On Monitoring and Analyzing Elastic Cloud Systems

DISSERTATION

zur Erlangung des akademischen Grades

Doktor der Technischen Wissenschaften

eingereicht von

Daniel Moldovan
Matrikelnummer 1227559

an der Fakultät für Informatik
der Technischen Universität Wien

Betreuung: Univ. Prof. Dr. Schahram Dustdar
Zweitbetreuung: Ass. Prof. Dr. Hong-Linh Truong

Diese Dissertation haben begutachtet:

Univ. Prof. Dr. Schahram Dustdar

Prof. Dr. Dr. h. c. Frank Leymann

Wien, 1. Dezember 2015

Daniel Moldovan

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

On Monitoring and Analyzing Elastic Cloud Systems

DISSERTATION

submitted in partial fulfillment of the requirements for the degree of

Doktor der Technischen Wissenschaften

by

Daniel Moldovan
Registration Number 1227559

to the Faculty of Informatics
of TU Wien

Advisor: Univ. Prof. Dr. Schahram Dustdar
Second advisor: Ass. Prof. Dr. Hong-Linh Truong

The dissertation has been reviewed by:

Univ. Prof. Dr. Schahram Dustdar

Prof. Dr. Dr. h. c. Frank Leymann

Vienna, 1st December, 2015

Daniel Moldovan

Erklärung zur Verfassung der Arbeit

Daniel Moldovan
Argentinierstrasse 8 184-1, Wien 1040, Austria

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. Dezember 2015 ______________________________
Daniel Moldovan

Danksagung

Diese Arbeit wurde durch das TU Wien ”Adaptive Distributed Systems” Doktorandenkolleg (von der Österreichischen Industriellenvereinigung gesponsert), und das European Community’s Seventh Framework Programme für Forschung, Technologischer Entwicklung und Demonstration (FP7-ICT) im Rahmen des CELAR FP7-Projektes (grant agreement no. 317790 FP7-ICT-2011-8) finanziert.

Acknowledgements

I first want to thank Univ. Prof. Dr. Schahram Dustdar and Ass. Prof. Dr. Hong-Linh Truong for having me in the TU Wien's Distributed Systems Group and guiding me throughout my PhD. Many thanks go to the current and former members of the Distributed Systems Group for laughing, joking, and sharing ideas.

Special thanks go to Prof. Dr. Dr. h. c. Frank Leymann for his insightful courses and talks on fundamentals of the cloud, and serving as the second reviewer for this thesis.

I also want to thank Prof. Dr. Ioan Salomie and the members of the Distributed Systems Group from Technical University of Cluj-Napoca, where I have had my first contact with research in Computer Science.

There are so many people involved in the development of a person, researcher or otherwise, that is virtually impossible to acknowledge all of them in way that does them justice. Thus, many thanks go to all the people who had impacted my career over time, and who are not explicitly mentioned above.

Finally, my greatest thanks go to my family for their unconditional support.

The research in this dissertation has received funding from TU Wien through the Doctoral College "Adaptive Distributed Systems" co-sponsored by the Federation of Austrian Industries (Industriellenvereinigung), and the European Community's Seventh Framework Programme for research, technological development and demonstration (FP7-ICT) under the CELAR FP7 project, grant agreement number 317790 FP7-ICT-2011-8.

Kurzfassung

Der derzeitige Stand der Technik in Cloud Computing und Virtualisierung ermöglicht die Entwicklung von komplexen elastischen Cloud-Systemen. Die Komponenten solcher Systeme können als virtuelle Container oder virtuelle Maschinen von mehreren Cloud-Anbietern in verschiedensten Ausprägungen gehostet werden. Um solche Systeme zu bauen und sowohl automatisch als auch manuell zu verwalten, müssen einige anspruchsvolle Herausforderungen bewältigt werden. Diese Herausforderungen zeigen sich in den unterschiedlichen Phasen des Lebenszykluses eines solchen Cloud-Systems und umfassen die Bereiche Entwurf, Bereitstellung und Laufzeitkontrolle.

Erstens, für die Bereitstellung eines elastischen Systems, ist es wichtig, die vom Cloud-Anbieter bereitgestellten Angebote auszuwählen die die erforderlichen Elastizitätskriterien unterstützen. Da elastische Systeme sowohl ihre Struktur, als auch die konsumierten Cloud-Dienste zur Laufzeit verändern, ist es wichtig diese nach der Bereitstellung zu überwachen und zu analysieren. Während elastische Systeme aufgrund steigender Leistungsanforderungen hochskalieren, sind die dabei entstehende Kosten der Hauptgrund die verwendeten Ressourcen wieder zu reduzieren. Deshalb ist es wichtig Entwicklern solcher Systeme die Möglichkeit zu bieten die Kosten von Systemen, die in öffentlichen Clouds laufen, zu überwachen und deren Kosteneffizienz zu analysieren.

Der Beitrag dieser Arbeit ist eine Reihe von Konzepten, Techniken und Algorithmen zur Analyse von elastischen Cloud-Systemen. Wir wollen damit deren Entwurf, Implementierung und Laufzeitmanagement verbessern, in dem wir die relevanten Informationen für die jeweiligen Phasen zur Verfügung stellen.

Wir quantifizieren die Elastizitätsfähigkeit von Cloud-Diensten, und bestimmen welche Cloud-Dienste die nötige Elastizität bereitstellen und darüber hinaus die Anforderungen an Ressourcen, Qualität und Kosten erfüllen. Für die Überwachung von elastischen Systemen stellen wir ein Modell, Techniken und unterstützende Werkzeuge vor, die die Komposition von Systemmetriken zur Erlangung der erforderlichen Informationen auf dem geforderten Niveau ermöglichen. Zur Charakterisierung des Verhaltens von elastischen Systemen definieren wir die Konzepte: Elastizität und Pathway, und stellen Algorithmen für deren Bestimmung basierend auf Systemüberwachungsdaten bereit. Weiter konzentrieren wir uns auf die Elastizitätsbeziehungen in Cloud-Systemen. Wir ermöglichen damit verschiedenen Akteuren die Elastizitätsbeziehungen zu verstehen, welche das Laufzeitverhalten von komplexen Cloud-Systemen steuern. Abschließend beschäftigen wir uns mit den Themen der Kostenüberwachung und der Analyse der Kosteneffizienz von komplexen Cloud Systemen die in öffentlichen Clouds laufen. Wir definieren Algorithmen zur Kostenüberwachung elastischer Systeme und bestimmen welche Systemkomponenten kostengünstiger zu skalieren sind, als auch wann skaliert werden sollte. Wir überprüfen unsere Techniken auf einem IoT (Internet der Dinge) System, in dem wir Entwurf, Überwachung und Laufzeitelastizitätskontrolle unterstützen.

Abstract

Considering the current state of the art in cloud computing and virtualization, we are at a point in time in which one can develop complex elastic cloud systems. Such systems can have components/units hosted in virtual containers, inside virtual machines, using different cloud offerings (from IaaS to SaaS), potentially from multiple cloud providers. However, to build such systems, and manage them either automatically or through human-based control, there are several important challenges to be addressed at different stages of the system's life-cycle, from design, to deployment and run-time.

First, for deploying an elastic system, it is crucial to select from the cloud providers the offerings which support the required system elasticity. After deployment, a major concern is monitoring and analyzing elastic systems, as they change their structure and used cloud services at run-time. While elastic systems are scaled-out due to performance requirements, cost is the main driver for scale-in. Thus, developers of such systems require support for monitoring the costs and analyzing cost efficiency of elastic systems running in public clouds.

In this thesis we bring as contribution a series of concepts, techniques and algorithms for analyzing elastic cloud systems. We aim to facilitate their design, deployment, and run-time management, by providing information crucial at each of these stages.

We aid in the design of elastic cloud systems by quantifying the elasticity capabilities of cloud services. We analyze which cloud services provide the necessary elasticity support and fulfill resources, quality, and cost requirements. For monitoring elastic systems we introduce a model, technique and supporting platform for composing system metrics, towards obtaining the required information at the needed level. For characterizing the behavior of elastic systems we define the concepts of elasticity and pathway, and provide algorithms for determining them based on multi-level monitoring information. We further focus on analyzing elasticity relationships in cloud systems, enabling different stakeholders, from developers to elasticity controllers, to understand the elasticity relationships governing the run-time behavior of complex cloud systems. Finally, we address the issue of monitoring costs and analyzing cost efficiency of elastic systems running in public clouds. We introduce a model for capturing the pricing schemes of cloud services. We define algorithms for monitoring costs of elastic systems, and evaluating which system component is more cost efficient to scale-in and when. We evaluate our techniques on a Data-as-a-Service elastic cloud system for IoT, by aiding in its design, monitoring, and run-time elasticity control.

Contents

List of Figures

List of Tables

List of Algorithms

List of publications

The work presented in this thesis is based on research that has been published in the following conference papers and journal articles. For reasons of brevity, these core papers, which build the foundation of this thesis, are listed here once, and will generally not be explicitly referenced again. Parts of these papers are contained in verbatim. Please refer to Appendix A for a full publication list of the author of this thesis.

- **Daniel Moldovan**, Hong-Linh Truong, Schahram Dustdar, *Cost-aware scalability of applications in public clouds*, International Conference on Cloud Engineering, IC2E, IEEE, Berlin, Germany, 4-8 April, 2016, *accepted*

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *MELA: Elasticity Analytics for Cloud Services*, International Journal of Big Data Intelligence, no. 1, vol. 2, 2015, `http://dx.doi.org/10.1504/IJBDI.2015.067569`

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *On Analyzing Elasticity Relationships of Cloud Services*, International Conference on Cloud Computing, CloudCom, IEEE, Singapore, 15-18 December, 2014, `http://dx.doi.org/10.1109/CloudCom.2014.93`

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *QUELLE - a Framework for Accelerating the Development of Elastic Systems*, European Conference on Service-Oriented and Cloud Computing, ESOCC, Springer Berlin Heidelberg, Manchester, UK, 2-4 September, 2014, `http://link.springer.com/section/10.1007/978-3-662-44879-3_7`

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *MELA: Monitoring and Analyzing Elasticity of Cloud Services*, International Conference on Cloud Computing, CloudCom, IEEE, Bristol, UK, 2-5 December, 2013, `http://dx.doi.org/10.1109/CloudCom.2013.18`

CHAPTER 1

Introduction

In this thesis we monitor and analyze elastic cloud systems[1] built on top of a mix of cloud services and proprietary software. We facilitate their design, deployment, and run-time management, by providing information crucial at each of these stages.

1.1 Context

Traditionally, IT systems where deployed on dedicated computing resources, making them tied to the used hardware, and difficult to update and extend. This lead to sub-optimal resources usage [6], and system vulnerability to hardware failure. Virtualization appeared as a solution to this problem, enabling the partition of physical resources between several virtual machines (VM). Virtual machines can still be used in the same way as dedicated hardware. However, their advantage is that they can be consolidated and migrated between hardware resources according to needs [7]. Ecosystems offering such virtualization where called virtualized datacenters [8]. Virtualized datacenters where usually proprietary, belonging to individual organizations and running the systems necessary for the organizations' operation. Using today's terminology, such environments can be considered as "private clouds". However, organizations still exhibited underutilization of hardware resources, and where tied in running their systems only on their proprietary data centers. This hindered the organizations' expansion in other geographical regions. Organizations had to buy the necessary hardware resources, build the data centers, maintain and pay staff for managing them [9]. This can become too expensive and complex to manage for small and medium enterprises (SMEs). A big step towards relieving companies of this complexity came with the Amazon Elastic Compute Cloud (EC2)[2]. Under EC2, Amazon offered their spare computing capacity as virtualized infrastructure, but, for the first time,

[1]In the thesis we use the term "system" to denote any complex system, application, or service built on top of or using cloud offered services.

[2]`https://aws.amazon.com/ec2/`

accessible on demand, over a network, from any corner of the world, and billed per usage. In just a couple of years this new type of computing service offered over a network gained worldwide attention [10], opening new research directions in theoretical and applied computer science. The economic model introduced by Amazon allowed for the complete separation (not only physical, but also geographical) of resource management from the user of virtual resources. This lead to the definition of cloud and cloud computing as a new computing paradigm [11, 12]. Now, companies or individuals could rent computing resources according to their demand, matching what they pay to what they need. This reduced the time to market for SMEs, letting companies focus on their core business by reducing the cost required to buy and maintain computing infrastructure [13, 14]. As more and more enterprises became interested in cloud, cloud users became more diverse, ranging from computer scientists, to people without IT training. In order to support this large array of clients, especially clients without previous of knowledge in infrastructure management, emerging cloud providers started to offer different types of services. Departing from only virtualized infrastructure as a service (now called IaaS), providers offered Platforms as a Service (PaaS) for developing systems directly "in the cloud". Further, Software as a Service (SaaS) has proven useful for clients having no IT experience, just wanting to use particular services for their business, such as data storage, or even business analytics. Each of these service types has advantages and disadvantages. The flexibility of using cloud services decreases from IaaS to SaaS, together with the knowledge required to use them. While an IaaS service exposes a virtualized resource which the user must fully configure (e.g, on a VM install OS and necessary software), for SaaS the user just needs to interact with a provided API.

Benefiting from the ever increasing set of cloud services, a new computing sub-discipline has emerged, *elastic computing*, in which distributed cloud systems are deployed on "top of" and "are using" multiple instances of cloud services. Such systems can scale up/out as long as the workload is high, and scale back in/down when possible, reducing cost while maintaining performance and quality [15]. Crucially, this scaling is performed by reconfiguring/allocating/deallocating instances of the used cloud services, from IaaS to SaaS. Generally, an elastic cloud system has the ability to respond to stress factors originating from different sources such as changes in system usage patterns, hosting clouds properties, or pricing schemes. The system response involves modifying its run-time structure through allocation/deallocation/reconfiguration of cloud services [16, 17]. One of the most important aspects of elastic systems is horizontal scalability, in which the system is capable of replicating parts of itself, to ensure a certain level of operation quality. Inversely, the system can reduce the number of such replicas when no longer needed, as to maintain a good cost per system usage ratio [18, 19]. To this end, such systems have at least some parts/components/units designed to be deployed and run independently on the other instances of the same unit (E.g., separate instances of a web server), making it easier to horizontally scale them. This principle of running multiple instances of system components depending on requirements has led to the emergence of software containers [20, 21]. Software containers (E.g., LinuX Containers[3]) introduce

[3]`https://wiki.archlinux.org/index.php/Linux_Containers`

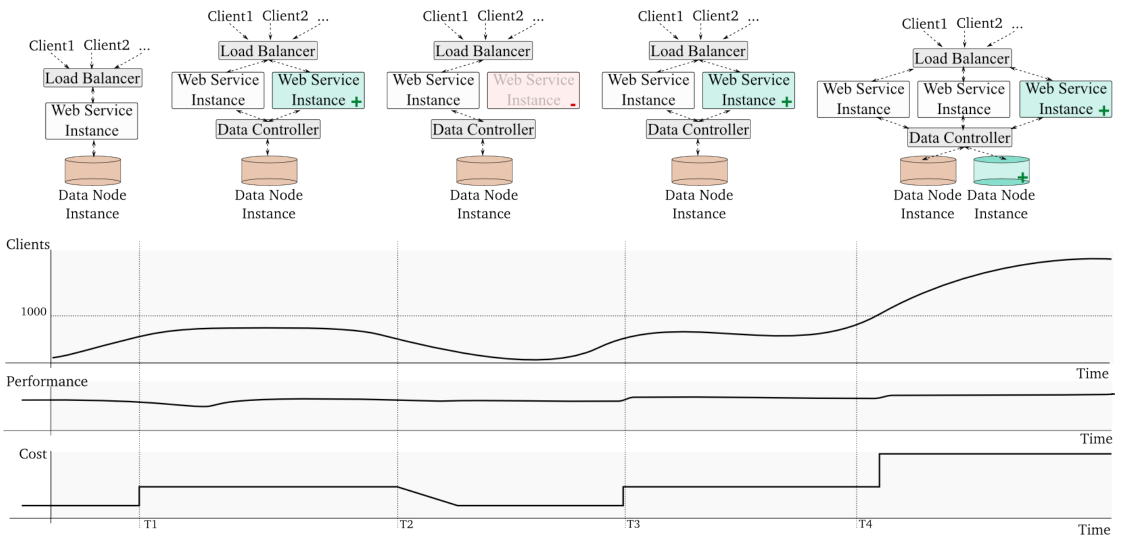


Figure 1.1: Elastic cloud system behavior (adapted from [1])

another abstraction layer over operating system virtualization, each container providing isolation for the software running inside it, from the other containers or host operating system. Multiple containers can run on the same host operating systems, which in turn can be hosted in a virtual, or hardware machine. This enables elastic systems to host new system components/units in dedicated virtual machines, and/or in software containers distributed among virtual and/or physical resources.

Elastic cloud systems can change themselves during run-time, increasing/decreasing computing capacity as needed [15]. Figure 1.1 depicts the behavior of an elastic cloud system relying on web servers reading/writing data from/to a distributed data end. Using a `Load Balancer`, `Web Service` instances can be easily added/removed at run-time. Similarly, the `Data End` is distributed, having a `Data Controller` acting as data access load balancer, and can add/remove `Data Node` instances as needed. An elastic system can start with a lighter initial configuration of virtual resources, and implicitly, with a lower running cost. When the number of clients increases at $T1$, another `Web Service` instance can be added to cope with the increasing number of clients, and thus, increasing the cost of running the system. At time $T2$, when the number of clients decreases, the additional `Web Service` instance can be deallocated, to reduce operation cost. At time $T4$, when the system load is even greater, both `Web Service` and `Data Node` instances could be added, to cope with the load.

Considering the current state of the art in cloud computing and virtualization, we are at a point in time in which one can develop complex elastic cloud systems. Such systems can have components/units hosted in virtual containers, inside virtual machines, using different cloud offerings (from IaaS to SaaS), potentially from multiple cloud providers. However, to build such systems, and manage them either automatically or through human-based control, there are several important challenges to be addressed at different stages of the system's life-cycle, from design, to deployment and run-time control. First, for deploying an elastic system, it is crucial to select from the cloud providers the offerings which support the required run-time system elasticity, i.e., run-time reconfiguration/allocation/deallocation frequency. After deployment, a major concern is monitoring elastic systems, as they change their structure and used cloud services at run-time, to fulfill their requirements. Given the potential complexity of such elastic systems, for run-time control is also important to understand which system units/parts behave properly, and which need to be scaled/reconfigured and when. While elastic systems are scaled-out due to performance requirements, cost is the main driver for scale-in. However, cost of elastic systems running in public clouds is complex, some cloud services having multiple cost elements. E.g., a VM service could be billed both every hour, and separately per each GB of generated I/O. Thus, developers of elastic systems require support for monitoring the costs and analyzing cost efficiency of elastic systems running in public clouds.

The issues mentioned in this section open a broad field of research questions to be solved for providing support in building and controlling elastic cloud systems.

1.2 Roadmap

The rest of this chapter is structured as follows. A detailed problem statement is given in Section 1.3, setting the scope of this thesis. The core research questions are defined in Section 1.4. The major thesis contributions are detailed in Section 1.5. Section 1.5.1 details the thesis organization.

1.3 Problem statement

To increase cloud adoption and foster innovation, we envision a platform enabling developers to design *elastic cloud systems*, deploy them in cloud environments, and manage them at run-time. The platform would reduce the complexity of designing and managing such systems, allowing cloud users to fully take advantage of the benefits of cloud computing. More companies would be able to rapidly build, deploy and manage complex systems offering functionality as a service. Elastic cloud systems could be first built by redesigning existing systems for cloud environments. They could also be developed directly in the cloud using Platform as a Service (PaaS) cloud offerings. Developers could also use Software as a Service (SaaS) cloud offerings, develop their own software components, or employ a combination of these approaches. Elastic systems would expose elasticity capabilities enabling run-time system reconfiguration through allocation/deallocation of cloud services. Elasticity control strategies for system units and whole systems would be designed, to fulfill requirements over system's performance, cost, and resource usage. The strategies would be enforced by full or semi-automatic (under human supervision) elasticity controllers, based on multi-level monitoring information.

