 2011. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2010.10.016.

[55] Evangelia Kalyvianaki, Themistoklis Charalambous, and Steven Hand. Self-adaptive and Self-configured CPU Resource Provisioning for Virtualized Servers Using Kalman Filters. In *Proceedings of the 6th International Conference on Autonomic Computing, ICAC '09*, pages 117–126, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: 10.1145/1555228.1555261.

[56] Pankaj Deep Kaur and Inderveer Chana. A resource elasticity framework for qos-aware execution of cloud applications. *Future Generation Computer Systems*, 37:14 – 25, 2014. ISSN 0167-739X. doi: http://dx.doi.org/10.1016/j.future.2014.02.018.

[57] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. soCloud: a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds. *Computing*, 98(5):539–565, 2016. ISSN 1436-5057. doi: 10.1007/s00607-014-0421-x.

[58] G. A. Moreno, J. Cámara, D. Garlan, and B. Schmerl. Efficient Decision-Making under Uncertainty for Proactive Self-Adaptation. In *2016 IEEE International Conference on Autonomic Computing (ICAC)*, pages 147–156, July 2016. doi: 10.1109/ICAC.2016.59.

[59] Petr Sobeslavsky. *Elasticity in Cloud Computing*. PhD thesis, Joseph Fourier University, ENSIMAG, 2011.

[60] Anca Iordache, Christine Morin, Nikos Parlavantzas, Eugen Feller, and Pierre Riteau. Resilin: Elastic MapReduce over Multiple Clouds. In *13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 261–268, Delft, Netherlands, May 2013. ACM.

[61] Rui Han, Moustafa M Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Generation Computer Systems*, 32:82–98, 2014.

[62] Luis M. Vaquero, Daniel Morán, Fermín Galán, and Jose M. Alcaraz-Calero. Towards Runtime Reconfiguration of Application Control Policies in the Cloud.

Journal of Network and Systems Management, 20(4):489–512, December 2012. ISSN 1064-7570. doi: 10.1007/s10922-012-9251-3.

[63] Sean Barker, Yun Chi, Hakan Hacigümüs, Prashant Shenoy, and Emmanuel Cecchet. ShuttleDB: Database-Aware Elasticity in the Cloud. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 33–43, Philadelphia, PA, June 2014. USENIX Association. ISBN 978-1-931971-11-9.

[64] Leander Beernaert, Miguel Matos, Ricardo Vilaça, and Rui Oliveira. Automatic Elasticity in OpenStack. In *Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management, SDMCMM '12*, pages 2:1–2:6, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1615-6. doi: 10.1145/2405186.2405188.

[65] Francisco Cruz, Francisco Maia, Miguel Matos, Rui Oliveira, João Paulo, José Pereira, and Ricardo Vilaça. MeT: Workload Aware Elasticity for NoSQL. In *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys '13*, pages 183–196, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-1994-2. doi: 10.1145/2465351.2465370.

[66] E. Kassela, C. Boumpouka, I. Konstantinou, and N. Koziris. Automated workload-aware elasticity of NoSQL clusters in the cloud. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 195–200, Oct 2014. doi: 10.1109/BigData.2014.7004232.

[67] Rui Han, Li Guo, M.M. Ghanem, and Yike Guo. Lightweight Resource Scaling for Cloud Applications. In *2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 644–651, May 2012. doi: 10.1109/CCGrid.2012.52.

[68] Pooyan Jamshidi, Aakash Ahmad, and Claus Pahl. Autonomic resource provisioning for cloud-based software. In *Proceedings of the 9th International Symposium on Software Engineering for Adaptive and Self-Managing Systems*, pages 95–104. ACM, 2014.

[69] Ioannis Konstantinou, Evangelos Angelou, Dimitrios Tsoumakos, Christina Boumpouka, Nectarios Koziris, and Spyros Sioutas. TIRAMOLA: Elastic NoSQL Provisioning Through a Cloud Management Platform. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, SIGMOD '12*, pages 725–728, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1247-9. doi: 10.1145/2213836.2213943.

[70] Laura R Moore, Kathryn Bean, and Tariq Ellahi. A coordinated reactive and predictive approach to cloud elasticity. *CLOUD COMPUTING 2013: The Fourth International Conference on Cloud Computing, GRIDs, and Virtualization*, pages 87–92, 2013.

[71] Moore, Laura R. and Bean, Kathryn and Ellahi, Tariq. Transforming Reactive Auto-scaling into Proactive Auto-scaling. In *Proceedings of the 3rd International Workshop on Cloud Data and Platforms, CloudDP '13*, pages 7–12, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2075-7. doi: 10.1145/2460756.2460758.

[72] F. J. A. Morais, F. V. Brasileiro, R. V. Lopes, R. A. Santos, W. Satterfield, and L. Rosa. Autoflex: Service Agnostic Auto-scaling Framework for IaaS Deployment Models. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 42–49, May 2013. doi: 10.1109/CCGrid.2013.74.

[73] Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *Proceedings of the USENIX International Conference on Automated Computing (ICAC'13). San Jose, CA*, pages 69–82, 2013.

[74] P. Di Sanzo, D. Rughetti, B. Ciciani, and F. Quaglia. Auto-tuning of Cloud-Based In-Memory Transactional Data Grids via Machine Learning. In *2012 Second Symposium on Network Cloud Computing and Applications (NCCA)*, pages 9–16, Dec 2012. doi: 10.1109/NCCA.2012.20.