Due to the heterogeneity of the available cloud services offered by different cloud providers, one must analyze which services are suitable for each system unit, depending on the system requirements and the envisioned elasticity control. During both design time and run-time of elastic systems, the type of used cloud services can have a different impact the elasticity of the system. For example, the rate at which cloud services can be allocated/deallocated (E.g., hourly, weekly, monthly) can influence the system's control frequency. If a system unit/component is to be reconfigured often by addition/removal of unit/component instances, it should use cloud services (E.g., VM, Network, Storage) which can be deallocated on short notice. Certain cloud providers could also restrict or impose certain combinations of cloud offered services, impacting the run-time availability of the system's elasticity capabilities. Available mechanisms for designing elastic systems such as [22, 23] are often limited to the exact specification of the required cloud services, without considering their elasticity. Currently, a system developer has to manually search through cloud providers, and select the needed services without support in evaluating if their elasticity capabilities support the required elasticity.

Once the system is deployed and running, it can change its used cloud offered services at run-time, by adding/removing service instances depending on requirements. To take appropriate actions, elasticity controllers must determine if a requirement violation originates in a poorly chosen cloud service, resource congestion, or failing system unit. This implies knowledge over the behavior of each system unit. Even more, it implies understanding on how behavioral dependencies present between system units influence the system's behavior with respect to its requirements. Elastic systems can have many units, distributed among different cloud services and even among cloud providers. Controlling such systems requires information about what requirements to enforce for each unit [24, 25]. The structure of elastic systems is dynamic, changing at run-time due to enforced elasticity control. Due to this complexity, it is hard to have all the information required to control elastic systems available beforehand. A simple example of such information is knowing what data end latency ensures the required client-perceived response time. Instead, such information needs to be determined by monitoring and analyzing the run-time behavior of elastic systems.

While elastic systems are usually scaled-out due to performance requirements, cost is the main driver for scale-in actions [26, 27]. Through cost-aware elasticity, controllers should consider the costs of different types of used cloud services, rather than just manipulating their number [28]. However, cost of elastic systems running in public clouds is complex. Cloud services can have multiple cost elements, e.g., a VM service could be billed both every hour, and separately per each GB of generated I/O. Certain costs can be static, such as costs for reserving a cloud service [29, 30, 31]. Other costs can be dynamic, such as discounts for certain service usage levels. Costs of cloud services can also depend on particular service combinations. Additionally, public cloud services are usually billed over pre-defined time and/or usage intervals. This means it might not be cost efficient to deallocate such services at any moment in time, as one might deallocate unused services, but paid in full. For example, paying for a cloud service per hour, but deallocating it after 30 minutes. Due to this cost complexity, developers of elastic systems running in public clouds require support for monitoring costs and developing cost-aware elasticity controllers.

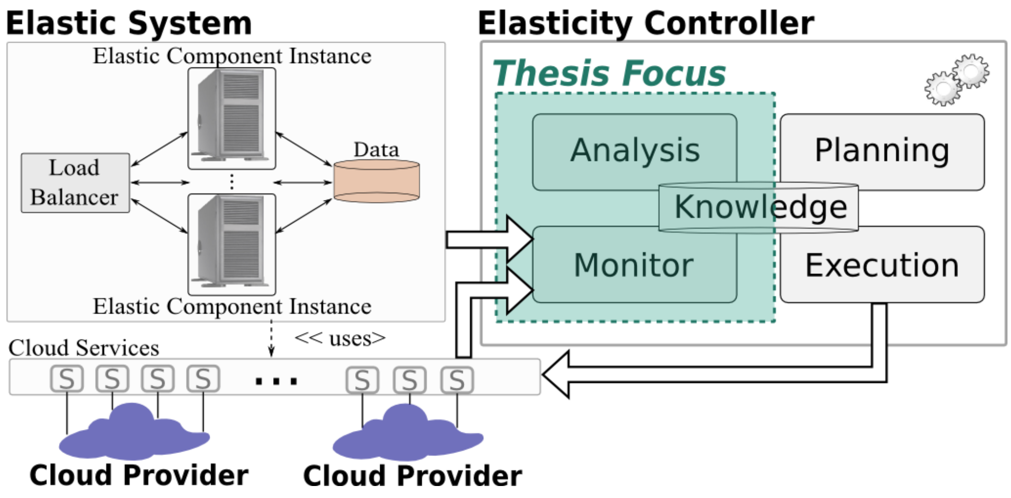


Figure 1.2: Thesis focus in the context of elastic systems

To achieve elasticity, one must understand at each phase of the system's life-cycle, from system design, to deployment, and run-time control, what are the crucial aspects influencing the system's elasticity. However, currently such information is either not available, or insufficient for supporting elasticity. Not solving the above issues can lead to sub-optimal elasticity control, which, relying on insufficient or incomplete data, cannot fully take advantage of the benefits of cloud computing.

To control elastic systems, different types of knowledge are required, both over the system's behavior and the behavior of the used cloud services. How to obtain such knowledge and providing it in a useful manner to elasticity controllers is the objective of this thesis (highlighted in Figure 1.2). We adopt the role of the cloud user, and view cloud providers as black boxes. Cloud providers offer APIs for allocating/deallocating services on-demand and querying their pricing schemes. Cloud users have no access to the inner workings of the employed cloud providers, interacting only with their APIs. System developers are using such cloud services to deploy and run their systems, and have certain requirements over the systems' performance and cost. Elastic systems have controllers able to monitor the system's state and ensure its requirements are fulfilled. Depending on the system's purpose, the control loop might be completely automated, or partially automated involving human supervision.

1.4 Research questions

Addressing the above-mentioned problems, this thesis investigates and answers the following research questions:

- **Question 1**: *How is the run-time elasticity of cloud systems influenced by the cloud services they use?*

Different cloud services might have particular configuration, combination options, and restrictions, depending on individual cloud providers. The used cloud services should provide to elastic systems the necessary control mechanisms for fulfilling their operation requirements. Understanding how the service's combination options and restrictions influence the available system control mechanisms is crucial for designing and managing elastic cloud systems.

- **Question 2**: *How can elastic systems be monitored and analyzed, considering their complexity and dynamic run-time structure?*

  The ability of elastic systems to re-configure individual components or the whole system at run-time generates several issues. Through system scaling, component instances can dynamically appear/disappear and cloud services can be allocated/deallocated dynamically at run-time. If monitoring information is associated only with each service (E.g., VM), it will be lost during scale-in operations which deallocate the service. For controlling elastic systems, monitoring must be able to deal with dynamic system structures, and analyze not only the behavior of individual instances, but also the overall component behavior over all its instances.

- **Question 3**: *How can the behavior of elastic systems be characterized towards aiding in their run-time control?*

  Controlling elastic systems requires understanding of the behavior of each system component with respect to the system's requirements. The behavioral dependencies between system components must also be understood for effective control of elastic systems. However, many times such information is not available beforehand. Thus, starting from potentially generic requirements (E.g., over end to end client-perceived response time), concrete enforceable requirements should be determined for each system component. Additionally, one must understand how different elasticity control actions impact the overall system behavior with respect to its requirements.

- **Question 4**: *How can elastic systems running in public clouds be controlled in a cost efficient manner?*

  Elastic systems can be reconfigured during run-time by allocating, deallocating, or changing used cloud services. However, public cloud providers offer their services under specific pricing schemes, billing them over usage and/or reservation time. This means it might not be cost efficient to deallocate cloud services at any moment in time, without considering their billing cycles. For cost efficient control of elastic systems, controllers must understand which system component instance is cost efficient to deallocate and when. Additionally, optimizing cost of elastic systems requires analysis over the contribution of each system component to the overall system costs.

1.5 Scientific contributions

Addressing the research questions presented in Section 1.4, this thesis makes the following contributions to the state of the art in the monitoring and analysis of elastic cloud systems:

- **Contribution 1**: *Method and algorithms for quantification elasticity of cloud services and recommending cloud services for building elastic systems*

  We define the process of elasticity quantification as a means of evaluating the elasticity support offered by cloud services. In the quantification process we consider both the system's run-time control capabilities and its requirements over cost and performance. We define algorithms and integrate them in a platform for recommending cloud services to be used by elastic systems, ensuring the cloud services do not restrict their run-time elasticity. The contribution is discussed in detail in Chapter 4 and was originally presented in [32].

- **Contribution 2**: *Method, models, and language for multi-level monitoring of elastic systems*

  We address the issue of monitoring elastic systems considering their dynamic rum-time structure. Elastic systems can, depending on requirements, add/remove resources to existing components, or allocate/deallocate component instances using cloud services. In this contribution we introduce a metric composition mechanism for associating monitoring information collected from used cloud services to the system's structure. We define algorithms for extracting higher level information from collected monitoring data. We implement a platform for monitoring elastic cloud systems, providing to elasticity controllers multi-level monitoring information. The contribution is discussed in detail in Chapter 5 and was originally presented in [33] and [1].

- **Contribution 3**: *Concepts of elasticity space and pathway and algorithms for analyzing the behavior of elastic systems*
  Elastic systems are an emerging concept, and there is lack of terminology for describing them, and characterizing their behavior. In this contribution we define new concepts for characterizing the behavior of such systems. We define the concepts of elasticity space and elasticity pathway. We develop algorithms for analyzing the behavior of elastic cloud systems from the whole system level to the underlying virtual infrastructure. The contribution is discussed in detail in Chapter 6 and was originally presented in [33] and [1].

- **Contribution 4**: *Concept of elasticity energy and algorithms for determining behavioral relationships present in elastic systems*
  In this contribution we analyze the behavioral relationships between components

of elastic cloud systems. Knowing relationships enables predictive elasticity control, understanding how system components influence each other. We define the concept of elasticity energy for characterizing behavioral relationships present in elastic cloud systems. We define algorithms for analyzing the relationships influencing the system's behavior with respect to its requirements, from the overall system behavior, to individual system components. The contribution is discussed in detail in Chapter 7 and was originally presented in [34].

- **Contribution 5**: *Method, models, and algorithms for costs analysis and cost-aware control of elastic systems in public clouds*

  Cost is one of the main drivers of elasticity control. While systems are scaled-out due to performance requirements, cost is the main driver for system scale-in. In this contribution we introduce a model for describing the pricing schemes of cloud providers. Based on the pricing schemes and system monitoring information, we define algorithms for evaluating the costs and cost efficiency of elastic systems running in public clouds. We further analyze and recommend which cloud service is cost efficient to deallocate during scale-in operations, and when. The contribution is discussed in detail in Chapter 8 and was originally presented in [35].

1.5.1 Thesis Organization

The remainder of this thesis is structured as follows. Chapter 2 provides background information, and introduces the concepts and terminology used throughout the rest of the thesis. Chapter 3 introduces our case study used to evaluate all thesis contributions. The case study is centered on a Data-as-a-Service elastic cloud system for IoT, which must be designed, deployed, and controlled according to requirements. The five contributions outlined in Section 1.5 are detailed into five main chapters. Chapter 4 details our contribution in recommending cloud services for building elastic systems, introducing our approach in quantifying elasticity of cloud services. Chapter 5 introduces our second contribution, detailing our mechanism for monitoring elastic systems. Chapter 6 covers the third contribution, introducing our algorithms for analyzing the elasticity space and pathway of cloud systems. Chapter 7 introduces our fourth contribution on analyzing and discovering elasticity relationships influencing the behavior of cloud systems. Chapter 8 introduces our approach for evaluating and improving the cost efficiency of elastic systems, detailing the last contribution of the thesis. The five core chapters (Chapters 4, 5, 6, 7, and 8) have a similar structure. They start with a detailed introduction and motivation presenting the context and reason behind each contribution. They contain one or more core sections detailing the contributions, and one evaluation section analyzing the introduced approach. They conclude with remarks specific to each addressed contribution. Chapter 9 presents related work categorized according to the research questions outlined in Section 1.4. Finally, Chapter 10 concludes the thesis, critically discussing the thesis contributions according to the posed research questions, and outlines future work.

CHAPTER 2

Background

In this chapter we introduce the background behind the work done in the thesis. We further introduce the core concepts and terminology which will be used throughout the rest of the thesis.

2.1 Virtualization

Virtualization is a fundamental technology used in cloud computing. According to National Institute of Standards and Technology (NIST)[4], virtualization is the simulation of the software and/or hardware upon which other software runs [36]. Virtualization is further classified by the European Commission in [37] as an essential technological characteristic of clouds which hides the technological complexity from the user and enables enhanced flexibility. Virtualization of computing resources can be full, OS-assisted (paravirtualization) or hardware-assisted [38]. In full virtualization, the guest OS requires no modification, and runs exactly as on physical hardware. Paravirtualization involves modifying the OS kernel to replace certain instructions with calls to the underlying virtualization layer. In turn, hardware-assisted virtualization provides a special hardware operation mode in which OS running inside the virtual machine can issue direct instructions to the physical CPU. Hardware assisted virtualization requires specific hardware support, as provided by Intel VT-x[5] and AMD-V [6]. However, all virtualization solutions rely on so called hypervisors [39], which are software, firmware, or hardware that creates and runs virtual machines, such as KVM[7], XEN[8], vSphere[9], or Hyper-V[10]

[4]http://www.nist.gov/

[5]http://ark.intel.com/Products/VirtualizationTechnology

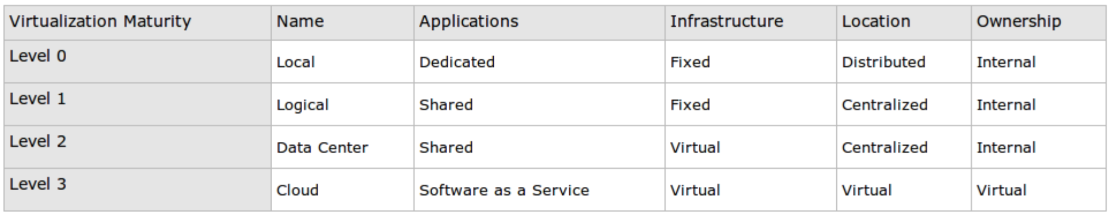
[6]http://www.amd.com/en-us/solutions/servers/virtualization

[7]http://www.linux-kvm.org/

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

[9]https://www.vmware.com/products/vsphere

[10]http://www.microsoft.com/en-us/server-cloud/solutions/virtualization.aspx



Virtualization Maturity	Name	Applications	Infrastructure	Location	Ownership
Level 0	Local	Dedicated	Fixed	Distributed	Internal
Level 1	Logical	Shared	Fixed	Centralized	Internal
Level 2	Data Center	Shared	Virtual	Centralized	Internal
Level 3	Cloud	Software as a Service	Virtual	Virtual	Virtual

Figure 2.1: Virtualization maturity levels (adapted from [44])

Usually hardware resources virtualization enables the emulation of several operating systems running in isolation on the same set pf physical resources. However, virtualization is also encountered in the software domain, providing an abstraction layer on top of the underlying operating system. An example of such a virtualization approach is the Java Virtual Machine [40]. The Java Virtual Machine acts an intermediary between the Java application code and the operating system (OS), enabling the same Java application to run on different operating systems.

As virtualization gained more and more interest, especially in the context of cloud computing, there was an increased interest in both reducing the performance impact of virtualizing resources [41, 42], and providing more flexible virtualization mechanisms. This has led to the emergence of virtualization containers, such as Docker[11]. Docker aims to be a lightweight alternative to VMs, isolating applications/systems/services in containers based on LXC technology [20]. Docker is based on LinuX Containers (LXC)[12], an OS-level virtualization method for running multiple isolated Linux systems (containers) on a single control host. LXC does not provide a virtual machine, but rather provides a virtual environment that has its own CPU, memory, block I/O, network, etc. Without containers, systems must be built for particular environments and operating systems, making their migration difficult, such as from bare metal to VM, or between different clouds. Containers address this by providing a common interface for migrating applications between environments, increasing the possibilities of building, testing, and operating distributed systems in the cloud [43].

2.2 Cloud Computing

The advent of virtualization has opened up numerous possibilities, enabling the emergence of cloud computing, by providing the ability to completely separate the systems that run in a cloud from the supporting hardware infrastructure. The contribution of virtualization to cloud computing is highlighted by Microsoft in their Green Maturity Model for Virtualization [44]. The model defines 4 (0-3) virtualization maturity levels (see Figure 2.1). The virtualization maturity levels 2 and 3 clearly separate the Data Center and Cloud concepts. The Data Center concept is defined by level 2 as having a

[11]https://www.docker.com/
[12]https://wiki.archlinux.org/index.php/Linux_Containers

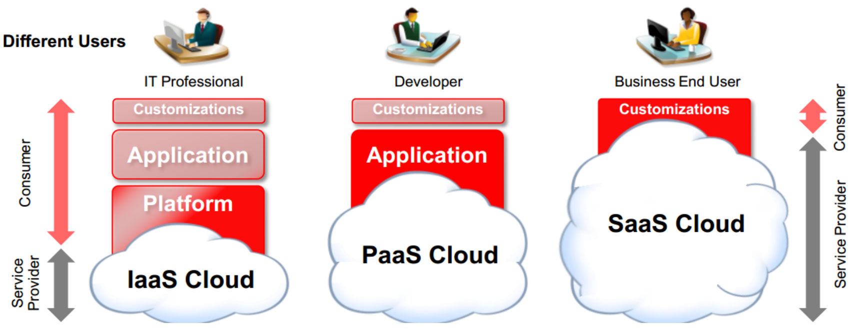


Figure 2.2: Cloud types and users (adapted from [46])

centralized location and an *internal* ownership, while, crucially, for the Cloud, location and ownership are virtualized.

This brings us to the definition of cloud computing, synthesized by US National Institute of Standards and Technology (NIST) [45]. NIST defines cloud computing as "a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources". Today we find a wide range of cloud vendors that provide almost anything under the form of a service in a pay-per-use manner, from hardware Infrastructure as a Service (IaaS) to Platform (PaaS), and Software as a Service (SaaS). These different cloud service models are aimed at different types of users [46], as depicted in Figure 2.2. IaaS offers users the ability of renting computing resources and using them as their own physical computers. IaaS provides maximum flexibility in service customization and usage, and thus targets users which want to run their own software on virtual infrastructure. PaaS services are aimed at users which want increased productivity in their domain, and are not interested in managing a virtual infrastructure. Such users can be developers which want fast access to complete development environments, without worrying about installing the required software stack for their objectives. Finally, SaaS users want to access out of the box services offering certain functionality over a network, without the need to developing them, or configuring software stacks.

2.3 Scalability in Cloud Computing

Scalability is not a new concept, and relates to a system's ability to maintain its operation parameters during changes in load and other stress factors. However, there is no unified consistent definition of scalability [47]. Thus, scalability can and was applied to define different domains and engineering areas. For example, an operating system (OS) can be described as scalable because it has "the ability to retain performance levels when adding additional processors" [48].

However, in the context of elastic distributed systems, scalability usually refers to either *vertical* or *horizontal* scalability [49]. Vertical scalability refers to adding more resources to the system or increasing the computing power of system components. For example, in Figure 2.3 a virtual machine hosting running a web service is scaled-up

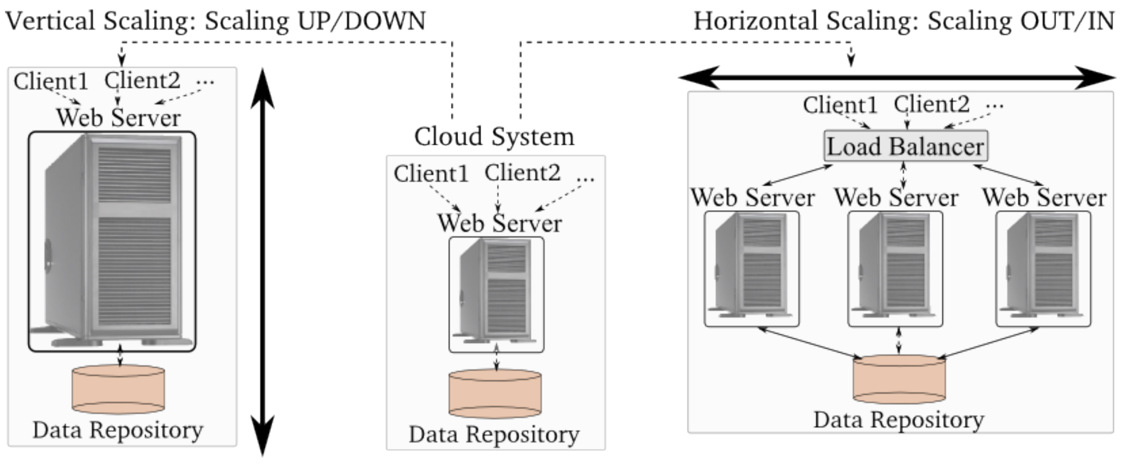


Figure 2.3: Vertical Vs. Horizontal scaling

vertically by adding more resources to it (E.g., memory, storage). The reverse process for vertical scaling is scale-down, i.e., resizing the component by removing computing resources. In turn, horizontal scalability defines the ability of a system to grow horizontally, by replicating components, and thus, computing power, according to need. For example, an elastic system providing functionality as web services, could scale-out horizontally by adding/removing web servers behind a load balancer. The reverse process to scale-out is scale-in, when instances of the replicated component are removed from the system.