[75] Marco Serafini, Essam Mansour, Ashraf Aboulnaga, Kenneth Salem, Taha Rafiq, and Umar Farooq Minhas. Accordion: Elastic scalability for database systems supporting distributed transactions. *Proceedings of the VLDB Endowment*, 7(12): 1035–1046, 2014.

[76] D. Serrano, S. Bouchenak, Y. Kouki, T. Ledoux, J. Lejeune, J. Sopena, L. Arantes, and P. Sens. Towards QoS-Oriented SLA Guarantees for Online Cloud Services. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 50–57, May 2013. doi: 10.1109/CCGrid.2013.66.

[77] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 285–294. IEEE, 2012.

[78] Luciano Baresi, Sam Guinea, Alberto Leva, and Giovanni Quattrocchi. A Discrete-time Feedback Controller for Containerized Cloud Applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016*, pages 217–228, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-4218-6. doi: 10.1145/2950290.2950328.

[79] Gabriel A. Moreno, Javier Cámara, David Garlan, and Bradley Schmerl. Proactive Self-adaptation Under Uncertainty: A Probabilistic Model Checking Approach. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2015*, pages 1–12, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3675-8. doi: 10.1145/2786805.2786853.

[80] S. Dutta, S. Gera, A. Verma, and B. Viswanathan. SmartScale: Automatic Application Scaling in Enterprise Clouds. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 221–228, June 2012. doi: 10.1109/CLOUD.2012.12.

[81] AWS Spot Instances. Website https://aws.amazon.com/ec2/spot/.

[82] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N Calheiros. InterCloud: Utility-oriented federation of cloud computing environments for scaling of application services. In *Algorithms and architectures for parallel processing*, pages 13–31. Springer, 2010.

[83] A. da Silva Dias, L. H. V. Nakamura, J. C. Estrella, R. H. C. Santana, and M. J. Santana. Providing IaaS resources automatically through prediction and monitoring approaches. In *2014 IEEE Symposium on Computers and Communication (ISCC)*, pages 1–7, June 2014. doi: 10.1109/ISCC.2014.6912590.

[84] Diego Perez-Palacin, Raffaela Mirandola, and Radu Calinescu. Synthesis of Adaptation Plans for Cloud Infrastructure with Hybrid Cost Models. In *Proceedings of the 2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications, SEAA '14*, pages 443–450, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5795-8. doi: 10.1109/SEAA.2014.57.

[85] D. Villela, P. Pradhan, and D. Rubenstein. Provisioning servers in the application tier for e-commerce systems. In *Twelfth IEEE International Workshop on Quality of Service, IWQOS 2004*, pages 57–66, June 2004. doi: 10.1109/IWQOS.2004.1309357.

[86] A. Najjar, X. Serpaggi, C. Gravier, and O. Boissier. Multi-agent Negotiation for User-centric Elasticity Management in the Cloud. In *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*, pages 357–362, Dec 2013. doi: 10.1109/UCC.2013.104.

[87] Bo An, Victor Lesser, David Irwin, and Michael Zink. Automated Negotiation with Decommitment for Dynamic Resource Allocation in Cloud Computing. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1, AAMAS '10*, pages 981–988, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems. ISBN 978-0-9826571-1-9.

[88] Anton Beloglazov and Rajkumar Buyya. Adaptive Threshold-based Approach for Energy-efficient Consolidation of Virtual Machines in Cloud Data Centers. In *Proceedings of the 8th International Workshop on Middleware for Grids, Clouds and e-Science, MGC '10*, pages 4:1–4:6, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0453-5. doi: 10.1145/1890799.1890803.

[89] Hua-Jun Hong, De-Yu Chen, Chun-Ying Huang, Kuan-Ta Chen, and Cheng-Hsin Hsu. Placing virtual machines to optimize cloud gaming experience. *IEEE Transactions on Cloud Computing*, 3(1):42–53, 2015.

[90] Kate Keahey, Patrick Armstrong, John Bresnahan, David LaBissoniere, and Pierre Riteau. Infrastructure Outsourcing in Multi-cloud Environment. In *Proceedings of the 2012 Workshop on Cloud Services, Federation, and the 8th Open Cirrus Summit, FederatedClouds '12*, pages 33–38, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1754-2. doi: 10.1145/2378975.2378984.

[91] Fawaz Paraiso, Philippe Merle, and Lionel Seinturier. Managing Elasticity Across Multiple Cloud Providers. In *Proceedings of the 2013 International Workshop on Multi-cloud Applications and Federated Clouds, MultiCloud '13*, pages 53–60, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2050-4. doi: 10.1145/2462326.2462338.

[92] T.C. Chieu, A. Mohindra, A.A. Karve, and A. Segal. Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment. In *IEEE International*

Conference on e-Business Engineering ICEBE '09, pages 281–286, Oct 2009. doi: 10.1109/ICEBE.2009.45.

[93] X. Dutreilh, N. Rivierre, A. Moreau, J. Malenfant, and I. Truck. From Data Center Resource Allocation to Control Theory and Back. In *2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pages 410–417, July 2010. doi: 10.1109/CLOUD.2010.55.

[94] Masum Z Hasan, Edgar Magana, Alexander Clemm, Lew Tucker, and Sree Lakshmi D Gudreddi. Integrated and autonomic cloud resource scaling. In *Network Operations and Management Symposium (NOMS)*, pages 1327–1334. IEEE, 2012.

[95] Michael Maurer, Ivona Brandic, and Rizos Sakellariou. Enacting SLAs in Clouds Using Rules. In *Proceedings of the 17th International Conference on Parallel Processing, Euro-Par'11 - Volume Part I*, pages 455–466, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-23399-9.

[96] A. Naskos, E. Stachtiari, A. Gounaris, P. Katsaros, D. Tsoumakos, I. Konstantinou, and S. Sioutas. Dependable Horizontal Scaling Based on Probabilistic Model Checking. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 31–40, May 2015. doi: 10.1109/CCGrid.2015.91.

[97] Brabra Hayet, Achraf Mtibaa, Walid Gaaloul, and Boualem Benatallah. Model-Driven Elasticity for Cloud Resources. In *2018 International Conference on Advanced Information Systems Engineering (CAiSE'2018)*, pages 187–202, June 2018.

[98] Harold C. Lim, Shivnath Babu, and Jeffrey S. Chase. Automated Control for Elastic Storage. In *Proceedings of the 7th International Conference on Autonomic Computing, ICAC '10*, pages 1–10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0074-2. doi: 10.1145/1809049.1809051.

[99] Harold C. Lim, Shivnath Babu, Jeffrey S. Chase, and Sujay S. Parekh. Automated Control in Cloud Computing: Challenges and Opportunities. In *Proceedings of the 1st Workshop on Automated Control for Datacenters and Clouds, ACDC '09*, pages 13–18, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-585-7. doi: 10.1145/1555271.1555275.

[100] Xavier Dutreilh, Sergey Kirgizov, Olga Melekhova, Jacques Malenfant, Nicolas Rivierre, and Isis Truck. Using reinforcement learning for autonomic resource allocation in clouds: Towards a fully automated workflow. In *Seventh International Conference on Autonomic and Autonomous Systems ICAS*, pages 67–74, 2011.

[101] Bhuvan Urgaonkar, Prashant Shenoy, Abhishek Chandra, Pawan Goyal, and Timothy Wood. Agile Dynamic Provisioning of Multi-tier Internet Applications. *ACM Transactions on Autonomous and Adaptive Systems*, 3(1):1:1–1:39, March 2008. ISSN 1556-4665. doi: 10.1145/1342171.1342172.

[102] Ahmad Al-Shishtawy and Vladimir Vlassov. ElastMan: Elasticity Manager for Elastic Key-value Stores in the Cloud. In *Proceedings of the 2013 ACM Cloud and Autonomic Computing Conference, CAC '13*, pages 7:1–7:10, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2172-3. doi: 10.1145/2494621.2494630.

[103] Sang-Min Park and Marty Humphrey. Self-Tuning Virtual Machines for Predictable eScience. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, CCGRID '09*, pages 356–363, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3622-4. doi: 10.1109/CCGRID.2009.84.

[104] Jinhui Huang, Chunlin Li, and Jie Yu. Resource prediction based on double exponential smoothing in cloud computing. In *2nd International Conference on Consumer Electronics Communications and Networks (CECNet)*, pages 2056–2060, April 2012. doi: 10.1109/CECNet.2012.6201461.

[105] H. Mi, H. Wang, G. Yin, Y. Zhou, D. Shi, and L. Yuan. Online Self-Reconfiguration with Performance Guarantee for Energy-Efficient Large-Scale Cloud Computing Data Centers. In *2010 IEEE International Conference on Services Computing (SCC)*, pages 514–521, July 2010.

[106] Zhenhuan Gong, Xiaohui Gu, and J. Wilkes. Press: Predictive elastic resource scaling for cloud systems. In *2010 International Conference on Network and Service Management (CNSM)*, pages 9–16, Oct 2010. doi: 10.1109/CNSM.2010.5691343.

[107] S. Khatua, A. Ghosh, and N. Mukherjee. Optimizing the utilization of virtual resources in Cloud environment. In *2010 IEEE International Conference on Virtual Environments Human-Computer Interfaces and Measurement Systems (VECIMS)*, pages 82–87, Sept 2010.

[108] Sadeka Islam, Jacky Keung, Kevin Lee, and Anna Liu. Empirical Prediction Models for Adaptive Resource Provisioning in the Cloud. *Future Generation Computer Systems*, 28(1):155–162, January 2012. ISSN 0167-739X. doi: 10.1016/j.future.2011.05.027.

[109] Jonathan Kupferman, Jeff Silverman, Patricio Jara, and Jeff Browne. Scaling into the cloud. *CS270-advanced operating systems*, 2009.

[110] Eddy Caron, Frédéric Desprez, and Adrian Muresan. Forecasting for Cloud computing on-demand resources based on pattern matching. Research Report RR-7217, INRIA, July 2010.

[111] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, Leyi Wang, and George Yin. VCONF: A Reinforcement Learning Approach to Virtual Machines Auto-configuration. In *Proceedings of the 6th International Conference on Autonomic Computing, ICAC '09*, pages 137–146, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-564-2. doi: 10.1145/1555228.1555263.

[112] G. Tesauro, N.K. Jong, R. Das, and M.N. Bennani. A Hybrid Reinforcement Learning Approach to Autonomic Resource Allocation. In *IEEE International Conference on Autonomic Computing ICAC '06*, pages 65–73, June 2006. doi: 10.1109/ICAC.2006.1662383.