While monolithic systems could also be deployed in cloud (depending on their size and the cloud provider capabilities), vertical scaling is limited by the actual physical resources of the used cloud provider. A virtual machine cannot occupy more than a physical one, the maximum VM size depending on the available physical hardware. Thus, most benefit from the cloud can be obtained by distributed systems having components which can be scaled independently, as highlighted by Fehling et al. [50]. However, being distributed is a necessary but not sufficient condition for benefiting from the cloud. To be truly independently scalable, system components must be as decoupled as possible from the other components. Communication mechanisms and software which promote decoupling and transparency are best suited for cloud environments [2]. According to the previous scalability examples, to scale horizontally a web server, a load balancing component could be used to act as intermediary between the server and its clients, enabling dynamic addition and removal of server instances. Similarly, a distributed data end designed to support addition and removal of data processing components is better suited for scaling in the cloud, compared to a single node database.

Cloud providers offer multiple types of services, as highlighted in Section 2.2. Thus, horizontal scaling does not only target allocation and deallocation of VMs. Instead, one can take advantage of multiple types of offered cloud services when scaling, such as allocating a new VM with a new instance of cloud storage, a new instance of a monitoring service to collect data about the VM, and allocating a backup service or configuring it for the new used storage. Thus, horizontal scaling could imply several operations for

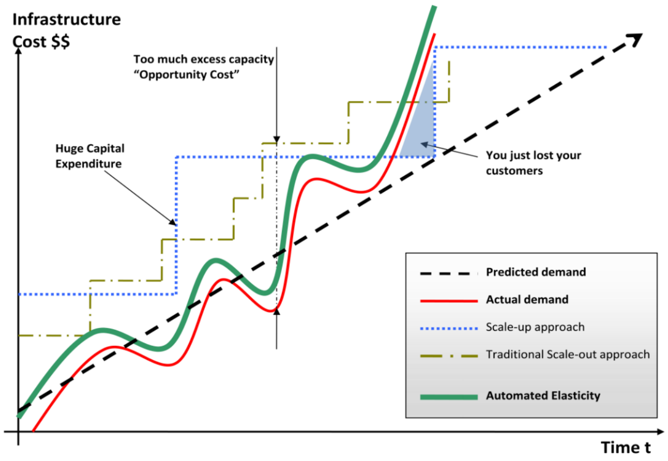


Figure 2.4: Automated elasticity and scalability (adapted from [2])

reconfiguration/allocation/deallocation of multiple cloud offered services, of potentially different types.

2.4 Elasticity in Cloud Computing

The advent of cloud computing has brought with it new concepts and terminologies, *e*lasticity being one of the most encountered attribute in describing cloud services. Before cloud computing, elasticity was used in physics, defining the property of a material to return to an initial form or state following deformation [51]. Elasticity is also a term widespread in economics, describing the change in the values of supply or demand of a product with respect to its change in price [52].

In computing, Amazon Elastic Compute Cloud (Amazon EC2[13]) popularized the term elasticity. In their best practices for architecting systems for the cloud [2], Amazon considers elasticity one of the fundamental properties of the cloud. Elasticity is considered to enable the cloud infrastructure to expand and contract, aligning itself to actual demand, thus increasing resource utilization and reducing cost (Figure 2.4). IBM[14] highlights that one significant difference between scalability and elasticity is the time interval in which changes happen and whether they can be predicted [53]. Thus, scalability is a static property used to describe the behavior of systems under predictable load. Elasticity implies rapid changes in system load, and thus requires more flexibility and even automation in dealing with such situations. Elasticity is also covered by NIST in their definition of cloud computing [45]. NIST defines rapid elasticity as an essential characteristic of cloud computing, enabling "capabilities to be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward to commensurate with demand".

[13]http://aws.amazon.com/ec2/
[14]http://www.ibm.com/

The research community also took interest in elasticity. Unlike traditional scalability which focuses on system performance, the fact that public cloud providers offer services under a pay-per-use model turns cost the driving factor of scale in/down decisions in cloud. This view is expressed by Dustdar et al. [15], determining three core dimensions influencing elasticity: system cost, resource usage, and quality/performance. The importance of the economic aspect of cloud elasticity is highlighted by Agmon et al. [54] on analyzing Amazon EC2 pricing, and Suleiman et al. [6] analyzing the economics and elasticity challenges in public cloud infrastructures. The economics behind elasticity and its impact on the applications and systems running in cloud environments has also been the focus of industry research. The difference between automatic elasticity and traditional horizontal/vertical scalability has been analyzed by Amazon in [55] (Figure 2.4). The business perspective is important as many systems running in cloud belong to businesses which want to obtain a profit.

Thus, a common way of viewing elasticity in cloud computing is including all the aspects impacting the behavior of the systems running in the cloud, not only system performance, but also its cost. Moreover, elasticity implies systems are dynamic and have a certain degree of automation. Such systems can cope with unexpected changes in their load, changes in the cost of used cloud services, or other unexpected factors.

2.5 Service Oriented Computing in the Cloud

Distributed systems benefit from cloud computing, being capable of allocating/deallocating cloud services on-demand. This makes them much more flexible than monolithic applications. In cloud computing, resources and other capabilities are offered as services over a network. Thus, service-oriented computing has emerged as a crucial paradigm for designing, operating and controlling cloud-based, potentially distributed, elastic systems. Originally defined outside of the cloud computing context, service-oriented computing promotes the idea of assembling system components into a network of loosely coupled services, creating flexible and agile systems. Such systems can span organizations and computing platforms, as underlined by Michael et al. [56]. To follow the service-oriented computing principle, systems can be designed having a service-oriented architecture (SOA). SOA is an architectural pattern in computer software design in which system components provide services to other components via a communications protocol, typically over a network [3]. Generically, in a service-oriented architecture, the individual services composing the overall system act as providers and consumers of functionality, collaborating to achieve the overall system's goals. Figure 2.5 depicts a basic collaboration flow, in which services expose their description to a registry, which then is used in finding services offering desired functionality.

However, giving the current development in cloud computing, and especially considering elasticity, the community noticed that to make the best use of the Cloud, service-oriented systems should be built from light cloud services. This gave way to the term of *microservice* architecture, describing a way of structuring cloud systems as a collection of independently deployable, scalable, and controllable services. While currently

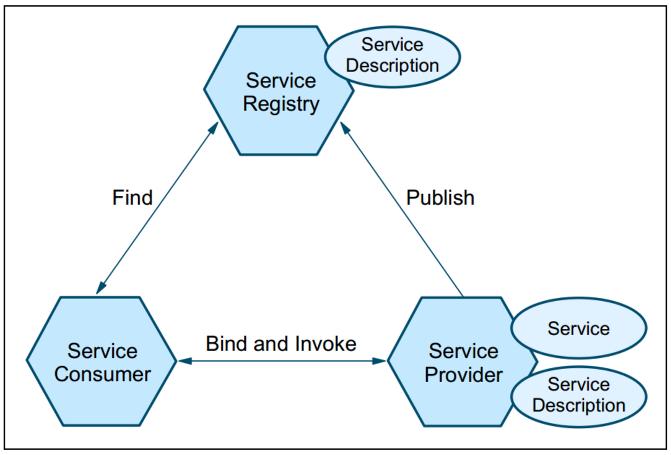


Figure 2.5: Collaborations between services in a service-oriented architecture (adapted from [3])

there is no unanimously accepted definition for microservices, one can see them as an approach to building distributed systems, promoting the use of finely grained services with their own lifecycle, which communicate together (Newman et al. [57]). Thones et al. [58] define a microservice as a small application which has a single responsibility, and which can be deployed, scaled, and tested independently. Thus, microservices can be used in building single or multi-cloud elastic systems, in which individual services can be controlled and managed independent on the other system services.

2.6 Autonomic Computing

IBM identified a "software complexity crisis" with systems becoming larger and larger to solve increasingly complex problems. This crisis could only be solved through *autonomic computing*. IBM's autonomic vision was synthesized by Kephart et al. [59], computing systems being able to manage themselves given high-level objectives from administrators. IBM's definition of autonomic systems relied on the self-* concept, in which systems are capable of self-configuration, self-optimization, self-healing, and self-protection. Compared to traditional systems in which humans need to configure each component, self-configuring systems are able to automatically configure its components, and seamlessly adjust to changes in the system's structure. Self-optimizing systems are able to detect performance problems and improve their performance and efficiency automatically during run-time. Self-healing systems are able to detect and adapt to failures in both the software and underlying hardware, while self-protecting systems are able to detect and automatically defend against malicious attacks.

For achieving autonomicity, IBM defined the MAPE-K control loop (Figure 2.6) with four control phases, Monitoring, Analysis, Planning, and Execution, using and generating Knowledge at each phase:

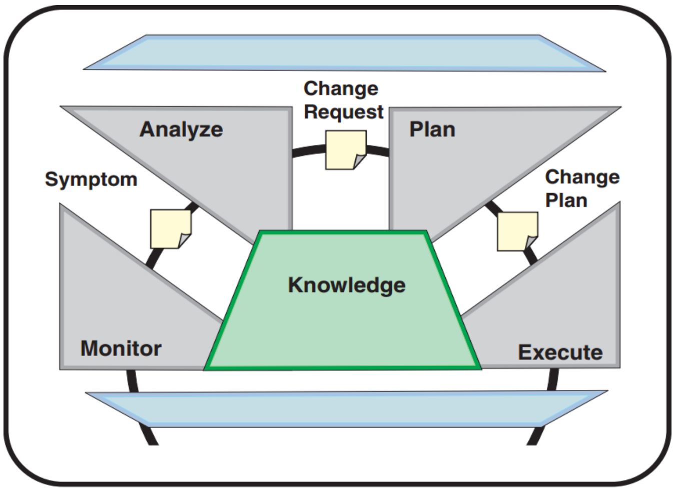


Figure 2.6: Control loop in autonomic computing (adapted from [4])

Monitoring : the behavior of the autonomic system must be monitored through sensors, sensing and reporting the current system state

Analysis : the system state retrieved from the monitoring phase is analyzed, determining with respect to received requirements, if and what is wrong with the system

Planning : based on the analysis result, a plan is generated, containing actions which, if executed, will bring the system back within desired parameters

Execution : the plan generated in the previous phase is executed, bringing the system to a state in which its requirements are fulfilled

Knowledge : all phases of the control loop require knowledge and generate knowledge used by the other phases

Due to the ability of controlling complex systems, autonomic computing can be applied to elastic cloud computing. This aids in the design and operation of smart elasticity controllers, capable of scaling and reconfiguring their systems dynamically during run-time, based on a set of specified requirements. To this end, one goal of this thesis is to ensure that in the *Monitoring* and *Analysis* phases we collect, structure, and provide the required information for enabling automatic elasticity in the cloud. However, due to the potential complexity of cloud systems, designing autonomic controllers requires knowledge about the expected system behavior and boundaries, knowledge not always available to human stakeholders. Thus, the *Monitoring* and *Analysis* phases must also provide human-understandable information, to be used by the human in supervising and configuring their autonomic controllers, and even in manually intervening in the control of elastic systems.

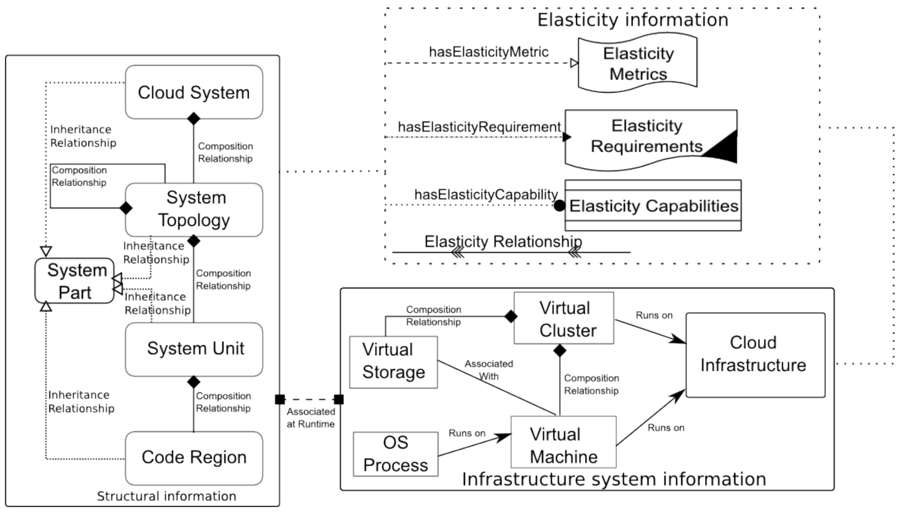


Figure 2.7: Elastic cloud system information model (adapted from [5])

2.7 Elastic Cloud Systems and Design by Units principle

According to the principle of "Design by Units" in elastic systems introduced by Tai et al. [60], any system component and used cloud service is viewed as a *unit*. According to this principle, elastic systems are composed from a collection of such units, units abstracting both human and compute resources. Elastic systems can consist of a multitude of components deployed on one or more cloud infrastructures, using different cloud services to achieve their target functionality. Moreover, elastic systems must expose mechanisms for reconfiguring them at run-time. Such mechanisms allow elasticity controllers to enforce actions to maintain the system within required parameters, such as cost or performance.

In the rest of the thesis we follow the elastic system representation model described in [5], adapting the "Design by Units" principle. The model defined in [5] captures the design-time structure of elastic systems, their run-time structure when running on top of cloud services, and their elasticity requirements and capabilities (Figure 2.7). From a design-time point of view, an elastic cloud system consist of *Units*, functional blocks of a system, capable of offering standalone functionality. One can use the unit concept to define any software artifacts, software components, or used cloud service which offer functionality independent of other units. For example, a unit can be a load balancer, a web server, or database engine. Units which work together to achieve a higher level of functionality can be grouped in higher level entities called *Topologies*. For example, a distributed data storage system could have one or more data controller units managing data access, and multiple data node units storing the data. In this case, the whole distributed storage could be grouped in a topology representing the system's data end.

Once the elastic system is deployed in a cloud, it is using cloud services, from virtual infrastructure services (IaaS), to more complex ones (E.g., PaaS, SaaS). For controlling elastic systems, one must understand how their run-time structure relates to the system's design-time description. To this end, this model captures the cloud infrastructure information, starting with the backbone of cloud computing, the *Virtual Machine* (VM) and *Virtual Storage*. Virtual Machines can in turn be located in certain *Virtual Clusters*, representing zones in a cloud provider where special rules apply. For example, a *Virtual Cluster* can be a cluster with high network performance, or a separate physical location of the cloud provider.

Providing support for system control, the model captures *Elasticity Requirements*, i.e., restrictions imposed over cost, quality, or resource usage of system units, topologies, or over the whole system. To enable analysis of system behavior with respect to these requirements, i.e., determine if requirements are fulfilled or not, requirements are defined over certain system metrics. Any metric targeted by an elasticity requirement is called an *Elasticity Metric*. Finally, to allow run-time control over the elastic system, certain system units can expose *Elasticity Capabilities*. An elasticity capability is any mechanism enabling run-time system reconfiguration, from simple scale-out/in mechanisms, to more complex configuration processes. The capabilities are used by control mechanisms in charge with ensuring that the system's requirements are fulfilled.

The above model is used throughout the thesis as a base for representing elastic systems running in cloud environments. We will refer with the "unit" term to any system/application component designed as a self-contained unit of functionality.

CHAPTER 3

Motivating Scenario

Considering the previous introduced concepts of elasticity, service, and service unit, let us consider a company offering services for maintenance of smart environments (E.g., buildings or vehicle fleets).

The company designs a cloud-based system with functionality for storing, retrieving and managing sensor data, exposing its functionality as a service (Figure 3.1). Sensor data is received by a `Sensor Data Queue` unit, intermediary analyzed by a `Local Data Processing` unit, then sent to an `Event Processing` unit, which analyzes it in detail and then stores it in a data store. If critical events have taken place, an external system is notified. Depending on the system requirements, the units belonging to the `Local Data Processing` topology can be deployed and executed on physical sensor gateways, closer to the sensors sending data, or can be deployed in a cloud on "virtual" gateways running on virtual machines (VMs).

Managing smart environments requires not only management of sensor data, but also data analysis. The company wants to market the data management system as a Data-as-a-Service (DaaS) pay-per-use service for IoT. The DaaS is to be used by clients who want to manage smart environments, but do not have the knowledge or goal to build

[14]Content from this chapter was partially presented in [32, 33, 1, 34, 35].

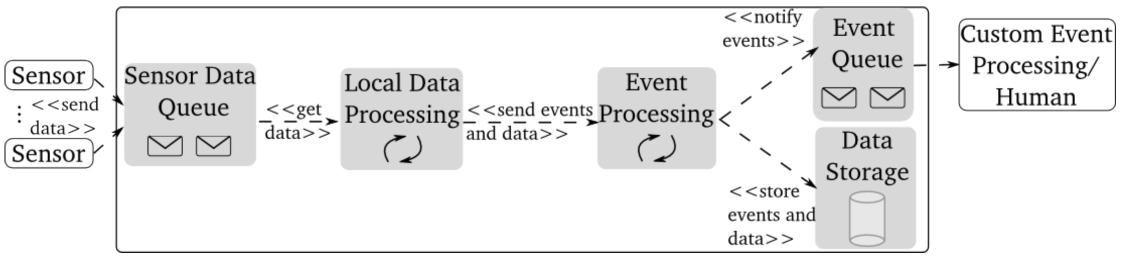


Figure 3.1: Design time view of Data-as-a-Service cloud system for IoT

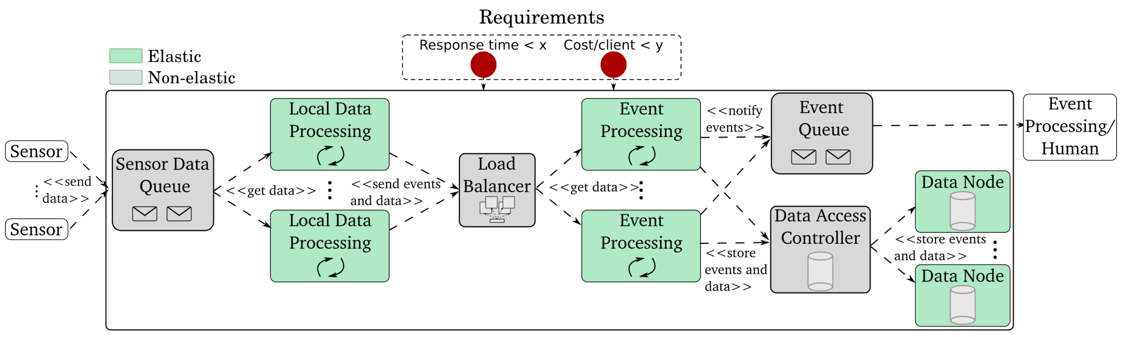


Figure 3.2: Data-as-a-Service elastic cloud system for IoT with elasticity capabilities

and maintain data management systems. Instead, such users want to focus on other areas crucial for their business, such as data analysis. As the company wants to charge its users, it expects the system to fulfill a set of requirements, such as performance levels (E.g., response time), and keep cost within certain limits, for price competitiveness.

Due to the system domain, i.e., smart environments, the system load is highly unpredictable, sensors sending data in a non-linear fashion, depending on environment conditions. As many times sensors rely on battery power, they might be enabled/disabled depending on requirements (E.g., expected data frequency), or changes in the managed environment (E.g., activate water pressure sensors during fire). The system must cope with varying demand, and keep operating costs down for price competitiveness. Thus, the system is designed as an elastic cloud systems (Figure 3.2), being capable of allocating/deallocating system units and associated cloud services at run-time.