[113] A. Ali-Eldin, J. Tordsson, and E. Elmroth. An adaptive hybrid elasticity controller for cloud infrastructures. In *2012 IEEE Network Operations and Management Symposium (NOMS)*, pages 204–212, April 2012. doi: 10.1109/NOMS.2012.6211900.

[114] Ewnetu Bayuh Lakew, Cristian Klein, Francisco Hernandez-Rodriguez, and Erik Elmroth. Towards Faster Response Time Models for Vertical Elasticity. In *Proceedings of the 2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing*, UCC '14, pages 560–565, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-7881-6. doi: 10.1109/UCC.2014.86.

[115] Qi Zhang, L. Cherkasova, and E. Smirni. A Regression-Based Analytic Model for Dynamic Resource Provisioning of Multi-Tier Applications. In *Fourth International Conference on Autonomic Computing ICAC '07*, pages 27–27, June 2007. doi: 10.1109/ICAC.2007.1.

[116] Lixi Wang, Jing Xu, Ming Zhao, Yicheng Tu, and J.A.B. Fortes. Fuzzy Modeling Based Resource Management for Virtualized Database Systems. In *2011 IEEE 19th International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems (MASCOTS)*, pages 32–42, July 2011. doi: 10.1109/MASCOTS.2011.70.

[117] R.N. Calheiros, R. Ranjan, and R. Buyya. Virtual Machine Provisioning Based on Analytical Performance and QoS in Cloud Computing Environments. In *2011 International Conference on Parallel Processing (ICPP)*, pages 295–304, Sept 2011. doi: 10.1109/ICPP.2011.17.

[118] Che-Lun Hung, Yu-Chen Hu, and Kuan-Ching Li. Auto-Scaling Model for Cloud Computing System. *International Journal of Hybrid Information Technology*, 5(2): 181–186, 2012.

[119] P. Leitner, W. Hummer, B. Satzger, C. Inzinger, and S. Dustdar. Cost-Efficient and Application SLA-Aware Client Side Request Scheduling in an Infrastructure-as-a-Service Cloud. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 213–220, June 2012. doi: 10.1109/CLOUD.2012.21.

[120] Zhipiao Liu, Shangguang Wang, Qibo Sun, Hua Zou, and Fangchun Yang. Cost-aware cloud service request scheduling for SaaS providers. *The Computer Journal*, pages 291–301, 2013.

[121] Di Niu, Hong Xu, Baochun Li, and Shuqiao Zhao. Quality-assured cloud bandwidth auto-scaling for video-on-demand applications. In *2012 Proceedings IEEE INFOCOM*, pages 460–468, March 2012. doi: 10.1109/INFCOM.2012.6195785.

[122] J.M. Tirado, D. Higuero, F. Isaila, and J. Carretero. Predictive data grouping and placement for cloud-based elastic server infrastructures. In *2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 285–294, May 2011. doi: 10.1109/CCGrid.2011.49.

[123] S. Spinner, S. Kounev, Xiaoyun Zhu, Lei Lu, M. Uysal, A. Holler, and R. Griffith. Runtime Vertical Scaling of Virtualized Applications via Online Model Estimation. In *2014 IEEE Eighth International Conference on Self-Adaptive and Self-Organizing Systems (SASO)*, pages 157–166, Sept 2014. doi: 10.1109/SASO.2014.29.

[124] L. Yazdanov and C. Fetzer. Vertical Scaling for Prioritized VMs Provisioning. In *2012 Second International Conference on Cloud and Green Computing (CGC)*, pages 118–125, Nov 2012. doi: 10.1109/CGC.2012.108.

[125] Yufeng Wang, Chiu C. Tan, and Ningfang Mi. Using Elasticity to Improve Inline Data Deduplication Storage Systems. In *Proceedings of the 2014 IEEE International Conference on Cloud Computing, CLOUD '14*, pages 785–792, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-5063-8. doi: 10.1109/CLOUD.2014.109.

[126] Weiming Zhao and Zhenlin Wang. Dynamic Memory Balancing for Virtual Machines. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '09*, pages 21–30, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-375-4. doi: 10.1145/1508293.1508297.

[127] Wesam Dawoud, Ibrahim Takouna, and Christoph Meinel. Elastic virtual machine for fine-grained cloud resource provisioning. In *Global Trends in Computing and Communication Systems*, pages 11–25. Springer, 2012.

[128] Yixin Diao, N. Gandhi, J.L. Hellerstein, S. Parekh, and D.M. Tilbury. Using mimo feedback control to enforce policies for interrelated metrics with application to the apache web server. In *Network Operations and Management Symposium, 2002. NOMS 2002. 2002 IEEE/IFIP*, pages 219–234, 2002. doi: 10.1109/NOMS.2002.1015566.

[129] Dynamic scaling of CPU and RAM. Website https://cwiki.apache.org/confluence/display/CLOUDSTACK/Dynamic+scaling+of+CPU+and+RAM, Visited 2016/02.

[130] S. Farokhi, E.B. Lakew, C. Klein, I. Brandic, and E. Elmroth. Coordinating CPU and Memory Elasticity Controllers to Meet Service Response Time Constraints. In *2015 International Conference on Cloud and Autonomic Computing (ICCAC)*, pages 69–80, Sept 2015. doi: 10.1109/ICCAC.2015.20.

[131] Lei Lu, Xiaoyun Zhu, R. Griffith, P. Padala, A. Parikh, P. Shah, and E. Smirni. Application-driven dynamic vertical scaling of virtual machines in resource pools. In *Network Operations and Management Symposium (NOMS), 2014 IEEE*, pages 1–9, May 2014.

[132] H. Jin, L. Deng, S. Wu, X. Shi, and X. Pan. Live virtual machine migration with adaptive, memory compression. In *2009 IEEE International Conference on Cluster Computing and Workshops, CLUSTER '09*, pages 1–10, Aug 2009. doi: 10.1109/CLUSTR.2009.5289170.

[133] Xiulei Qin, Wei Wang, Wenbo Zhang, Jun Wei, Xin Zhao, and Tao Huang. Elasticat: A load rebalancing framework for cloud-based key-value stores. In *2012 19th International Conference on High Performance Computing (HiPC)*, pages 1–10, Dec 2012. doi: 10.1109/HiPC.2012.6507481.

[134] Rajeswari Sridhar S. Kirthica1. Provisioning rapid elasticity by light-weight live resource migration. *International Journal of Modern Trends in Engineering and Research*, 2:99–106, July 2015. ISSN 2349-9745.

[135] Yi Zhao and Wenlong Huang. Adaptive Distributed Load Balancing Algorithm Based on Live Migration of Virtual Machines in Cloud. In *Proceedings of the 2009 Fifth International Joint Conference on INC, IMS and IDC, NCM '09*, pages 170–

175, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3769-6. doi: 10.1109/NCM.2009.350.

[136] D. Bruneo. A Stochastic Model to Investigate Data Center Performance and QoS in IaaS Cloud Computing Systems. *IEEE Transactions on Parallel and Distributed Systems*, 25(3):560–569, March 2014. ISSN 1045-9219. doi: 10.1109/TPDS.2013.67.

[137] Sudipto Das, Shoji Nishimura, Divyakant Agrawal, and Amr El Abbadi. Albatross: Lightweight Elasticity in Shared Storage Databases for the Cloud Using Live Data Migration. *Proc. VLDB Endowment*, 4(8):494–505, May 2011. ISSN 2150-8097. doi: 10.14778/2002974.2002977.

[138] Sijin He, Li Guo, and Yike Guo. Real Time Elastic Cloud Management for Limited Resources. In *2011 IEEE International Conference on Cloud Computing (CLOUD)*, pages 622–629, July 2011. doi: 10.1109/CLOUD.2011.47.

[139] Michael R Hines and Kartik Gopalan. Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning. In *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 51–60. ACM, 2009.

[140] Stelios Sotiriadis, Nik Bessis, Cristiana Amza, and Rajkumar Buyya. Vertical and Horizontal Elasticity for Dynamic Virtual Machine Reconfiguration. *IEEE Transactions on Services Computing*, PP(99), 2016. ISSN 1939-1374. doi: 10.1109/TSC.2016.2634024.

[141] Nicolo M Calcavecchia, Bogdan A Caprarescu, Elisabetta Di Nitto, Daniel J Dubois, and Dana Petcu. DEPAS: a decentralized probabilistic algorithm for auto-scaling. *Computing*, 94(8-10):701–730, 2012.

[142] T.C. Chieu and Hoi Chan. Dynamic Resource Allocation via Distributed Decisions in Cloud Environment. In *8th IEEE International Conference on e-Business Engineering (ICEBE)*, pages 125–130, Oct 2011. doi: 10.1109/ICEBE.2011.45.

[143] Melanie Siebenhaar, The An Binh Nguyen, Ulrich Lampe, Dieter Schuller, and Ralf Steinmetz. Concurrent Negotiations in Cloud-based Systems. In *Proceedings of the 8th International Conference on Economics of Grids, Clouds, Systems, and Services, GECON'11*, pages 17–31, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-28674-2. doi: 10.1007/978-3-642-28675-9_2.

[144] Seokho Son and Kwang Mong Sim. A price-and-time-slot-negotiation mechanism for cloud service reservations. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 42(3):713–728, 2012.

[145] G. Copil, D. Moldovan, H. L. Truong, and S. Dustdar. On Controlling Cloud Services Elasticity in Heterogeneous Clouds. In *2014 IEEE/ACM 7th International Conference on Utility and Cloud Computing (UCC)*, pages 573–578, Dec 2014. doi: 10.1109/UCC.2014.88.

[146] Google App Engine. Website https://cloud.google.com/appengine/.

[147] T. C. Chieu, A. Mohindra and A. A. Karve. Scalability and Performance of Web Applications in a Compute Cloud. In *2011 IEEE 8th International Conference on*

e-Business Engineering (ICEBE), pages 317–323, Oct 2011. doi: 10.1109/ICEBE.2011.63.

[148] Nikolay Grozev and Rajkumar Buyya. Multi-cloud provisioning and load distribution for three-tier applications. *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, 9(3):13, 2014.

[149] Sebastian Moller and Alexander Raake. Quality of experience. *New York, US: Springer*, 2014.

[150] T. Hobfeld, R. Schatz, M. Varela, and C. Timmerer. Challenges of QoE Management for Cloud Applications. *IEEE Communications Magazine*, 50(4):28–36, April 2012. ISSN 0163-6804. doi: 10.1109/MCOM.2012.6178831.