Elasticity capabilities are designed and implemented in the system, providing the ability to control it during run-time. This enables run-time elasticity, enabling the system to allocate more or better cloud services to guarantee expected performance, or reduce the number of used cloud services to reduce cost. As the sensors send data to a queue, the number of `Local Data Processing` unit instances retrieving data from the queue can be changed at run-time. Further, a `Load Balancer` (E.g., HAProxy[15]) is added between the `Local Data Processing` and the `Event Processing`. This enables run-time addition and removal of `Event Processing` units. To ensure elasticity at the data end, a distributed data storage engine (E.g., Cassandra[16]) is used, consisting of a `Data Controller` distributing data requests between multiple instances of a horizontally scalable `Data Node` unit.

For realizing elastic cloud systems multiple stakeholders are involved. Figure 3.3 depicts the use cases and stakeholders required for designing and managing elastic cloud systems. The first stakeholders are the *System Developer* and *System Owner*, which have the responsibility of designing *Elastic Systems*. They *Define System Requirements* over performance and cost. An important responsibility is the *Selection of Cloud Services* suitable for the designed elastic system, considering its requirements and elasticity

[15]`http://www.haproxy.org/`
[16]`http://cassandra.apache.org/`

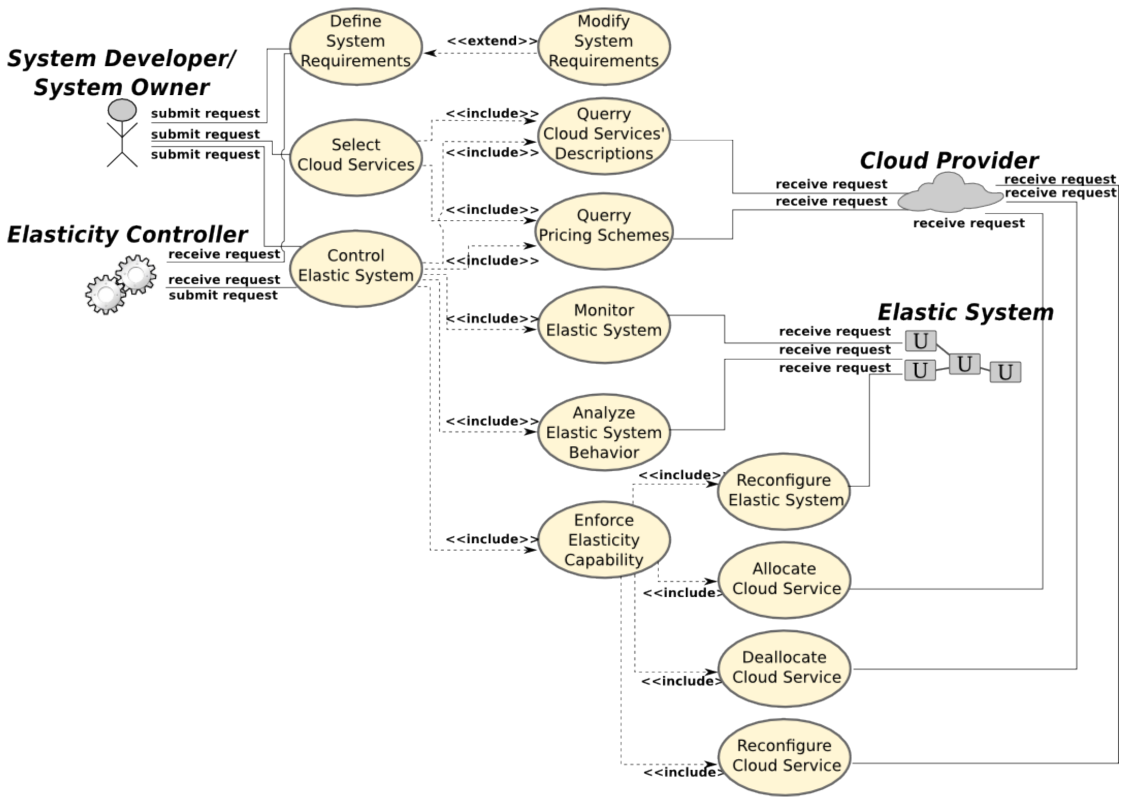


Figure 3.3: Stakeholders and use cases in controlling elastic cloud systems

capabilities. For selecting cloud services, the developer and/or system owner collaborate with the *Cloud Provider*. Each *Cloud Provider* exposes APIs for *Querying its Cloud Services' Descriptions* and their *Pricing Schemes*. The cloud providers are black boxes, system developers having no access to the inner workings of the used cloud providers.

System elasticity is achieved by an *Elasticity Controller*, ensuring requirements are fulfilled by dynamically reconfiguring the system during run-time. Elasticity controllers can be fully automatic, or under the supervision of system developers. System owners/developers submit their system requirements to the used elasticity controller, and trigger the start of the process for *Controlling the Elastic System*. During elasticity control, controllers *Monitor Elastic Systems*, and *Analyze* their behavior.

When required, controllers *Enforce Elasticity Capabilities* implemented on the controlled system, to bring the system in a state in which its requirements are fulfilled. To enforce the capabilities, controllers can query the description of the services offered by cloud providers, and *Allocate/Deallocate Cloud Services* to meet system demand. After any allocation/deallocation, controllers *Reconfigure the Elastic System* to acknowledge the addition/removal of cloud services.

The introduced elastic system will act as base for evaluating the contributions in this thesis. In the rest of the thesis we will reference the stakeholders and use cases described here.

Each of the presented use cases introduces particular challenges. Addressing these challenges, in the rest of this thesis we facilitate the design and run-time management of elastic systems by:

- quantifying the elasticity offered by different cloud services and recommend appropriate services depending on the system's designed elasticity capabilities
- analyzing and describing the elasticity behavior of elastic cloud systems from the whole system level to the underlying virtual infrastructure
- determining behavioral relationships between components of elastic cloud systems influencing system's behavior with respect to its requirements
- evaluating costs and cost efficiency of elastic systems running in public clouds, providing support for cost-aware elasticity control

CHAPTER 4

Building Cloud-native Elastic Systems

4.1 Introduction

The appropriate cloud services must be determined and used for deploying elastic systems. The used cloud services must both provide support for the required system elasticity, and ensure the needed performance and cost. For example, we should avoided selecting a service which must be reserved for 1 year, when service instances are to be created/destroyed hourly. Although several frameworks allow a developer to model such systems, they are often limited to the exact specification of the used cloud services [22, 23], without considering their elasticity. Currently, a system developer has to manually search through cloud providers, and select services for the system s/he needs to construct, without support in evaluating if their elasticity capabilities support its system's required elasticity. To accelerate the development of such elastic cloud systems, we quantify the elasticity capabilities of cloud services. We further provide suitable functions for recommending systems deployment configurations using cloud services providing the necessary elasticity capabilities, and which fulfill resources, quality, and cost requirements.

As mentioned in Chapter 2.2, rapid development in cloud computing has introduced diverse types of cloud services offered by a large number of cloud software providers. To understand the different types of cloud services, we focus on a subset of the main cloud providers Amazon EC2[17], Rackspace[18], HPCloud[19], and Windows Azure[20]. We compile a taxonomy of cloud offerings (Table 4.1), which, while not exhaustive, provides a good idea into what types of services are currently offered by cloud providers, and their

[16]The contributions from this chapter where originally presented in [32].

[17]`http://aws.amazon.com/ec2/`

[18]`http://www.rackspace.com/`

[19]`http://www.hpcloud.com/`

[20]`https://azure.microsoft.com/en-us/`

Category	**Subcategory**	**Cloud providers**			
		Amazon	**Rackspace**	**HPCloud**	**Windows Azure**
Infrastructure	Virtual Machine	m1.small, m1.medium, m1.large, m3.xlarge, ...	FG 256MB,FG 512MB,FG 1GB,...	Extra Small, Small, Medium, ...	Extra Small, Small, Medium, Large, Extra large, ...
	Persistent Storage	Elastic Block Store	Block Storage	Block Storage	Data Disk
	Shared Storage	S3, Glacier, Cloud Front	Cloud Files	Object Storage, CDN	Blob Storage
	Network	Public IP, Cluster Networking, Virtual Private Cloud	Public IP, DNS	Public IP, HP Cloud DNS	Public IP, Virtual Network
Software	Message Queue	Simple Queue Service	-	HP Cloud Messaging	Azure Service Bus, Azure Queue
	Workflow Engine	Simple Workflow Service	-	-	Azure Workflows, SharePoint Workflows
	Communication Service	Simple Email Service, Simple Notification Service	-	-	Notification Hubs
	Data Processing	Oracle E-Business Suite, SAP BusinessObjects, Microsoft SharePoint	Microsoft SharePoint	-	HDInsight, Azure SQL Reporting, Microsoft SharePoint
	Load Balancer	Elastic Load Balancing, Auto Scaling	Cloud Load Balancers	Cloud Load Balancer	Traffic Manager
	Database Services	SQL Server, ElastiCache, RedShift, DynamoDB	Cloud Databases	Cloud Relational Database	SQL Server: Web, Business, and Premium
Management	Monitoring	CloudWatch	Cloud Monitoring	Cloud Monitoring	Health and availability monitoring
	Backup	-	Cloud Backup	-	Azure Backup

Table 4.1: Representative subset of cloud services offered by main cloud providers

heterogeneity. The classification focuses on three main categories of services which can be used to build elastic systems: (i) Infrastructure, (ii) Software, and (iii) Management. The *Infrastructure* category contains all services offering some type of resource on demand (E.g., computing, network, storage), and are the backbone of cloud computing. The second category, *Software*, contains software elements that can be offered as a service, either standalone or accompanying a Virtual Resource service (E.g., message queue, web server, operating system). The third category, *Management* has emerged from the need of exposing management functionality in turn as a service. The management capabilities available in private clouds and datacenters allow cloud users to monitor, analyze, and control the cloud services they use.

This ever increasing plethora of available cloud services has led to increasing effort in understanding how to leverage the multitude of offered services towards building elastic systems directly in the cloud [61, 62, 63]. As highlighted in Chapter 3, elastic systems are designed with certain elasticity capabilities in mind, enabling run-time reconfiguration. Thus, the first crucial step in obtaining such systems is analyzing how the services offered by particular cloud providers influence the designed/envisioned elasticity of the systems using them. This leads to the challenging question on how to quantify the elasticity of these services to ensure that they meet the system's elasticity requirements. Elasticity appears at run-time, through dynamic system reconfiguration depending on certain requirements. For elastic systems it is crucial to select the cloud services providing the necessary elasticity capabilities. This selection takes place not only when deploying an elastic system, but also during run-time system reconfiguration through allocation/deallocation of new service instances. The service selection must consider the system's elasticity. For example, one should avoid selecting a service which must be reserved for 1 year, when service instances are to be created/destroyed hourly. Although several platforms allow a developer to model elastic systems, they are often limited to the exact specification of the used cloud services [22, 23], without analyzing their elasticity. Currently, a system developer has to manually search through cloud providers, and select services for the system s/he needs to construct, without support in evaluating if their elasticity capabilities are suitable for the designed system elasticity.

In the rest of this chapter we provide a mechanism for quantifying the elasticity capabilities of cloud services and recommend services for elastic systems based on their elasticity. The mechanism can be incorporated in different phases of the elastic system's development. To this end, we define an *Elasticity Quantification* function for quantifying the elasticity of cloud services. Based on the quantification function and multi-level system requirements over cost, quality, and resources, we provide algorithms and a platform for aiding in the construction of elastic systems. The platform recommends system deployment configurations and cloud services, based on the desired system elasticity.

In this chapter we make the following contributions:

- a model for describing elasticity capabilities of cloud offered services and multi-level elasticity requirements of cloud systems
- the concept of elasticity quantification and algorithms for evaluating the elasticity of cloud services
- platform for recommending suitable system deployment configurations w.r.t. elasticity requirements, to be used by system developers, automatic cloud service composition tools, or elasticity controllers

4.2 Roadmap

The rest of this chapter is structured as follows. Section 4.3 expands the motivation and approach. Section 4.4 discusses elasticity quantification of cloud services. Section 4.5

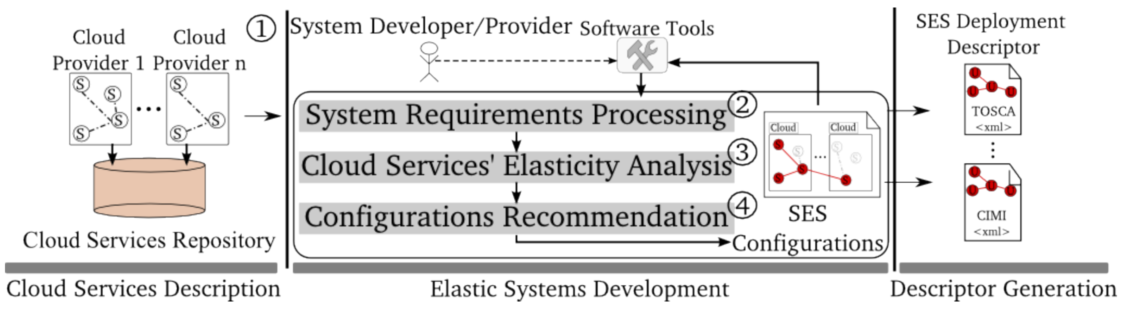

Cloud Provider 1
Cloud Provider n
①
Cloud Services Repository
System Developer/Provider
Software Tools
System Requirements Processing ②
Cloud Services' Elasticity Analysis ③
Configurations Recommendation ④
SES Configurations
SES Deployment Descriptor
TOSCA <xml>
CIMI <xml>
Cloud Services Description
Elastic Systems Development
Descriptor Generation

Figure 4.1: Constructing elastic systems using cloud services

introduces the algorithms for recommending configurations of elastic systems made from cloud services. Section 4.6 presents our prototype platform for solving the aforementioned issue, and validates it through experiments on building the mentioned elastic DaaS system. Section 4.7 concludes the chapter.

4.3 Motivation and approach

To understand the challenges in constructing the Data-as-a-Service (DaaS) elastic system presented in Chapter 3, we must consider that cloud systems can be constructed from custom software running on top of or along a set of used cloud services. Such services can range from the most basic IaaS offerings such as VMs, storage, or network, to SaaS offerings such as message queues, or even business analytics offerings. Thus, our DaaS system can be built from several cloud services, from basic IaaS VM services, to SaaS offerings for data storage, and a message oriented middleware for events notifications. In this scenario we assume the units of the *Local Processing Topology* are not deployed in the cloud, and instead run on IoT gateways located closer to the IoT sensors. Thus, in the rest of the chapter we focus on the cloud-based part of the DaaS, the *Event Processing* and *Data End* topologies. To develop the DaaS, using current approaches such as introduced by Patiniotakis et al. [64] and Kamateri et al. [65], a developer has to manually investigate all services offered by various cloud providers, and evaluate if their elasticity capabilities provide the required elasticity control options. Then, a developer can use existing design and modeling tools such as Winery [22] or c-Eclipse [66] to design and deploy the DaaS on cloud infrastructures. However, manually selecting each service needed for constructing the DaaS is laborious, complex, and error prone.

To address the aforementioned issues and reduce the complexity in constructing elastic cloud systems, we design a cloud system construction process (Figure 4.1) having the following phases:

- The elasticity capabilities of services from different cloud providers are captured and modeled. The capabilities are used in determining if the analyzed cloud services provide support for the designed system elasticity (indicated by ①)

- The system requirements in terms of performance, resources, and cost are captured. The requirements are used in determining if the analyzed cloud services offer the required system functionality ②)
- We define an elasticity quantification function for evaluating cloud services' support for the envisioned system elasticity. The function is used in ordering cloud services after their elasticity, i.e., after their cost, resources, or performance elasticity (indicated by ③)
- Based on the quantification function, we apply algorithms for recommending system configurations composed of cloud services. The configurations are generated in software-interpretable deployment description formats to be used by existing cloud deployment tools (indicated by ④)

4.4 Quantifying elasticity of cloud services

4.4.1 Modeling elasticity capabilities of cloud services

Elasticity capabilities of a cloud service define how its cost, quality, and resources can be configured during its life-cycle. This determines the available control options for particular service properties during its instantiation and run-time. Following the multi-dimensional principle of elasticity defined by Dustdar et al. [15], we define elasticity capabilities of a service as *configuration possibilities with respect to cost, quality, resources, and associations with other services, and the dependencies among them.* Thus, an elasticity capability defines what resource, cost, quality or associations among services can be created. The elasticity capability also specifies when the service can be reconfigured, such as during instantiation or run time, and how often. By studying main cloud providers, such as Amazon EC2[21], Rackspace[22], HPCloud[23], and Windows Azure[24], and from other studies [6], we noticed that similar cloud services can have different available configurations at different phases of their life-cycle. The available configurations for a cloud service can depend either on its combination with other services, or on the cloud provider offering it. For example, certain Amazon EC2 VM types can only be used in conjunction with Amazon EBS storage service, while others can be instantiated and used standalone. Therefore, the elasticity capabilities of both individual and associations of services are crucial in providing a base for evaluating which services are suitable for a particular system's elasticity. Thus, in the following we aim to understand and model the elasticity capabilities of cloud services and their dependencies, and quantify the elasticity of cloud services to support the development of elastic cloud systems.

To evaluate if a cloud service provides the necessary elasticity capabilities for the envisioned run-time elasticity control, we need to capture when we can use an elasticity

[21]http://aws.amazon.com
[22]http://www.rackspace.com
[23]http://www.hpcloud.com/
[24]http://azure.microsoft.com/en-us/

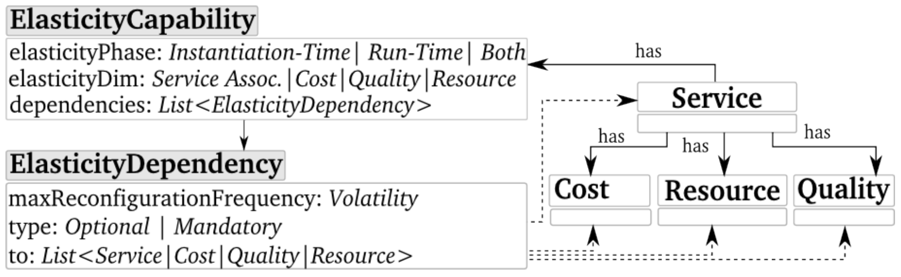

ElasticityCapability
elasticityPhase: Instantiation-Time | Run-Time | Both
elasticityDim: Service Assoc. | Cost | Quality | Resource
dependencies: List<ElasticityDependency>
ElasticityDependency
maxReconfigurationFrequency: Volatility
type: Optional | Mandatory
to: List<Service | Cost | Quality | Resource>
has
Service
has
has
has
Cost
Resource
Quality

Figure 4.2: Representing elasticity capabilities of cloud services

capability (elasticity phase). Further, we must capture how often can the capability be invoked (volatility), and if the service can be used standalone or not (dependency type). Most existing cloud services representation models capture resources and quality properties, such as defined by et Goncalves al. [67] and Villegas et al. [29]. We start from such common representation models, and focus on enriching them with elasticity capabilities (Figure 4.2).

We define an *Elasticity Capability* having an *elasticityPhase*, specifying if it can be invoked during the service's *Instantiation-Time*, *Run-Time*, or *Both*. The elasticity dimension associated to the capability is defined by the *elasticityDim* property, and is one of *Cost*, *Quality*, *Resource*, or *Service Associations*. The dimension can be used in analyzing which "type" of elasticity a service supports. Cost elasticity implies the service has multiple cost options under which it can be used. E.g., in Amazon EC2 there are multiple VM cost options, Spot, On Demand, Reserved, but not all available to all VM types. Resources elasticity indicates the service has multiple resource configurations between which a service user can choose when creating a service instance. E.g., choosing between a x64 and x86 CPU architecture for a VM type. Quality elasticity, similar to resource elasticity, indicates the service can be used under multiple quality configurations. E.g., instantiating an Amazon EBS storage service with different level of IO performance. Finally, service association elasticity indicates the service can be used in conjunction with other cloud services. E.g, only certain VM types in Amazon EC2 provide support the Amazon ECB storage service.