[151] Francesc D Munoz-Escoı and José M Bernabeu-Aubán. A Survey on Elasticity Management in the PaaS Service Model. *Technical Report ITI-SIDI-2015/002, Universitat Politècnica de Valeǹcia, Spain.*

[152] E. Kafetzakis, H. Koumaras, M. A. Kourtis, and V. Koumaras. QoE4CLOUD: A QoE-driven multidimensional framework for cloud environments. In *2012 International Conference on Telecommunications and Multimedia (TEMU)*, pages 77–82, July 2012. doi: 10.1109/TEMU.2012.6294736.

[153] A. Najjar, C. Gravier, X. Serpaggi, and O. Boissier. Modeling User Expectations Satisfaction for SaaS Applications Using Multi-agent Negotiation. In *2016 IEEE/WIC/ACM International Conference on Web Intelligence (WI)*, pages 399–406, Oct 2016. doi: 10.1109/WI.2016.0062.

[154] Datapipe. Website https://www.datapipe.com/gogrid/.

[155] Paul Marshall, Henry Tufo, and Kate Keahey. Provisioning policies for elastic computing environments. In *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, pages 1085–1094. IEEE, 2012.

[156] Linlin Wu, Saurabh Kumar Garg, and Rajkumar Buyya. SLA-based admission control for a Software-as-a-Service provider in Cloud computing environments. *Journal of Computer and System Sciences*, 78(5):1280–1299, 2012.

[157] Tania Lorido-Botran Jose Miguel-Alonso and Jose A. Lozano. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. *Journal Grid Computing*, 12(4):559–592, December 2014. ISSN 1570-7873. doi: 10.1007/s10723-014-9314-7.

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

[159] Y. Kuno, K. Nii, and S. Yamaguchi. A Study on Performance of Processes in Migrating Virtual Machines. In *2011 Tenth International Symposium on Autonomous Decentralized Systems*, pages 567–572, March 2011. doi: 10.1109/ISADS.2011.79.

[160] D. Kapil, E. S. Pilli, and R. C. Joshi. Live virtual machine migration techniques: Survey and research challenges. In *2013 IEEE 3rd International Advance Computing Conference (IACC)*, pages 963–969, Feb 2013.

[161] V V.Vinothina, Dr.R.Sridaran, and Dr.Padmavathi Ganapathi. A Survey on Resource Allocation Strategies in Cloud Computing. *International Journal of Advanced Computer Science and Applications (IJACSA)*, 3(6):97–104, 2012.

[162] M. G. Hafez and M. S. Elgamel. Agent-Based Cloud Computing: A Survey. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 285–292, Aug 2016. doi: 10.1109/FiCloud.2016.48.

[163] D. Talia. Clouds Meet Agents: Toward Intelligent Cloud Services. *IEEE Internet Computing*, 16(2):78–81, March 2012. ISSN 1089-7801. doi: 10.1109/MIC.2012.28.

[164] CloudSim. Website http://www.cloudbus.org/cloudsim/.

[165] Sareh Fotuhi Piraghaj, Amir Vahid Dastjerdi, Rodrigo N Calheiros, and Rajkumar Buyya. ContainerCloudSim: An Environment for Modeling and Simulation of Containers in Cloud Data Centers. *Software: Practice and Experience*, 47(4): 505–521, 2017.

[166] GreenCloud. Website https://greencloud.gforge.uni.lu.

[167] OMNeT++. Website https://omnetpp.org/.

[168] iCanCloud. Website http://www.arcos.inf.uc3m.es/~icancloud/Home.html.

[169] SimGrid. Website http://simgrid.gforge.inria.fr.

[170] EMUSIM. Website http://www.cloudbus.org/cloudsim/emusim/.

[171] Raphaël Bolze, Franck Cappello, Eddy Caron, Michel Daydé, Frédéric Desprez, Emmanuel Jeannot, Yvon Jégou, Stephane Lanteri, Julien Leduc, Noredine Melab, Guillaume Mornet, Raymond Namyst, Pascale Primet, Benjamin Quetier, Olivier Richard, El-Ghazali Talbi, and Iréa Touche. Grid'5000: A Large Scale And Highly Reconfigurable Experimental Grid Testbed. *The International Journal of High Performance Computing Applications*, 20(4):481–494, 2006. doi: 10.1177/1094342006070078. URL https://doi.org/10.1177/1094342006070078.

[172] World Cup 98 Trace. Website http://ita.ee.lbl.gov/html/contrib/WorldCup.html.

[173] ClarkNet. Website http://ita.ee.lbl.gov/html/contrib/ClarkNet-HTTP.html.

[174] Keerthi Bangari and Chittipothula CY Rao. Real Workload Characterization and Synthetic Workload Generation. *IJRET*, 05(05):417—-429, August 2016. ISSN 2319-1136.

[175] Ahmed Ali-Eldin, Johan Tordsson, Erik Elmroth, and Maria Kihl. Workload classification for efficient auto-scaling of cloud resources. Technical report, Technical Report, 2005.[Online]. Available: www8.cs.umu.se/research/uminf/reports/2013/013/part1.pdf, 2005.

[176] Rubbos. Website http://jmob.ow2.org/rubbos.html.

[177] Rubis. Website http://rubis.ow2.org.

[178] TCP-W. Website http://www.tpc.org/tpcw/.

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

[180] YCSB. Website https://en.wikipedia.org/wiki/YCSB.

[181] A. Sangroya, D. Serrano, and S. Bouchenak. Benchmarking Dependability of MapReduce Systems. In *2012 IEEE 31st Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30, Oct 2012. doi: 10.1109/SRDS.2012.12.

[182] fio. Website http://freecode.com/projects/fio.

[183] CloudSuite. Website http://cloudsuite.ch.

[184] Shengsheng Huang, Jie Huang, Jinquan Dai, Tao Xie, and Bo Huang. *The HiBench Benchmark Suite: Characterization of the MapReduce-Based Data Analysis*, pages 209–228. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011. ISBN 978-3-642-19294-4. doi: 10.1007/978-3-642-19294-4_9.

[185] Containers: The Next Big Thing in Cloud? Website http://insights.wired.com/profiles/blogs/what-s-the-next-big-thing-in-cloud.

[186] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *International Symposium on IEEE Performance Analysis of Systems and Software (ISPASS)*, pages 171–172, March 2015. doi: 10.1109/ISPASS.2015.7095802.

[187] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: State of the art and challenges. In *2015 Ninth International Conference on Complex, Intelligent, and Software Intensive Systems (CISIS)*, pages 70–75. IEEE, 2015.

[188] Dirk Merkel. Docker: Lightweight Linux Containers for Consistent Development and Deployment, March 2014. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=2600239.2600241.

[189] An Introduction to CoreOS System Components. Website https://www.digitalocean.com/community/tutorials/an-introduction-to-coreos-system-components.

[190] LXC: What Is LXC. Website https://linuxcontainers.org/lxc/introduction/.

[191] LXD. Website https://linuxcontainers.org/lxd/introduction/.

[192] BSDJails. Website https://www.freebsd.org/cgi/man.cgi?query=jail&format=html.

[193] OpenVZ. Website https://openvz.org/Main_Page.

[194] The Container Ecosystem Project. Website https://sysdig.com/the-container-ecosystem-project/.

[195] Kubernetes. Website http://kubernetes.io/.

[196] Swarm v. Fleet v. Kubernetes v. Mesos. Website http://radar.oreilly.com/2015/10/swarm-v-fleet-v-kubernetes-v-mesos.html.

[197] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, Omega, and Kubernetes. *Commun. ACM*, 59(5):50–57, April 2016. ISSN 0001-0782. doi: 10.1145/2890784.

[198] Containers: Package, ship and run any software as a self-sufficient unit. Website https://coreos.com/using-coreos/containers/.

[199] Apache Mesos. Website http://mesos.apache.org.

[200] Polo Sony, Ivin. *Inter-Cloud application migration and portability using Linux containers for better resource provisioning and interoperability.* PhD thesis, Dublin, National College of Ireland, 2015.

[201] Chen Yang. Checkpoint and Restoration of Micro-service in Docker Containers. *icmii-15*, pages 915—-918, 2015. ISSN 2352-538X.

[202] VPS.NET. Website https://www.vps.net.

[203] Ming Mao and Marty Humphrey. A Performance Study on the VM Startup Time in the Cloud. In *2012 IEEE 5th International Conference on Cloud Computing (CLOUD)*, pages 423–430. IEEE, 2012.

[204] Alessandro Vittorio Papadopoulos, Ahmed Ali-Eldin, Karl-Erik Arzen, Johan Tordsson, and Erik Elmroth. PEAS: A Performance Evaluation Framework for Auto-Scaling Strategies in Cloud Applications. *ACM Trans. Model. Perform. Eval. Comput. Syst.*, 1(4):15:1–15:31, August 2016. ISSN 2376-3639. doi: 10.1145/2930659.

[205] Chenhao Qu, Rodrigo N. Calheiros, and Rajkumar Buyya. Auto-scaling Web Applications in Clouds: A Taxonomy and Survey. *CoRR*, abs/1609.09224, 2016.

[206] Amjad Ullah, Jingpeng Li, Yindong Shen, and Amir Hussain. A control theoretical view of cloud elasticity: taxonomy, survey and challenges. *Cluster Computing*, May 2018. doi: 10.1007/s10586-018-2807-6. URL https://doi.org/10.1007/s10586-018-2807-6.

[207] Docker Inc. What is Docker? Web site https://www.docker.com/what-docker.

[208] Sam Newman. *Building Microservices*. O'Reilly Media, Inc., 2015.

[209] Kyoung-Taek Seo, Hyun-Seo Hwang, Il-Young Moon, Oh-Young Kwon, and Byeong-Jun Kim. Performance Comparison Analysis of Linux Container and Virtual Machine for Building Cloud. *Advanced Science and Technology Letters*, 66:105–111, 2014.

[210] IBM. An Architectural Blueprint for Autonomic Computing. *IBM White Paper*, June 2006.

[211] Raja Appuswamy, Christos Gkantsidis, Dushyanth Narayanan, Orion Hodson, and Antony Rowstron. Scale-up vs Scale-out for Hadoop: Time to Rethink? In *Proceedings of the 4th Annual Symposium on Cloud Computing*, SOCC '13, pages

20:1–20:13, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2428-1. doi: 10.1145/2523616.2523629.

[212] Red Hat, Inc. Web site https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_Linux/6/html/Resource_Management_Guide/sec-cpu.html.