As one capability might indicate multiple configuration possibilities, the elasticity capability has a set of *Elasticity Dependency* instances. An *ElasticityDependency* specifies to which *Cost*, *Quality*, *Resource*, or *Service* a cloud service can be associated using the *to* property. *Volatility* is the most important dependency property, defining its minimum "usage" time, determining the frequency at which the dependency can be allocated/deallocated for the service (E.g., hourly, or monthly), and thus influencing the service's elasticity. For example, a service having dependencies which can be allocated/deallocated hourly is more elastic than one with dependencies which can be reconfigured only on a monthly basis. We describe if a dependency is *Mandatory*, or *Optional* using the *type* property. Mandatory dependencies decrease the elasticity of a service by requiring for the dependency to be always allocated with the service, reducing its usage flexibility. This model provides a base for evaluating if services' configuration options are appropriate for

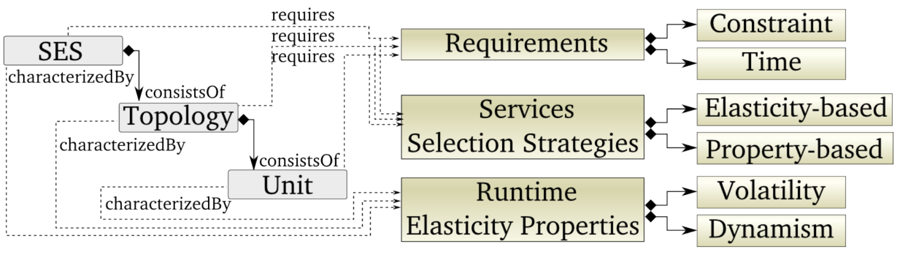


Figure 4.3: Representing requirements for elastic systems

particular system's elasticity control.

4.4.2 Representing requirements for elastic systems

Using the previous elasticity capabilities model, towards supporting the development of elastic systems, we provide a customizable function for quantifying the elasticity of cloud services. The function is to be used in recommending services best suited to the expected system run-time elasticity control. For this we must understand and model requirements, run-time properties, and service selection strategies. Different stakeholders might have different perspectives over a system. Using our platform, requirements can be specified at different system levels, according to the model defined by Copil et al. [5] and presented in Chapter 2.7. An elastic system is composed of units, logically grouped in topologies (Figure 4.3).

Elasticity of a system appears at run-time, through dynamic reconfiguration with respect to system requirements. Thus, describing and analyzing the expected run-time properties of the system is crucial in discovering services that support the expected behavior. Through *Runtime Elasticity Properties*, we capture the expected run-time behavior of a system using *Volatility* and *Dynamism*. Volatility is applied in recommending services with suitable capabilities for the expected unit usage time. Dynamism describes the number of units expected to be allocated/deallocated within a time period in a time interval. For example, we can describe a system unit which uses its instances on average one hour (volatility), and allocates/deallocates 10 instances within 5 minutes every hour (dynamism).

Depending on its purpose, a system might require different elasticity control strategies over its units, topologies, or whole system, such as maximizing performance for a unit, and quality for another. Thus, for selecting services which support the required control, we employ *Services Selection Strategies.* We first define *Elasticity*-based selection strategies, which are used to order and select cloud offered services based on their elasticity capabilities. These strategies are crucial in considering the elasticity capabilities of cloud services when building systems. We define five *Elasticity-based* strategies: *Max {Overall, Cost, Quality, Resource, Service Association} Elasticity*, ordering cloud services based on their particular elasticity dimensions.

While cloud services must support elasticity control, they must also fulfill a set of resource, performance, and cost requirements, as any other system. Thus, we consider such requirements as requirements expressed over the cloud services' properties, and introduce *Property-based* strategies: *Max {Fulfilled Requirements, Quality, Resources}*, and *Min Cost*. Multiple different strategies can be specified for each system unit, topology, or whole system, covering all potential system requirements.

For using property-based service ordering strategies, we need to evaluate requirements over those properties. To this end, *Requirements* specify the cost, quality and resources required by the system, and are represented as functions of form $f_{elReq}(constraint, g(time))$. The `constraint` is a function depending on the cost, quality or resource metric on which the requirement is made, the type of constraint (E.g., greater than) and the requested values ($h_{constraint}(metric, operator, value)$). A `time` parameter enables the specification of time-varying requirements.

4.4.3 Function for quantifying elasticity of cloud services

Different systems can have different elasticity requirements, depending on system requirements, and designed elasticity control mechanisms. For example, one system might require more cost control options, and thus cost elasticity would be more important than quality elasticity. Thus, we provide a set of customizable coefficients for quantifying the elasticity of services, which can be tailored to suit particular system requirements. Quantifying elasticity enables an ordering of services after their elasticity, crucial in recommending services for system configurations.

One important factor in evaluating elasticity of cloud services is the phase during the service's lifetime when elasticity capabilities are active: instantiation-time, run-time, or both. Let v_i, v_r, and v_{ir} be user-defined values representing the importance of `Instantiation-Time`, `Run-Time`, and `Both` phases, respectively, for a particular system; $v_i, v_r, v_{ir} \in [0, 1]$. Thus, we define an `ElPhaseQ` coefficient for quantifying the *phase* in which a service can exhibit elasticity, as follows:

$$ElPhaseQ(phase) = \begin{cases} v_i & \text{if } phase = \texttt{Instantiation-Time} \\ v_r & \text{if } phase = \texttt{Run-Time} \\ v_{ir} & \text{if } phase = \texttt{Both} \end{cases} \tag{4.1}$$

Typically, to obtain system configurations with maximum elasticity, v_r should be at least twice as v_i, and v_{ir} their sum (E.g., $v_i = 0.33$, $v_r = 0.67$, and $v_{ir} = 1$).

Dependencies between services increase (optional dependencies) or decrease (mandatory dependencies) the service's elasticity. Let v_o, v_m be user-defined values representing the "importance" of `Optional` and `Mandatory` dependencies, respectively, for a particular system; $v_o, v_m \in [-1, 1]$. We define an `ElDepQ` for quantifying the elasticity dependencies between services as follows:

$$ElDepQ(dependency) = \begin{cases} v_o & \text{if } dependency.type = \texttt{Optional} \\ v_m & \text{if } dependency.type = \texttt{Mandatory} \end{cases} \tag{4.2}$$

Typically, to obtain system configurations with maximum elasticity, v_o and v_m should have the same value but opposite signs, with $v_m < 0$ as mandatory dependencies decrease elasticity (E.g., $v_o = 1$, and $v_m = -1$).

The `Volatility` of a cloud service heavily influences the service's elasticity, and might have different importance for different systems. Thus, we consider a custom `VolatilityQ` coefficient for quantifying volatility, supplied as to suit particular system requirements. Typically, `VolatilityQ` would have the form $numberOfAllowedReconfigurations/timeInterval$.

Based on the above coefficients, we quantify a single elasticity capability of a cloud service as ECQ:

$$ECQ(C) = ElPhaseQ(C.phase) * \Sigma_{dep \in C.dependencies}\ VolatilityQ(dep) * ElDepQ(dep) \quad (4.3)$$

where C is an elasticity capability, *C.phase* its elasticity phase, *C.dependencies* its elasticity dependencies, and *dep* a single elasticity dependency.

For evaluating the overall elasticity of a cloud service S over all elasticity dimensions (Cost, Quality, Resource, and Services Associations) we define an `Elasticity Quantification (EQ)` function as:

$$EQ(S) = \Sigma_{D \in cost,quality,res,servicesAssoc}\ W_D * \Sigma_{C \in D.capabilities}\ ECQ(C) \quad (4.4)$$

where D is an elasticity dimension, $W_D \in [0, 1]$ is its weight, and C is an elasticity capability of S on dimension D. Different W_D coefficients for each dimension D can be set to suit particular system requirements. For example, a system interested only in cost elasticity would set W_{cost} to 1, and the other W_D coefficients to 0.

4.5 Recommending cloud services for elastic systems

In this section we introduce algorithms for recommending system deployment configurations based on the elasticity capabilities of existing cloud services. As one service could be instantiated under different configurations depending on its elasticity capabilities, Algorithm 1 evaluates an entity (service, quality, cost, or resource) with respect to a system unit requirements. The algorithm's output is a set of potential configurations for the entity's elasticity dependencies (`entityCfgs`), depending on the requirements they fulfill. One cloud service might have different mandatory and optional elasticity dependencies on other entities with different properties (E.g., different cost). Thus, after the algorithm evaluates the static properties of the cloud service in Line 2 (`EvalRequirements` function), it continues by applying the `GetEntityCfgs` function recursively over its *mandatory* dependencies (must be used). Lines 3-6 determines the unit requirements fulfilled by the dependencies' configuration options, and add these options to the `entityCfgs`. Next, the complete set of possible configurations for the entity is generated. To this end, the potential configurations of the entity's optional dependencies are evaluated against requirements (Lines 7-10), and their configurations added to the `entityCfgs`.

Algorithm 1 Evaluating cloud service against system unit requirements

Input: *entity*,*requirements*; **Output:** *entityCfgs*

```
 1: function GetEntityCfgs(entity, requirements)
 2:     fulfilledReqs = EvalRequirements(entity, requirements)
 3:     for d in entity.elasticityCapabilities.mandatoryDependencies do
 4:         capabilityCfgs = GetEntityCfgs(d,requirements)
 5:         entityCfgs.addCapabilityCfgs(d, capabilityCfgs)
 6:     end for
 7:     for d in entity.elasticityCapabilities.optionalDependencies do
 8:         capabilityCfgs = GetEntityCfgs(d,requirements)
 9:         entityCfgs.addCapabilityCfgs(d, capabilityCfgs)
10:     end for
11:  return entityCfgs
12: end function
```

Algorithm 2 Elasticity-driven system configurations generation

Input: *system*, *services*, *cfgsCount*

Output: *cfgs* - set of possible system configurations

```
 1: function RecommendSystemCfgs(system, services, cfgsCount)
 2:     unitsRequirements = MapRequirements(system.requirements)
 3:     for unit in unitsRequirements do
 4:         EQ = system.eqFunction(unit)
 5:         potentialCfgs = []
 6:         for s in services do
 7:             entityCfgs = GetEntityCfgs(s,unit.reqs)
 8:             if entityCfgs != empty then
 9:                 potentialCfgs.add(entityCfgs, EQ(entityCfgs))
10:             end if
11:         end for
12:         cfgs.add(unit, potentialCfgs.getBest(cfgsCount, system.strategies(unit)))
13:     end for
14:  return cfgs
15: end function
```

Algorithm 2 applies elasticity quantification functions to generate a user-specified number of decreasingly elastic system deployment configurations. Input `system` description contains requirements, run-time properties, service selection strategies, and custom `EQ` functions defined at any system level, from the whole system, to topologies and units, which are mapped to system units (Line 2). If conflicts are detected between levels, the lower level is applied. For each unit, its elasticity quantification function `EQ` is retrieved from the supplied `system` description (Line 4). Then, for each cloud service, `GetEntityCfgs` (Algorithm 1) is called, obtaining a set of potential service

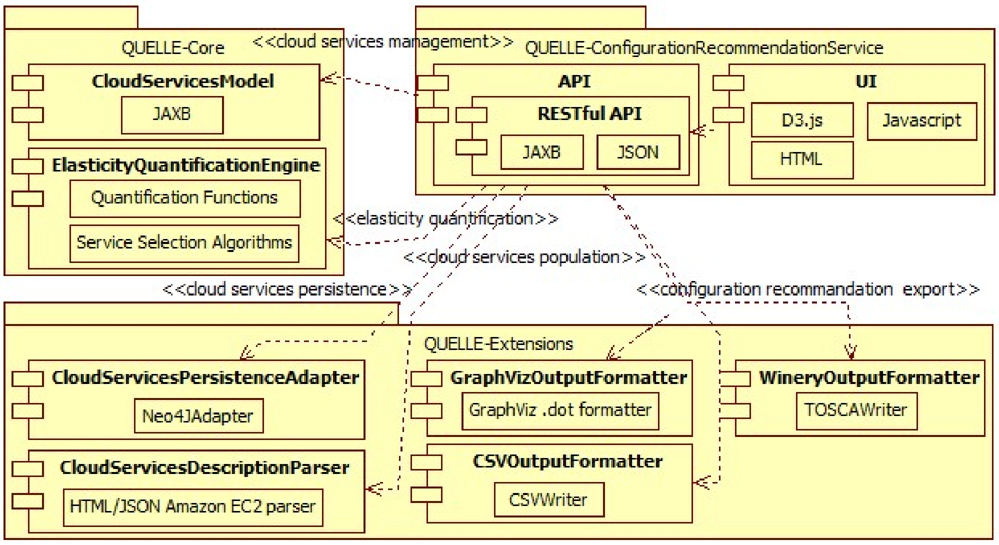

QUELLE-Core
<<cloud services management>>
QUELLE-ConfigurationRecommendationService
CloudServicesModel
JAXB
ElasticityQuantificationEngine
Quantification Functions
Service Selection Algorithms
API
RESTful API
JAXB
JSON
UI
D3.js
Javascript
HTML
<<elasticity quantification>>
<<cloud services population>>
<<cloud services persistence>>
<<configuration recommandation export>>
QUELLE-Extensions
CloudServicesPersistenceAdapter
Neo4JAdapter
CloudServicesDescriptionParser
HTML/JSON Amazon EC2 parser
GraphVizOutputFormatter
GraphViz .dot formatter
CSVOutputFormatter
CSVWriter
WineryOutputFormatter
TOSCAWriter

Figure 4.4: QUELLE platform

configurations `entityCfgs` (Lines 6-11). The `EQ` function for the system unit is used to quantify the elasticity of the potential service configurations from `entityCfgs` (Line 9). Finally, supplied unit strategies `system.strategies(unit)` are applied sequentially in recommending from `potentialCfgs` the best `cfgsCount` decreasingly elastic configurations, according to their elasticity quantification (Line 12).

Quantifying elasticity towards selecting cloud services ensures that during the system execution, an elasticity controller has the appropriate control options to be enforced depending on system requirements and run-time behavior.

4.6 Evaluation

4.6.1 Prototype

The functions, algorithms and models described in Sections 4.4 and 4.5 are implemented and integrated in QUELLE, a platform[25](Figure 4.4) for quantifying the elasticity of cloud services. For managing the *Cloud Services Model*, a graph-based Neo4j[26] *Cloud Services Persistence Adapter* was implemented. The population of the cloud services' repository (see Figure 4.1) should ideally be an automatic process, with the increase in cloud providers' description APIs. However, currently we rely on available custom description services and HTML parsing to populate our model. For enabling QUELLE's integration in existing software engineering processes, system requirements constructed

[25]Prototype and supplement materials: `http://tuwiendsg.github.io/QUELLE/`
[26]`http://www.neo4j.org/`

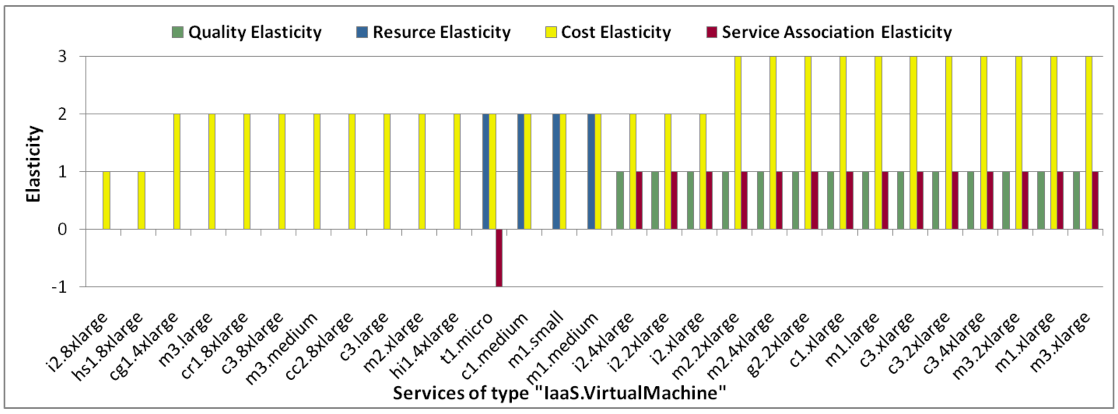


Figure 4.5: Elasticity quantification and evaluation of Amazon EC2 IaaS services

by third party tools are submitted as XML, and configuration recommendations returned as XML for easy processing. Currently, a TOSCA[27]-based output is generated using QUELLE's output formatter, which can be visualized using Winery [22].

4.6.2 Evaluating elasticity of Amazon EC2 cloud services

Most cloud providers still offer only basic cloud services, with reduced configuration and combination options, and implicitly, reduced elasticity. This limits our options of using real cloud services in our experiments. Thus, we focus on a single cloud provider, Amazon EC2, from which we model 29 IaaS VM cloud services, each with various elasticity capabilities, describing a total of 253 possible service configurations, sufficient for showcasing our elasticity quantification functions. Additionally, Amazon provides services for cloud storage (EBS), Monitoring, and Messaging, each with individual elasticity capabilities, sufficient for building the DaaS.

As the desired elasticity might vary depending on the stakeholder, our platform provides a customizable quantification function relying on user-defined `VolatilityQ`, `ElDepQ`, and `ElPhaseQ` coefficients. As the user is interested in building an elastic system, s/he expects services to be allocated/deallocated often. As Amazon bills its services minimum on a hourly basis, the supplied volatility quantification coefficient is `VolatilityQ = 1/minLifetime (Hours)`, generating a volatility of 1 for hourly reserved services, and 1 / (365 * 24) for yearly reserved services. As the user wants to use services which have as few dependencies on other services, s/he supplies an elasticity dependency quantification coefficient `ElDepQ = {1 if Optional, and -1 if Mandatory}`. Finally, the supplied elasticity phase quantification coefficient is `ElPhaseQ = {0.33 if Instantiation-Time, 0.67 if Run-Time, 1 if Both}`, and all elasticity dimensions have same weight coefficient W_d=1.

[27]https://www.oasis-open.org/committees/tosca

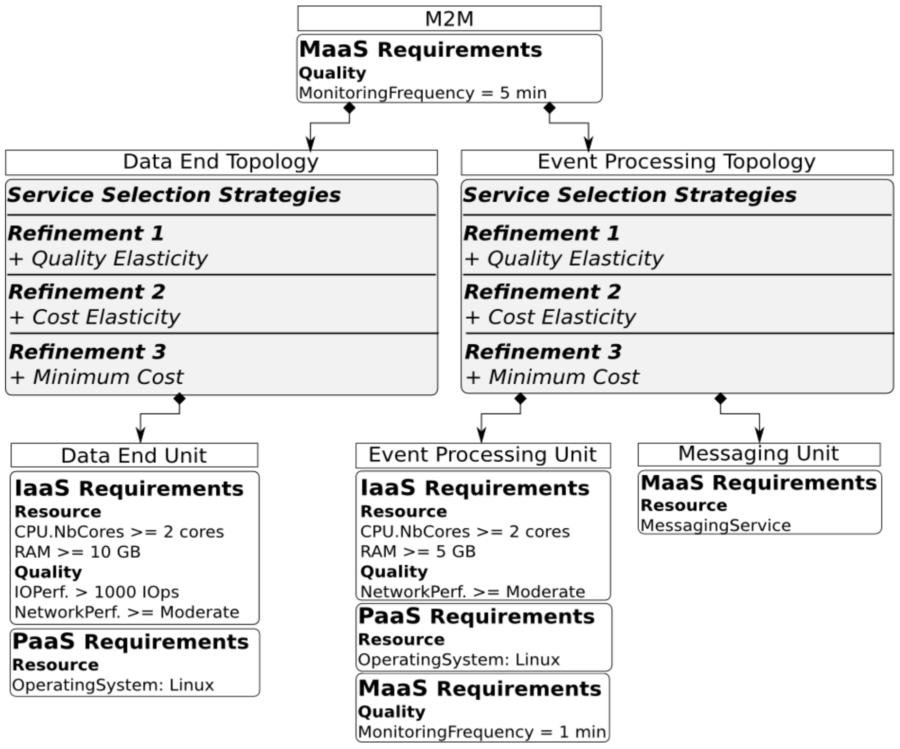


Figure 4.6: Multi-level DaaS elasticity requirements

The result of quantifying the elasticity of Amazon EC2[28] services over cost, quality, resources, and services associations is depicted in Figure 4.5. As the defined `VolatilityQ` function quantifies close to zero all options of reserving a service for 1 or 3 years, the cost elasticity of most services, such as `m3.large` is quantified close to 2. Amazon EC2 services which have optional dependencies have additional cost and quality control options. Thus, Amazon EC2 IaaS services which can be associated with an `EBS` service have their service association elasticity quantified to $\geq$ `1`, and cost elasticity quantified to $\simeq$ `3`.

4.6.3 Recommending system configurations

We aim to accelerate the development of elastic systems by recommending deployment configurations using cloud services providing the required elasticity capabilities. Thus, we have defined a four phase recommendation process (Figure 4.1): (i) processing system requirements, (ii) quantifying elasticity of cloud services, (iii) recommending elasticity-driven system configurations, and (iv) exporting system configurations as cloud deployment descriptor.

As a user might not initially know the complete system requirements, we apply an iterative approach. The recommended configurations are analyzed by a user, the system

[28]Services' description accurate at time of writing

Service Selection Strategies	Recommended IaaS Services	Quality Elasticity			Cost Elasticity		
		Avg.	Min.	Max.	Avg.	Min.	Max.
Max Requirements	23	0.6	0	1	2.39	1.0004	3.0004
+ Quality Elasticity	14	1	1	1	2.78	2.004	3.0004
+ Cost Elasticity	11	1	1	1	3.0004	3.0004	3.0004
+ Minimum Cost	1	1	1	1	3.0004	3.0004	3.0004

Table 4.2: Iterative services selection for Event Processing unit

requirements refined accordingly, and resubmitted. First, mixed system requirements w.r.t. cost, resource and quality, are described by the user in a top-down fashion, from the entire system to individual units. At the *system* level, a requirement for a Management as a Service (MaaS) service with a monitoring frequency of 5 minutes is specified, which will be applied to all system's units. As the units belonging to the *Event Processing Topology* level perform sensitive computation, a MaaS requirement for a service with 1 minute monitoring frequency is specified, overriding the 5 minutes frequency system level requirement. The *Event Processing Unit* requires an IaaS service with over 2 CPU cores and 5 GB of RAM, and a Moderate network performance. In turn, the *Messaging Unit* requires a PaaS service of type messaging. Similarly, the *Data End Unit* requires an IaaS service providing at least 10 GB of RAM, I/O Performance of at least 1000 IOps together with at least a Moderate network performance.

While in the following we focus on IaaS services as they are most abundant and exhibit most elasticity in current cloud computing, we can apply the same approach for PaaS and MaaS requirements, as shown at the end of this section.

First iteration: The user submits to QUELLE system requirements, without elasticity-based selection strategies. Focusing on the *Event Processing Unit*, the user sees that 23 IaaS services were recommended (Table 4.2), with varying quality and cost elasticity. **Second iteration:** The user adds a *Quality Elasticity* strategy, maximizing the quality options available at run-time. In turn, 14 services are recommended, with quality elasticity equal to 1, as the only modeled quality elasticity capability is an *EBS Optimized* storage option. **Third iteration:** The user adds a *Cost Elasticity* strategy, ensuring the system can switch between as many pricing schemes as possible during run-time. Thus, 11 services are recommended, with cost elasticity of $\simeq 3$, due to supplied *VolatilityQ* function evaluating yearly cost schemes $\simeq 0$, and hourly pricing schemes (E.g., *Spot*) to *1*. **Fourth iteration:** The user also wants *Minimum Cost*, reducing the recommended services to 1, fulfilling most resource requirements, having maximum quality and cost elasticity, and minimum cost.

In Table 4.3 we showcase the importance of quantifying elasticity capabilities of cloud services in system construction. To this end we compare the usage of *Elasticity-based* service selection strategies with only using the *Property-based* strategies *Minimum Cost* and *Max Requirements*. With the later strategies, requirements are matched and services with minimum cost selected in a traditional fashion, recommending 3 services with varying quality and cost elasticity. Applying *Elasticity-based* strategies, the system's elasticity is increased, recommending an *m1.xlarge* service with more control options over its quality

Service Selection Strategies	Recommended IaaS Services	Avg. Quality Elasticity	Avg. Cost Elasticity
Max Requirements + Minimum Cost	m3.large, m1.large m2.xlarge	0.33	2.33
Max Requirements + Quality Elasticity + Cost Elasticity + Minimum Cost	m1.xlarge	1	3.0004

Table 4.3: Elasticity versus property-based service selection for Event Processing unit

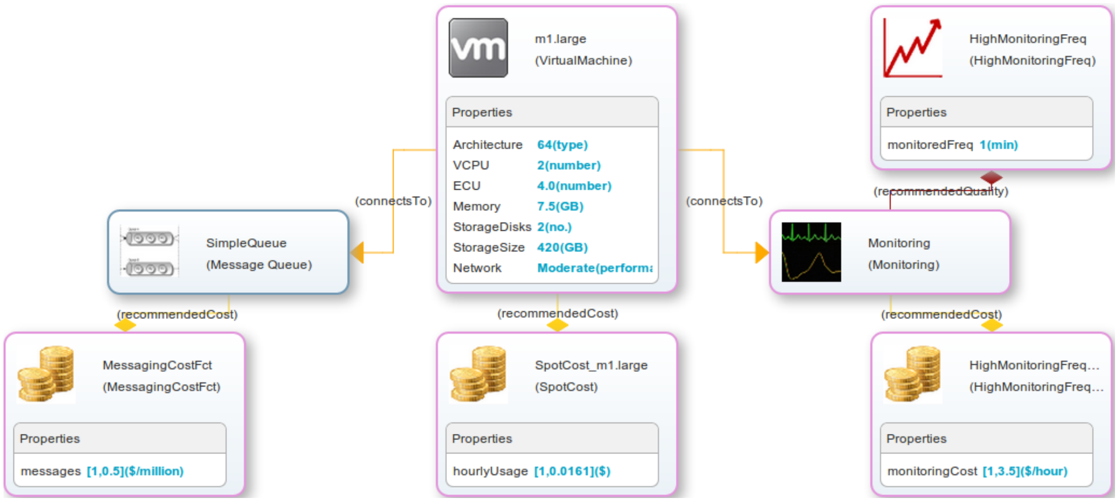


Figure 4.7: Complete configuration recommendation for Event Processing topology

and cost elasticity dimensions.

Processing all IaaS, PaaS, and MaaS requirements refined above, our prototype generates a TOSCA descriptor containing the recommended system configuration. For the *Event Processing Topology*, the recommendation is visualized in (Figure 4.7) using Winery [22], a TOSCA modeling and visualization tool. The recommendation contains an *m1.large* IaaS service fulfilling the resource and network performance quality requirements with associated *SpotCost*, due the *Minimum Cost* strategy. A SaaS *Monitoring Service* with a *High Monitoring Frequency* is recommended for the monitoring frequency requirement, and a MaaS *SimpleQueue* service for the message oriented middleware requirements. In a similar fashion, recommendations are provided for the *Data End Topology*.

Applying the same process for the *Data End Topology*, our prototype generates a solution containing three cloud services (Figure 4.8). The recommendation for the data end contains as for the event processing unit, a *m1.xlarge* IaaS service fulfilling the resource and network performance quality requirements. In addition, due to I/O performance requirements, a IaaS *EBS* is recommended, with associated *High I/O Performance* and *High I/O Performance Cost*. Further, a *Monitoring Service* is recommended, but, as the data end does not require high monitoring frequency as the event processing, a *Standard Monitoring Frequency* option is recommended for the service, with associated cost option.

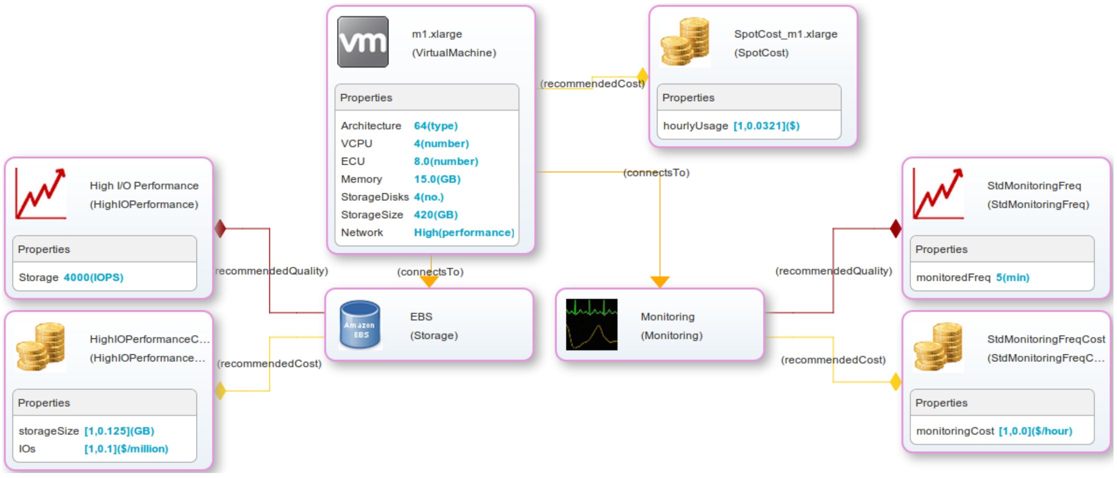


Figure 4.8: Complete configuration recommendation for Data End topology

In these experiments we highlighted that, using our platform, a system developer does not have to search through all cloud providers for services providing necessary elasticity, and thus, accelerates the system's time to deployment.

4.7 Conclusion

In this chapter we have focused on reducing the complexity of building elastic systems running in the cloud. We highlighted that selecting the suitable cloud services, their particular configuration options, and associated cost, is not trivial, especially when considering complex systems. Complex systems can have units using multiple services, with potentially divergent requirements over the same type of service. Addressing this issue, we introduced models, algorithms and supporting platform for analyzing and quantifying the elasticity capabilities of cloud services, and dependencies among them. We have shown that our approach significantly reduces this complexity, by providing a set of system configuration recommendations. The recommendations consider both the system's requirements over resources, performance, and cost, and the envisioned system elasticity.

CHAPTER 5

Monitoring Elastic Cloud Systems

In the previous chapter we have focused on designing elastic systems from proprietary software and a mix of different types of cloud services, from IaaS to PaaS. In this chapter we focus on monitoring such elastic systems, deployed and running in cloud.

5.1 Introduction

Elastic systems can re-configure individual components/units at run-time, due to various requirements. As discussed in Chapter 2, such run-time re-configuration usually focuses on: (i) vertical scaling, adding/removing resources to existing components; or (ii) horizontal scaling, duplicating components/instantiating more instances of the same service type. This generates two main issues from monitoring point of view. First, as system units using virtual machines or other types of cloud services can dynamically appear/disappear at run-time, a monitoring system for elastic systems must be able to dynamically start/stop collecting metrics from individual cloud services, especially virtual machines. Secondly, as virtual machines themselves are allocated/deallocated dynamically at run-time, if monitoring information is associated only with each virtual machine, it will be lost during scale-in operations which deallocate the used virtual machines. While existing monitoring systems such as JCatascopia [68], Ganglia[29] or Nagios[30] can address the first issue, addressing the structure volatility issue requires a different approach in managing collected monitoring information.

In this chapter we make the following contributions:

- a model for representing multi-level monitored information of cloud systems

[28]The contributions from this chapter where originally presented in [33] and [1].

[28]In the original content ([33] and [1]) we refer to cloud systems as cloud services. For consistency, in this chapter we maintain the terminology used up to this point in the thesis.

[29]`http://ganglia.sourceforge.net/`

[30]`https://www.nagios.org/`

- a domain-specific language for constructing multi-level monitoring snapshots for elastic systems

- a platform for monitoring elastic cloud systems, to be used by system developers, owners, or elasticity controllers

5.2 Roadmap

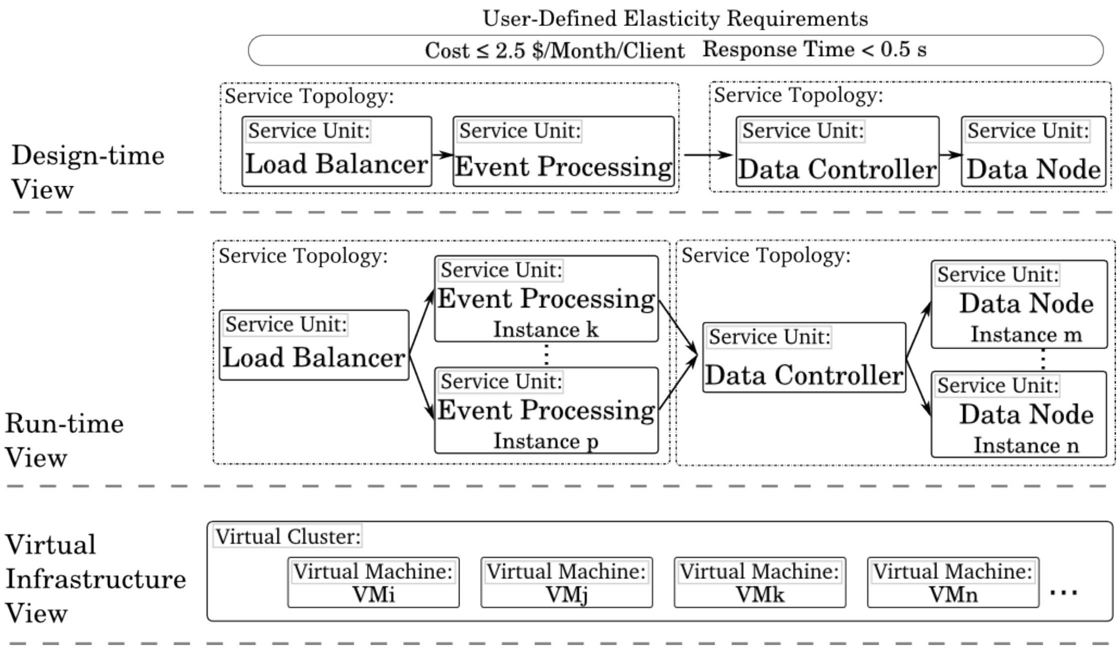
The rest of this chapter is structured as follows. Section 5.3 expands the motivation. Section 5.4 introduces our approach on classifying monitoring information according to the multidimensional elasticity principle defined by Dustdar et al. [15]. Section 5.4.1 presents our model for capturing multi-level information about elastic systems. Section 5.4.2 introduces our approach and language for custom composition of monitoring information in higher level information. Section 5.4.3 describes our prototype exposing elasticity analytics as a service, and evaluates our approach by monitoring the elastic DaaS system introduced in Chapter 3. Section 5.4.6 concludes the chapter.

5.3 Motivation

In this scenario we assume the units of the `Local Processing Topology` of the elastic DaaS are not deployed in the cloud, and instead run on IoT gateways located closer to the IoT sensors. Thus, in the rest of the chapter we focus on the cloud-based part of the DaaS, the `Event Processing` and `Data End` topologies.

In general, we have can see cloud systems from three perspectives, focusing on: (i) design-time, (ii) run-time, (iii) or the virtual infrastructure. The perspectives are exemplified for the DaaS in Figure 5.1. The design-time perspective contains the system structure, i.e., the whole cloud system with its topologies and units, and the system requirements. At this level usually requirements are defined over the system's performance, resource usage or cost, either for the whole system or individual system components. Thus, this level is also crucial for elasticity controllers such as [5, 69], which reason if the whole system or particular components are fulfilling their requirements. If needed, the controllers enforce actions to bring the system back in a state with all requirements fulfilled. The second perspective focuses on the run-time structure of the system, capturing not only the system components, but also their instances and used cloud services. This view is crucial for controlling the elastic system through existing cloud system management tools such as SALSA [70] or Slipstream[31]. Such tools can dinamically allocate/deallocate system units, provisioning and managing the required cloud services. Lastly, the virtual infrastructure perspective is concerned with the cloud services used by the elastic system. This view is especially used by monitoring tools such as JCatascopia [68], Ganglia[32] or

[31]`http://sixsq.com/products/slipstream/`
[32]`http://ganglia.sourceforge.net/`



User-Defined Elasticity Requirements
Cost ≤ 2.5 $/Month/Client
Response Time < 0.5 s
Design-time View
Service Topology:
Service Unit: Load Balancer
Service Unit: Event Processing
Service Topology:
Service Unit: Data Controller
Service Unit: Data Node
Run-time View
Service Topology:
Service Unit: Load Balancer
Service Unit: Event Processing Instance k
Service Unit: Event Processing Instance p
Service Topology:
Service Unit: Data Controller
Service Unit: Data Node Instance m
Service Unit: Data Node Instance n
Virtual Infrastructure View
Virtual Cluster:
Virtual Machine: VMi
Virtual Machine: VMj
Virtual Machine: VMk
Virtual Machine: VMn

Figure 5.1: Elastic cloud system views

Nagios[33], which monitor the virtual machines, without considering the structure of the system running on top of them.

Existing cloud monitoring systems either focus on the virtual infrastructure perspective, or on the design time view (monitoring system level metrics, such as overall response time). Thus, they fail to associate the monitoring information retrieved from the virtual infrastructure to the system structure. This makes it difficult for elasticity controllers to perform higher level reasoning. To aid in the process of controlling elastic systems, we first collect monitoring information, map it to the system structure, and extract from it higher level information used to characterize the system's behavior. While analyzing the behavior of a monitored element (E.g., load balancer or data node) is already challenging, we cannot just deal with single instances. We must focus on understanding both the behavior of individual instances, and the overall behavior of the system components. For example, for an `Event Processing Unit` we need to understand if a particular unit instance has too high response time. But we also need to understand if on average, the response time of the unit is within acceptable boundaries. Further, depending on the system requirements, if requirements are defined over entire topologies (e.g, over a whole data end) or the entire system, the overall behavior of the topology or system must be determined, to provide support to elasticity controllers in their decision making.

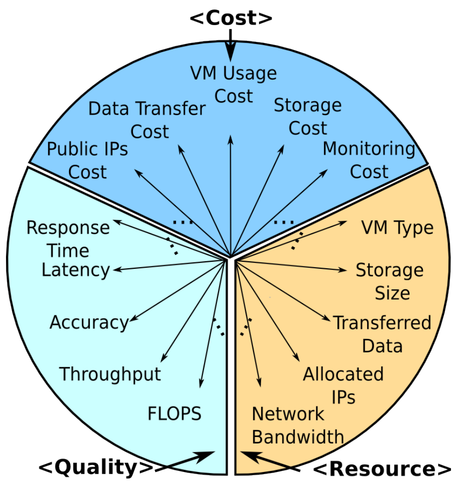


Figure 5.2: Elasticity dimensions

5.4 Multi-dimensional Elasticity

In order to support and analyze the multi-dimensional elasticity of cloud systems, we categorize monitoring data in three dimensions: *Cost*, *Quality*, and *Resource* (Figure 5.2). These categories are sufficient for capturing data about any monitored element (E.g., system topology or unit) within a cloud system, and can be used for understanding the behavior of that system. For an elastic cloud system, the *Quality* dimension would capture metrics characterizing the system's quality, such as response time or throughput. The *Cost* dimension would in turn capture all metrics influencing cost, such as cost of using the virtual machine (E.g., hourly or monthly cost), cost of data transferred over the network, or separate cost of using storage (E.g., cost per each 10 GB of stored data). The *Resource* dimension metrics capture resource usage and allocation information, such as the amount of data transferred over the network.