[213] Checkpoint/Restore. Website https://criu.org/Checkpoint/Restore.

[214] Jose Monsalve, Aaron Landwehr, and Michela Taufer. Dynamic CPU Resource Allocation in Containerized Cloud Environments. In *2015 IEEE International Conference on Cluster Computing*, pages 535–536. IEEE, 2015.

[215] Eric A. Brewer. Kubernetes and the Path to Cloud Native. In *Proc. of the Sixth ACM Sym. on Cloud Comp.*, SoCC '15, pages 167–167, New York, USA, 2015. ACM. ISBN 978-1-4503-3651-2. doi: 10.1145/2806777.2809955.

[216] G. Blair, N. Bencomo, and R. B. France. Models@ run.time. *Computer*, 42(10): 22–27, Oct 2009. ISSN 0018-9162. doi: 10.1109/MC.2009.326.

[217] Open Cloud Computing Interface Open Standard. http://occi-wg.org. Accessed: 2017-12-21.

[218] JA Forrester Consulting. Maximize Container Benefits With A Top-Down Approach. Website http://tinyurl.com/redhadContainers, April 2015.

[219] Ralf Nyrén, Andy Edmonds, Alexander Papaspyrou, Thijs Metsch, and Boris Parák. Open Cloud Computing Interface - Core, September 2016.

[220] Jean Parpaillon, Philippe Merle, Olivier Barais, Marc Dutoo, and Fawaz Paraiso. OCCIware - A Formal and Tooled Framework for Managing Everything as a Service. In CEUR, editor, *Projects Showcase @ STAF'15*, volume 1400 of *Proceedings of the Projects Showcase @ STAF'15*, pages 18 – 25, L'Aquila, Italy, July 2015.

[221] Philippe Merle, Olivier Barais, Jean Parpaillon, Noël Plouzeau, and Samir Tata. A Precise Metamodel for Open Cloud Computing Interface. In *8th IEEE International Conference on Cloud Computing (CLOUD 2015)*, pages 852 – 859, New York, United States, June 2015. IEEE. doi: 10.1109/CLOUD.2015.117.

[222] Faiez Zalila, Stéphanie Challita, and Philippe Merle. A Model-Driven Tool Chain for OCCI. In *25th International Conference on Cooperative Information Systems (CoopIS)*, pages 389–409, Rhodes, Greece, October 2017.

[223] Jean Bézivin. On the Unification Power of Models. *Software and System Modeling (SoSym)*, 4(2):171–188, 2005. URL https://hal.archives-ouvertes.fr/hal-00442700.

[224] Jean Bézivin, Richard F. Paige, Uwe Aßmann, Bernhard Rumpe, and Douglas C. Schmidt. Manifesto - model engineering for complex systems. *CoRR*, abs/1409.6591, 2014. URL http://arxiv.org/abs/1409.6591.

[225] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2nd edition, 2009. ISBN 0321331885.

[226] Adam Piórkowski, Aleksander Kempny, Adrian Hajduk, and Jacek Strzelczyk. Load balancing for heterogeneous web servers. In *International Conference on Computer Networks*, pages 189–198. Springer, 2010.

[227] Roundrobin. https://d2c.io/post/haproxy-load-balancer-part-2-backend-section.

[228] Load Balancer algorithms. Website https://d2c.io/post/haproxy-load-balancer-part-2-backend-section-algorithms.

[229] OCCI-WG Andy Edmonds. Open cloud computing interface-infrastructure. *Deliverable GFD*, 184(06):270, 2011.

[230] Tracking Scaling Policy. Website https://docs.aws.amazon.com/autoscaling/ec2/userguide/as-scaling-target-tracking.html.

[231] Cron Expressions. https://docs.oracle.com/cd/E12058_01/doc/doc.1014/e12030/cron_expressions.htm.

[232] B. Zhang, Yahya A. Dhuraibi, Romain Rouvoy, Fawaz Paraiso, and Lionel Seinturier. CloudGC: Recycling Idle Virtual Machines in the Cloud. In *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pages 105–115, April 2017. doi: 10.1109/IC2E.2017.26.

[233] Scott D. Lowe. Best Practices for Oversubscription, 2013. URL https://communities.vmware.com/docs/DOC-34283.

[234] Anshul Gandhi, Parijat Dube, Alexei Karve, Andrzej Kochut, and Li Zhang. Adaptive, Model-driven Autoscaling for Cloud Applications. In *11th International Conference on Autonomic Computing (ICAC 14)*, pages 57–64, Philadelphia, PA, 2014. USENIX Association. ISBN 978-1-931971-11-9.

[235] Sören Frey and Wilhelm Hasselbring. The CloudMIG Approach: Model-Based Migration of Software Systems to Cloud-Optimized Applications. 2012.

[236] Zabbix: Enterprise-class Monitoring Platform. https://www.zabbix.com. [Online; Accessed: 2017-06-02].

[237] Willy TARREAU. HAProxy-The Reliable, High Performance TCP/HTTP Load Balancer, 2013. URL http://www.haproxy.org.

[238] Flask. http://flask.pocoo.org.

[239] Redis. https://redis.io.

[240] Synergy Research Group. Website https://www.srgresearch.com/articles/cloud-revenues-continue-grow-50-top-four-providers-tighten-grip-market, July 27, 2018.