Conceptually, to capture monitoring data associated with a monitored element at a specific time t, we define the monitoring snapshot, ms, containing monitoring data about *Cost*, *Quality* and *Resource* elasticity dimensions (Eq. 5.1).

$$ms = \{(\langle c_i \rangle, \langle q_j \rangle, \langle r_k \rangle, t) |\ c_i \in Cost,\ q_j \in Quality,\ r_k \in Resource\} \tag{5.1}$$

5.4.1 Elastic cloud system model

Structuring monitoring information and describing the behavior of elastic systems from all three previously mentioned perspectives requires appropriate models. To this end we leverage the model defined in [5] and presented in Chapter 2.7. We represent a cloud system composed of topologies, each topology containing system units, deployed on virtual machines belonging to virtual clusters (Figure 5.3). A *System Unit* is a functional element

[33] `https://www.nagios.org/`

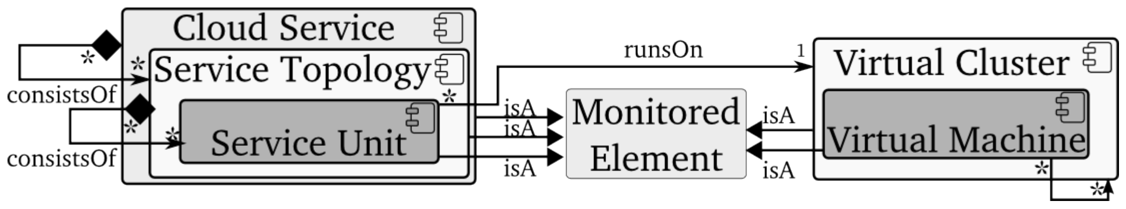


Figure 5.3: Elastic cloud system monitoring model

of a cloud system, which can run inside a virtual machine, either standalone or along other system units. A *System Topology* does not have an equivalent in the virtual cloud infrastructure, instead it logically groups related system units, e.g., a Data Controller and Data Nodes belonging to a distributed Data End. Conceptually, a system topology would contain all system units that have a related logical role. E.g., a data end topology containing all units having a role in providing data storage and management services. Instances of units run on *Virtual Machines* belonging to *Virtual Clusters* (E.g., network clusters, or different cloud providers). Thus, our model can represent cloud systems which are composed of other sub-systems (system topologies), deployed in federated cloud environments or in different availability zones (virtual clusters). System components are considered *Monitored Elements*, and each can have associated a list of metrics.

The structure of the cloud system is defined by users or elasticity controllers, and used for structuring monitoring information and analyzing the system's behavior. For easy integration with other tools, we use an XML-based representation format for describing the system's structure (Listing 5.1). Each monitored element is defined with the `MonitoredElement` tag, and contains 2 mandatory properties, `id`, and `level`, and an optional `name` property. The `level` indicates if the monitored element is one of `VM, VIRTUAL_CLUSTER, SYSTEM_UNIT, SYSTEM_TOPOLOGY, or SYSTEM`. Each monitored element can contain zero or more monitored elements with a lower level. For the `VM` level, the `id` is the IP of each virtual machine. At run-time, this description would be managed by a controller or human, and updated with the added/removed virtual machines that run system units instances, after each scaling in/out action.

Listing 5.1: XML-based dependency model

```
<MonitoredElement id="" level="SYSTEM">
  <MonitoredElement id="" level="SYSTEM_TOPOLOGY">
      <MonitoredElement id="" level="SYSTEM_UNIT">
        <MonitoredElement id="10.x.x.x" level="VM"/>
     </MonitoredElement>
...
```

5.4.2 Cross-layered metric composition

Existing tools for monitoring cloud systems such as Ganglia[34] or JCatascopia [68] are agnostic of the system's logical structure and associate monitoring information with the individual virtual machines running system units' instances. As one unit can have multiple instances distributed among different VMs, associating information with VMs does not give any indicator about the overall behavior of the system units, information crucial for elasticity controllers. Therefore, to obtain a complete view over the cloud system behavior, from low level metrics to higher ones, we compose monitoring snapshots following the previously presented cloud system model (Figure 5.3).

Metric composition language

Obtaining higher level information from collected monitoring snapshots through snapshot composition introduces the problem of combining/aggregating metrics. Depending on the type of metrics, a valid composition of two metrics might involve different operations, depending on the metric type and the information which must be obtained through composition. For example, one would average response time for a system unit to get an indicator of the system's performance, but sum up its throughput to get overall number of performed operations. Depending on particular requirements, beside total throughput of a system unit, one might also require the maximum throughout achieved by each instance of the unit, to understand the maximum achievable performance. Thus, for providing support for extracting custom information about the behavior of the system, we define an XML-based domain-specific language. The language describes metric composition rules as a cascading sequence of operations which apply one or more operators over one or more operands. An operand can be a static value or a metric reference, and the set of available operations is presented in Table 5.1. The language grammar is shown in Listing 5.2, and the XML format for specifying rules is shown in Listing 5.3. For each rule there are at least one *reference metric*, one *resulting metric*, and several *operations*. The *reference metric* is used as a base for computing the composite metric, and it is searched in the metrics of the target monitored element children having the *TargetMonitoredElementLevel*. The *resulting metric* defines the name and unit of the composite metric being created. Defining the composition rules requires domain specific knowledge, such as knowing which operation is appropriate for which metric (E.g., SUM

[34] http://ganglia.sourceforge.net/

Listing 5.2: Metric composition rules grammar

```
rule      := operation "=>" metric
operation := operator "(" operand { "," operand } ")"
operator  := "+"|"-"|"*"|"/"|"AVG"|"SUM"|"MAX"|"MIN"|"SET"|"KEEP"
operand   := metric | number | string
metric    := name, measurementUnit, [monitoredElementID],
             monitoredElementLevel
```

Operation	Description
+, -	Adds/Subtracts a value (metric or static) to another value (from another operation or metric)
*, /	Multiplies/Divides a value (from another operation or metric) with another value (metric or static)
AVG	Computes the average value of a sequence of metric values
SUM	Computes the sum of a sequence of metric values
MAX, MIN	Extracts the maximum/minimum value from a sequence of metric values
SET	Assigns a static value to a metric
KEEP	Returns unchanged a metric value or the result of another operation

Table 5.1: Metric composition operations

Listing 5.3: XML-based metric composition rule format

```
<CompositionRule TargetMonitoredElementLevel="LEVEL">
   <TargetMonitoredElementID>id</TargetMonitoredElementID>
   <ResultingMetric type="METRIC_TYPE"
                    measurementUnit="text" name="text"/>
   <Operation MetricSourceMonitoredElementLevel="LEVEL"
              type="OPERATION␣TYPE">
      <ReferenceMetric type="METRIC_TYPE"
                       measurementUnit="text" name="text"/>
      <SourceMonitoredElementID>ID</SourceMonitoredElementID>
   </Operation>
</CompositionRule>
```

for cost, AVG for response time). The metric XML representation of the composition rules can also be produced by using a fluent API implemented based on the grammar described in Listing 5.2.

Monitoring information structuring and enrichment

Monitoring snapshots capture metrics, but not the same as elasticity requirements defined by the user, or at the same level. For example, a monitoring snapshot might include `throughput` per VM, while the elasticity requirements might target `numberOfClients` over the whole system. Using the above metric composition language, metrics collected from the VM level can be associated to the upper system levels. Moreover, new metrics can be created, according to individual system requirements, applying composition operations, as follows. First, the system level for which the new metric is created should be specified using the `TargetMonitoredElementLevel`, and can be one of `VM, VIRTUAL_CLUSTER, SYSTEM_UNIT, SYSTEM_TOPOLOGY, or SYSTEM`, according to the system dependency model. Optionally, if the rule should be applied on a specific monitored element, (a unit, a topology or the whole system), its `ID` can be specified with the `TargetMonitoredElementID` tag. Information about the new metric to be created must be specified using the `ResultingMetric` tag, including name, measurement

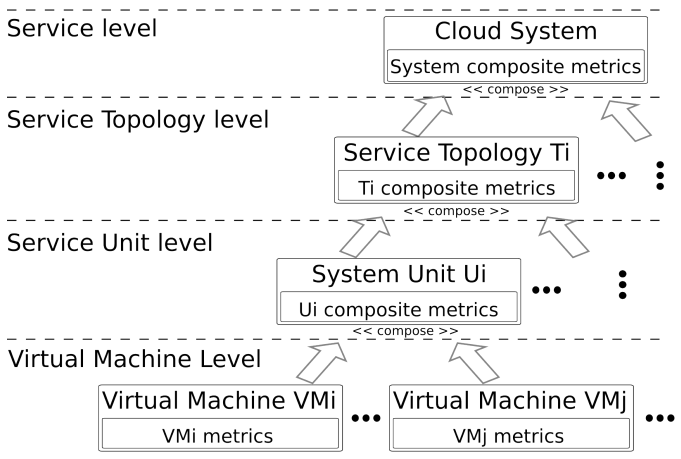


Figure 5.4: Multi-level metric composition process

unit, and type (one of elasticity dimensions). Then, a cascading list of operations can be defined, each operation having a `type`, defined in Table 5.1, and a `ReferenceMetric`, indicating the metric over which the operation should be applied. If the operation should be applied over metrics from a specific monitored element, its ID can be specified with the `SourceMonitoredElementID` tag.

Metric composition process

The metric composition process has as aim to create from lower level metrics, higher level ones, offering information better suited for providing support to system elasticity control (Figure 5.4). Algorithm 3 describes our process of constructing multi-level monitoring snapshots by applying custom metric composition rules. The rules are used to structure monitoring information, compute cost, and/or enrich monitoring information. The *BuildMonitoringSnapshot* function applies metric composition rules bottom-up on the system's monitored elements, first at the *virtual machine* level, then at *system unit*, *system topology* and *system* levels, creating the cross-layer monitored snapshot. Each composition rule is applied over its target monitored elements belonging to that specific level (Lines 6-12). The *ApplyCompositionRule* function handles the application of metric composition rules. First, if a rule specifies a specific monitored element from which the metric values are to be collected, the element is considered as the source for monitored values. Otherwise, monitored values are extracted from the element itself, or from all children of the element which belong to the rule's metric *source level* (Line 19). From all monitored element data sources, the source metric is searched, and its values collected (Lines 20-22). Over the collected values, the rule's composition operations are applied sequentially, the output of a previous operation acting as input for the next (Lines 24-26). The output of the last operation is returned as result of the metric composition process.

Algorithm 3 Cross-layered metric composition algorithm

Input: *system*,*compositionRules*,*monitoring*

Output: *snapshot* - multi layer monitoring snapshot

```
 1: function BuildMonitoringSnapshot(system, compositionRules, monitoring)
 2:     snapshot = createInitialSnapshot(monitoring)
 3:     for all elementLevel in [VM,SystemUnit, SystemTopology, System] do
 4:         levelElements = system.getElementsOnLevel(elementLevel)
 5:         levelRules = compositionRules.getRulesOnLevel(elementLevel)
 6:         for all rule in levelRules do
 7:             targetElements = getRuleTargets(rule, levelElements)
 8:             for all element in targetElements do
 9:                 compositionValue= ApplyCompositionRule(rule, element, snapshot)
10:                 snapshot.addNewMetric(rule.metric,compositionValue, element)
11:             end for
12:         end for
13:     end for
14: return snapshot
15: end function
16:
17: function ApplyCompositionRule(rule, monitoredElement, snapshot)
18:     sourceElements = getSourceElements(rule.metric, monitoredElement)
19:     values = []
20:     for all element in sourceElements do
21:         values.add(snapshot.getMetricValue(element,rule.metric))
22:     end for
23:     for all operation in rule.operations do
24:         values = operation.apply(values)
25:     end for
         return values
26: end function
```

5.4.3 Evaluation

In this section we apply the introduced mechanism for composing monitoring information on the DaaS system presented in Chapter 3. In this scenario we assume the units of the `Local Processing Topology` are not deployed in the cloud, and instead run on IoT gateways located closer to the IoT sensors. Thus, in the rest of the chapter we focus on the cloud-based part of the DaaS, the `Event Processing` and `Data End` topologies.

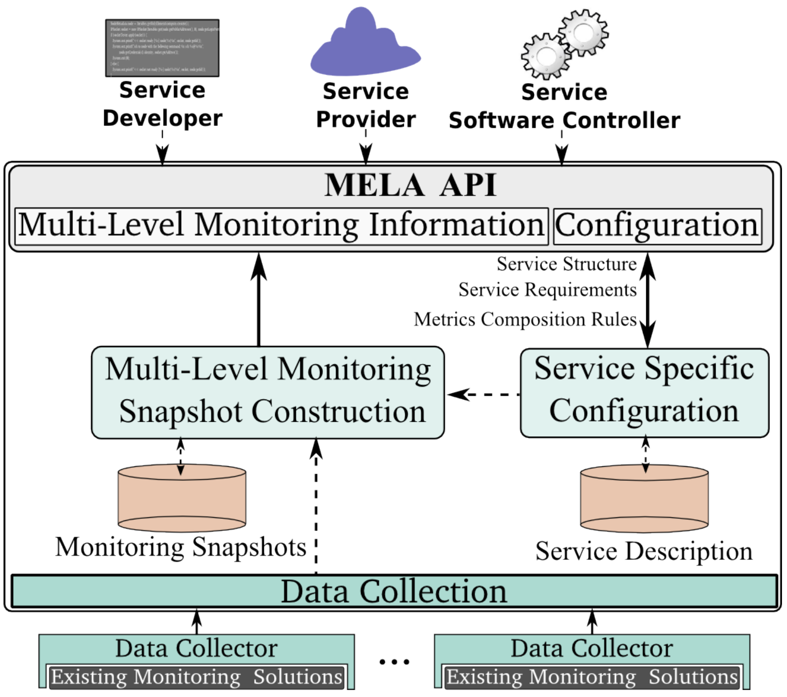

Service Developer
Service Provider
Service Software Controller
MELA API
Multi-Level Monitoring Information
Configuration
Service Structure
Service Requirements
Metrics Composition Rules
Multi-Level Monitoring Snapshot Construction
Service Specific Configuration
Monitoring Snapshots
Service Description
Data Collection
Data Collector
Existing Monitoring Solutions
...
Data Collector
Existing Monitoring Solutions

Figure 5.5: MELA overview

5.4.4 Prototype: MELA Elasticity Analytics as a Service

Based on our concepts in Section 5.4.2, we develop MELA[35], an elasticity analytics as a service (Figure 5.5). MELA contains a core *MELA Service*, and *Data Collector* nodes. A *Data Collector* node is a customizable component that gathers, from existing monitoring solutions, data associated with a dependency model level or monitored element (E.g., `responseTime` or `throughput` for the `Event Processing` system topology), and sends it for processing and analysis to the *MELA Service*.

Monitoring data from existing monitoring systems is usually associated with a single level, e.g., virtual infrastructure, system topology or system unit. An important MELA feature is the linking of these levels, implying a configuration step using the *Configuration API*, defining the system structure, the metrics composition rules to be applied for the monitored elements at each level, and the system requirements. The *Multi-level Monitoring Snapshot Construction* component uses the system specific configuration to provide composite monitoring snapshots from data collected from the *Data Collector* nodes. Using the *Multi-level Monitoring Information API*, the MELA user can retrieve the elasticity space and pathway for the whole system, or specific monitored elements.

As MELA is designed to analyze elastic systems which can allocate/deallocate virtual machines depending on elasticity requirements, it provides two mechanisms for managing this volatile system structure. The default mechanism is to delegate this responsibility

[35]Prototype and supplement materials at `http://tuwiendsg.github.io/MELA/`

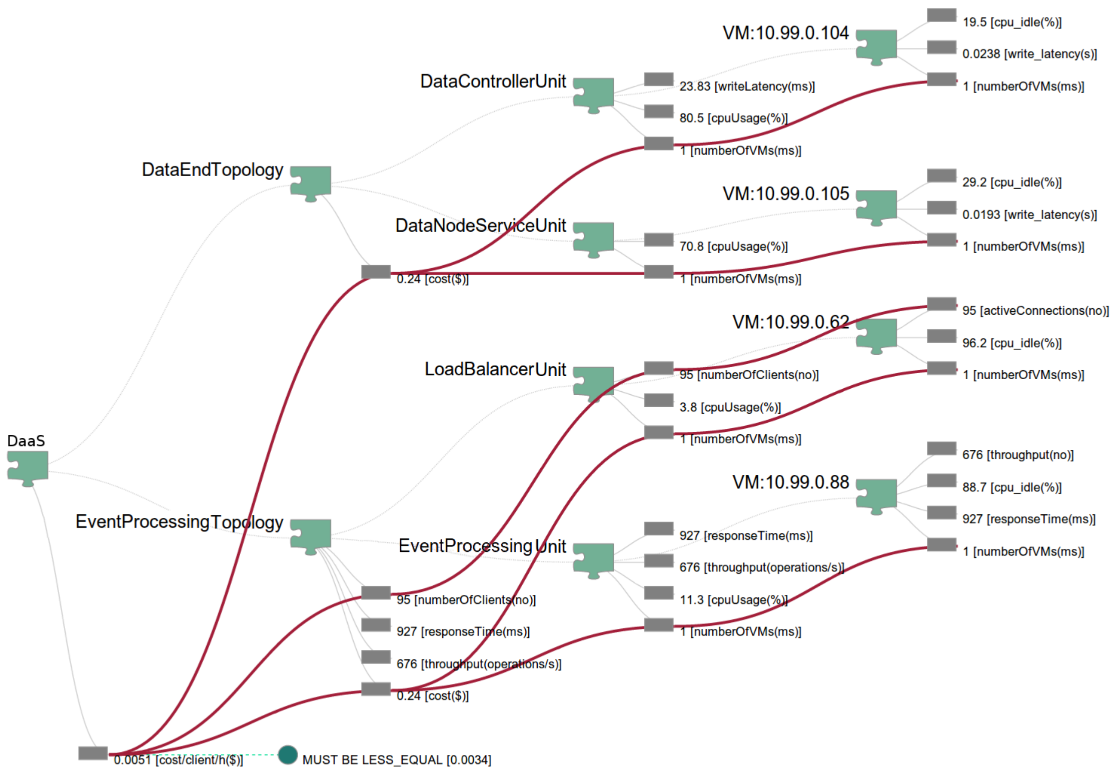


Figure 5.6: MELA visualization of multi-level monitoring data with complex metric composition

to the MELA user (cloud system developer/provider or controller), which, in this, case uses the MELA API to update the system structure after a new virtual machine running system units has been added or removed. The second behavior is to let MELA detect automatically when a new virtual machine has been added or removed to which system units the machines belong. This behavior is achieved by reporting the system units' ids hosted by a virtual machine using a virtual machine metric, and configuring MELA to use the metric to automatically update the system structure.

As we expect different cloud systems to use a wide range of monitoring mechanisms, we provide to MELA several adapters for collecting monitoring information. A poll-based adapter is implemented for Ganglia[36] and JCatascopia [68]. Additionally, a generic push-based adapter is provided, exposing a queue for receiving data.

5.4.5 Monitoring elastic cloud systems

Using the cross-layered metric composition mechanism, the MELA user (system developer/provider) starts by defining a composition rule for extracting the cost of running the system in cloud, customizing the template provided in Listing 5.3. The overall system

[36] http://ganglia.sourceforge.net/

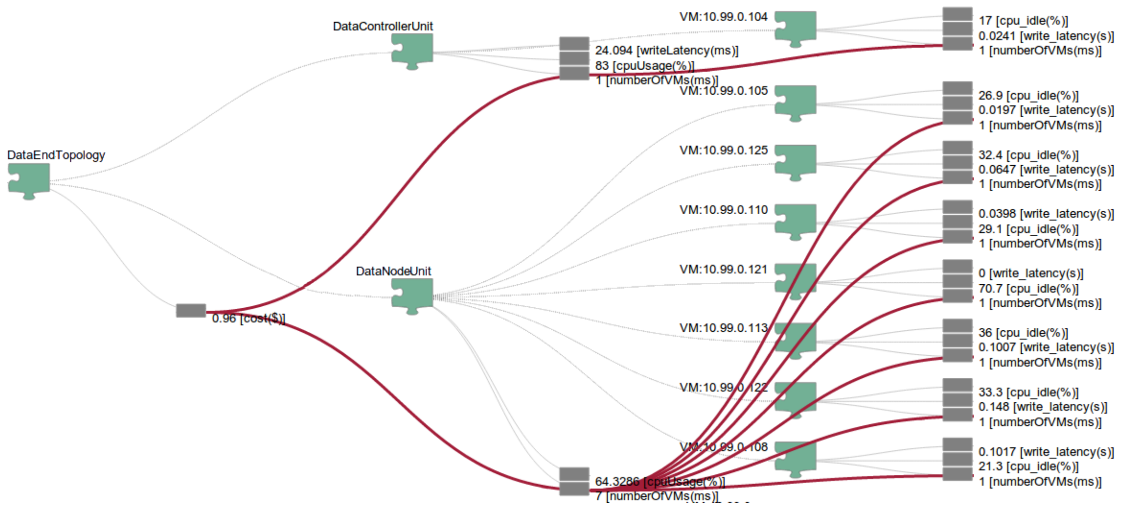


Figure 5.7: Complex metric composition for scaled out Data Node

cost metric is defined as a composition of the cost of running each virtual machine, assumed 0.12 $ per VM per hour. MELA allows metric composition rules to be defined for any service level with data from any of its children level elements. The MELA user composes the `cost` of each Service Topology instance by multiplying the cost per virtual machine with the sum of the `numberOfVMs` (number of service unit instances) of each Service Unit belonging to the Service Topology. As the service provider is interested in the overall cost per client, additional metric composition rules are defined. A rule propagates the `activeConnections` metric from the virtual machines running the *LoadBalancer* system unit to the System Unit level as a renamed `numberOfClients` metric. The new `numberOfClients` metric is further propagated to the system unit's parent *Event Processing* topology. Obtaining the `cost/client/h`, a *Cloud System* level rule sums the `cost` of the children System Topology instances, and divides it by the `numberOfClients` metric from the *EventProcessing* topology. Figure 5.6 shows with thick lines how, using MELA, such a complex `cost/client/h` metric is composed and evaluated by combining available metrics with additional cost information.

To monitor the performance of the DaaS, the MELA user extracts for the *Data Controller* unit its `writeLatency` by averaging the *write_latency* reported by all its instances running in virtual machines. For the *Event Processing* unit the average `responseTime` and total `throughput` are extracted from its virtual machines, and propagated to the *Event Processing* topology. Additionally, a metric composition rule is defined to extract the average `cpuUsage` for all system units' instances. Figure 5.6 shows a snapshot of the DaaS monitoring information enriched using MELA. The snapshot contains the simple and composite metrics created through custom metric composition.

To better understand the importance of metric composition, Figure 5.7 depicts the `Data End` topology, with 7 running instances of the `Data Node`. In this case, metric composition provides the means of obtaining a higher level view over the overall behavior of the `Data Node`, in this case at the service unit level. Without metric composition, we could only reason at the level of individual `Data Node` instance, VMs in this case. When considering that elastic systems can have multiple elastic units, reasoning only at the instance level significantly hinders the control of such services, as the controller is not able to understand the contribution of each instance to the overall behavior of the unit.

5.4.6 Conclusions

In this chapter we have focused on monitoring elastic systems, dealing with their structure volatility. We have introduced a model, technique and supporting platform for composing system metrics, towards obtaining the required information at the needed level. The information is to be used in run-time control of elastic systems. We applied our approach for composing metrics collected from multiple instances of the same unit in order to obtain the overall unit behavior. We further aggregated the unit metrics into topology and system level ones, providing a multi-level behavioral view over the behavior of the monitored elastic system.

We have shown that by providing higher-level composite metrics, such as `cost/client/h`, structured after the system structure in topologies and units, MELA facilitates system behavior analysis. The introduced language for specifying metric composition rules enables users to apply our approach to any monitoring information and elastic system, maximizing the approach's applicability. Using data retrieved from MELA, system controllers can analyze the system's behavior at multiple levels, from simple virtual infrastructure metrics, to system-level metrics composed of other simple or composite metrics.

CHAPTER 6

Analyzing Elasticity Space and Pathway of Cloud Systems

In this chapter we focus on analyzing the behavior of running elastic cloud systems, towards providing support for refining the system's elasticity requirements. The analysis is based on the multi-level monitoring information retrieved using the approach introduced in the previous chapter.

6.1 Introduction

As described in Chapter 3, elastic systems can be reconfigured during run-time, according to certain requirements with respect to their resource usage, performance, or cost. However, to achieve a system behavior in which all system requirements are fulfilled, one cannot assume that individual system units can be controlled individually. For example, if a requirement over the response time of the DaaS system from Chapter 3 is violated, there might be a need to reconfigure both the event processing topology, and the data end. The whole set of requirements over all system's units might not be known in advance. This can be due to the system's complexity, limited user knowledge of the overall system behavior, or other factors. Furthermore, it might not be clear how to determine the proper cost and quality indicators and their boundaries, and utilize them for optimizing and controlling the system during run-time.

Capturing, describing and analyzing the behavior of elastic cloud systems is crucial both for developers and software controllers. Developers build and optimize elastic systems. In turn, software controllers change the systems' topology at run-time, enforcing user-defined elasticity requirements. Thus, elasticity controllers such as introduced by

[36]The contributions from this chapter where originally presented in [33] and [1].

[36]In the original content ([33] and [1]) we refer to cloud systems as cloud services. For consistency, in this chapter we maintain the terminology used up to this point in the thesis.

Copil et al. [5] or Ming et al. [69] need a mechanism for extracting elasticity characteristics. Such a mechanism should be used to refine user-defined elasticity requirements and predict the service behavior, leading to better service control and quality.

In this chapter we make the following contributions:

- the concepts of *elasticity space* and *elasticity pathway* for analyzing elasticity of cloud systems at multiple levels
- customizable mechanisms for extracting runtime boundaries over the cloud system's behavior while fulfilling user-defined elasticity requirements

6.2 Roadmap

The rest of this chapter is structured as follows. Section 6.3 presents the motivation and research problems. Section 6.4 introduces the concepts of elasticity space and pathway, together with our approach for analyzing elasticity of cloud systems. Section 6.5 presents our prototype and experiments. Section 6.6 concludes the chapter.

6.3 Motivation

The whole set of requirements over all system's units might not be known, due to the system complexity, limited system knowledge of the overall system, or other factors. Furthermore, it might not be clear how to determine the proper cost and quality indicators and their boundaries, and utilize them for optimizing and controlling the system at runtime. Considering the DaaS introduced in Chapter 3, while the required system response time is known, there is no knowledge about what data latency is required from the data end to ensure the required response time. Existing monitoring and analysis tools can present metrics related to performance, cost, or resource usage of the whole cloud system as in Singh et al. [71], or Trihinas et al. [68], or from the underlying virtual infrastructure as in Wang et al. [72] and Shicong et al [73]. However, they do not provide a cross-layered, multi-level system elasticity behavior picture, thus hindering the discovery of the cause for system requirements violations. Managing elastic cloud systems would benefit from a multi-level monitoring and analysis view, which provides means for reasoning about the system behavior at multiple levels. In particular, we argue that in order to understand elastic cloud systems, we need to investigate new concepts that can be used to characterize the cloud system's elastic behavior based on multi-dimensional monitoring data.

In this chapter we introduce the concepts of *elasticity space* and *elasticity pathway*, and apply them in evaluating elasticity of cloud systems. First, the elasticity space is used in capturing the behavior of elastic cloud systems. Second, the elasticity pathway characterizes the systems' evolution through the elasticity space, and can be used to predict the systems' behavior. We apply our technique for determining the elasticity space and pathway over the multi-level monitoring snapshots constructed with the approach presented in the previous chapter. We extend MELA with *elasticity space* and *elasticity pathway* analysis. We allow cloud system developers, providers and automatic controllers to analyze their system behavior from the whole system level to the underlying virtual infrastructure, extracting characteristics and providing crucial insights in their elasticity.

6.4 Elasticity Space and Pathway of Cloud Systems

In this section we introduce the concepts of *elasticity space* and *elasticity pathway* for analyzing and characterizing the behavior of elastic cloud systems.

6.4.1 Elasticity Boundary

Monitoring snapshots capture metrics, but do not provide information about boundaries over the metric's values in which user-defined elasticity requirements are fulfilled. Therefore, in order to analyze the behavior of elastic a monitored element, we represent metric boundaries using the *elasticity boundary* concept:

Definition 1 *An* elasticity boundary *describes the upper and lower bound over a set of metrics for a monitored element.*

Conceptually, an elasticity boundary, *elBoundary*, is defined as follows:

$$elBoundary = (\langle c_i^u, c_i^l \rangle\rangle, \langle q_j^u, q_j^l \rangle\rangle, \langle r_k^u, r_k^l \rangle\rangle) \tag{6.1}$$

where c_i^u and c_i^l denote the upper bound and the lower bound of metric $c_i \in Cost$, respectively, q_j^u and q_j^l for $q_j \in Quality$, and r_k^u and r_k^l for $r_i \in Resource$.

We use the elasticity boundary to capture both user-defined elasticity requirements (*user-defined elasticity boundary*), and detected/evaluated elasticity requirements (*evaluated elasticity boundary*). Using the user-defined elasticity boundary we represent requirements over the user's elastic system's cost, quality and resources. The user-defined requirements expressed as boundaries indicate the c_i, g_j, r_k under which the cloud system should behave. From these parameters, we evaluate collected monitoring information and determine elasticity boundaries for all monitored elements of a cloud system. Thus, from a set of supplied requirements for a particular monitored element (E.g., cloud system and system unit), we determine the requirements for all cloud system monitored elements. Thus, we provide information to control the elasticity from the whole elastic system to individual units.

6.4.2 Elasticity Space

We start with a given set of monitoring snapshots and user-defined elasticity boundaries. For supporting run-time control of system's elasticity, we need to understand when a monitored element is in elastic behavior. We must further understand if its behavior violates the user-defined elasticity boundaries, and if we can characterize the system behavior using some specific "pathways". Naturally, we expect that the meaning of "elasticity" will depend on the types of monitored elements, their runtime settings and requirements. To this end, we define the concept of *elasticity space* to determine and evaluate when a monitored element is in elastic behavior:

Definition 2 *An* elasticity space *captures all runtime metrics described in the user-defined elasticity boundary and all other metrics influencing the user-defined elasticity boundary.*

To respect its elasticity boundaries, an elastic system must scale out/in its cost, quality and resources at run-time. By allocating/deallocating cloud services, the system copes with variations in pricing, quality and load. We mean by elastic behavior the behavior of a system which is dynamically reconfigured at run-time by software controllers. Formally, let $f_{elSpace}$ be an elasticity space function, and $MS = \{ms_i\}$ be the set of monitoring snapshots. Then an elasticity space *elSpace* can be defined as: $elSpace = f_{elSpace}(MS)$. A $f_{elSpace}$ has to perform two steps: (i) detect when an elastic behavior starts and stops, and (ii) extract monitoring data describing the system's behavior while respecting the user-defined elasticity boundaries. In principle, there could be several elasticity space functions, which can be developed for and applied to different types of monitored elements. E.g., to specific types of system units, topologies, or the whole system.

An elasticity space function is designed to extract useful information about the overall behavior of the cloud system when elasticity requirements are fulfilled. For example, given a user-defined elasticity requirement over *cost/client/h*, an elasticity space might contain only the *throughput* and *cost/VM/h* metrics from which the *cost/client/h* targeted by requirements can be determined. It would not include metrics that have no impact on the *cost/client/h*. Thus, using the elasticity space, one can determine the elasticity boundaries to be enforced on the metrics that influence the user-defined elasticity requirements. Moreover, one can analyze if the behavior of an elastic cloud system is within expected user-defined elasticity boundaries by checking the elasticity boundaries of its elasticity space. For example, the upper elasticity boundary of the *cost/client/h* from the determined elasticity space could have a different value than expected by the user.

6.4.3 Elasticity Pathway

While the elasticity space enables cloud system elasticity analysis, it does not provide insight into relationships and dependencies between metrics influencing the elastic behavior over time. For example, *throughput* and *cost/VM/h* might or might not follow a linear relationship. In order to characterize the elastic behavior from specific views/perspectives over a cloud system, we define the concept of *elasticity pathway.*

Definition 3 *Given a specific view on metrics $V = \{m_1, m_2, \cdots, m_n\}$, an elasticity pathway for V characterizes the elasticity relationship among m_i over the time.*

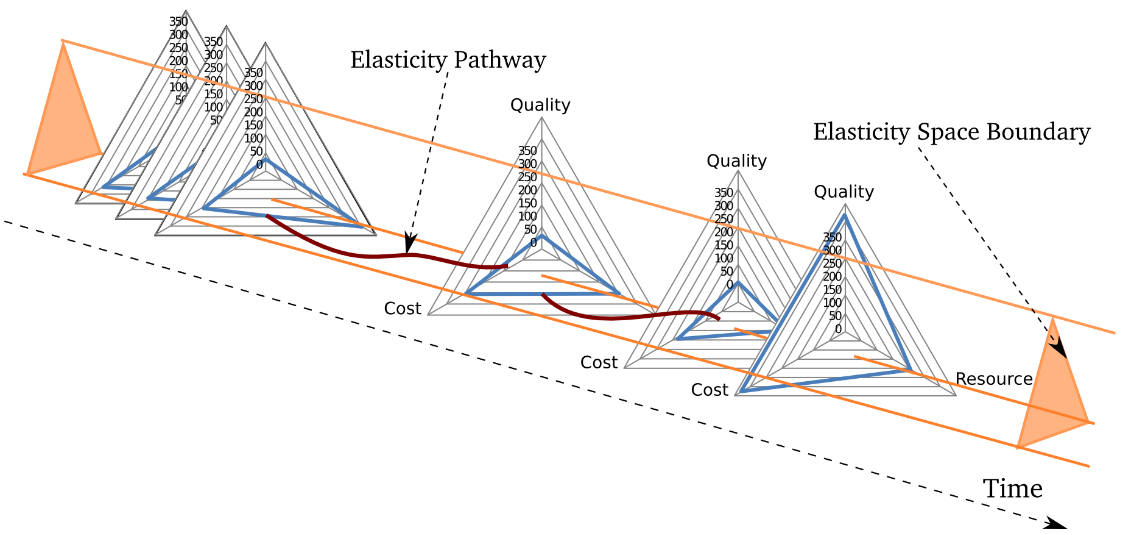


Figure 6.1: Elasticity Space and Pathway concepts

A view over a set of metrics is a subset of metrics chosen for analysis, which potentially influence the user-defined requirements. For example, given a user-defined elasticity requirement over $cost/client/h$, a view could contain the *throughput* and $cost/VM/h$ metrics from which the $cost/client/h$ can be derived. An elasticity pathway function is designed to perform a complex evaluation of the cloud system behavior, determining characteristics that can be used to predict the system's behavior.

Formally, an elasticity pathway $elPtw$ is determined by a function f_{elPtw} which takes as input an elasticity space $elSpace$ and a view V over the space's metrics, and returns another function describing behavioral patterns or characteristics of the monitored element: $elPtw = g(V) = f_{elPtw}(elSpace, V)$. Various elasticity pathway functions can be defined over the elasticity space, enabling space analysis from multiple perspectives. As the elasticity pathway function is applied over an elasticity space, the quality of the determined elasticity pathway is heavily influenced by the data in the elasticity space.

The *elasticity space* and *elasticity pathway* concepts are conceptually visualized in Figure 6.1. Informally, one can consider the elasticity space as the set of value ranges taken by each system metric, and the pathway as the "way" the space changes, i.e., the value ranges change, as time passes during the system's run-time.

6.4.4 Multi-dimensional Elasticity Space and Pathway analysis

Based on the metric composition mechanism defined in Chapter 5, we provide an analysis mechanism for determining the elasticity space and pathway at different system levels, providing a multi-level decomposition of the cloud system behavior. The behavior decomposition is beneficial to elasticity controllers. Controllers can use our approach to detect which monitored element from which system dependency model level violates system requirements, and reason in terms of metrics located at that particular level. Providing a

Algorithm 4 Evaluating elasticity space

Input: *system*, *requirements*, *metricCompositionRules*, *monitoringData*

```
1: function EvaluateElasticitySpace(system, requirements, metricCompositionRules, monitoringData)
2:     snapshot = BuildMonitoringSnapshot(system, metricCompositionRules, monitoringData)
3:     elasticitySpace = getElasticitySpaceLearnedSoFar()
4:     for all snapshot in monSnapshot.elements do
5:         elementElSpace = elasticitySpace.get(snapshot.element)
6:         elementRequirements = systemRequirements.get(snapshot.element)
7:         if snapshot.fulfillsAll(elementRequirements) then
8:             elementElSpace = elementElSpace.updateElasticityBoundaries(snapshot)
9:         else
10:            elementElSpace = elementElSpace.store(snapshot)
11:        end if
12:    end for
13: return elasticitySpace
14: end function
```

complete view over the behavior of cloud system enables system providers/developers and software controllers to reason on the same system using separate perspectives.

Elasticity Space analysis

Algorithm 4 describes the process of evaluating the elasticity space, in which custom elasticity space functions can be used to update the elasticity boundaries (`updateElasticityBoundaries` function). The system structure, elasticity requirements, metric composition rules, and collected monitoring data are received as input. For each new collected monitoring snapshot, the cross-layered enriched monitoring snapshot is computed by applying metric composition rules in Line 2. For each system monitored element, the elasticity space is evaluated by updating the elasticity boundaries. Lines 7-11 check if all elasticity requirements are fulfilled, and if yes, the upper and lower elasticity boundaries are updated. If not, the monitoring snapshot is stored for future reference.

We understand that analyzing and controlling elasticity of cloud systems can be a continuous process in which elasticity requirements or even the system structure are refined during run-time. We define the following changes that have impact on the process of determining the elasticity space and boundaries: (i) metric composition rules, (ii) elasticity requirements, and (iii) system structure. In the first case, we support addition and removal of metrics by altering the supplied composition rules, the elasticity space evaluation continuing by enriching the space determined so far using the metrics available in each new monitoring snapshot. In the latter 2 cases, the whole elasticity space is recomputed based on historical monitoring information, updating all elasticity boundaries.

In the current prototype, we implement an *elasticity space function* which determines as space boundaries the maximum and minimum encountered metric values when the user-defined elasticity requirements are respected. The function uses as input the user-defined elasticity requirements for the whole cloud system, and is applied for all system topologies, units, and unit instances.

Elasticity Pathway analysis

Based on an elasticity space, *elasticity pathway* functions can be defined for each system unit, topology, or whole system, enabling custom analysis of system behavior. Based on the determined elasticity space, for the supplied `monitored element`, we apply its elasticity pathway functions. We analyze its behavior and extract behavioral characteristics, using Algorithm 5.

For the current prototype, we adapt as *elasticity pathway* function an unsupervised behavior learning technique using self-organizing maps (SOMs) presented by [74], and classify monitoring snapshots by their encounter rate in DOMINANT, NON-DOMINANT, and RARE. Such a pathway is important for understanding if the regular behavior of the system respects user-defined elasticity requirements. As SOMs are unsupervised neural networks that map multi-dimensional spaces into low dimensional ones, they are suitable for classifying monitored snapshots, as snapshots can contain many different metrics. Each SOM's neuron value is derived from its snapshots. Each monitoring snapshot to be classified is mapped to the group from which it has the smallest distance. With each new snapshot, the group and its SOM neighbors are updated using the function $V_{new}(group) = V_{old}(group) + A * N(group)(V(snapshot) - V_{old}(group))$, where A is a discount factor, and $N(group)$ is a neighborhood function determining the degree with which a group value is updated. In unsupervised learning the initialization of the system is important. As we classify groups after the number of snapshots mapped to them, we do not initialize the SOM with random values, as it might generate groups which are close together in value but far away in terms of location in the SOM. In such a case similar snapshots might be assigned to separate groups, diluting the number of snapshot mapped to a SOM entry. Thus, we initialize the SOM with snapshot groups having all metrics equal to 0, and rely on its self-adaptive nature to map the input data. We use a neighborhood function of 1 for the directly targeted group and of

Algorithm 5 Evaluating elasticity pathway

Input: *elasticitySpace*, *systemElement*

```
1: function EvaluateElasticityPathways(elasticitySpace, monitoredElement)
2:     elPathwayFunctions = getPathwayFunctionsFor(monitoredElement)
3:     elPathways = []
4:     for all elPathwayFunction in elPathwayFunctions do
5:         elPathways.push(elPathwayFunction(elasticitySpace))
6:     end for
7: return elPathways
8: end function
```

$1/neighbourLevel/neighboursCount$ for its neighbors. Updating the neighbors creates and updates new groups, mapping the input data better. The discount learning factor is $1/neighbourLevel$, the neighborhood is 2 and the map size is 10×10. A filtering step merges groups with same value, consolidating the monitored snapshots. The average absolute deviation AAD (the average of the absolute deviations) of the number of snapshots mapped per group is computed. Using the average absolute deviation (AAD), a group is RARE if its deviation is negative and its absolute higher than the AAD, DOMINANT if its deviation is higher than the AAD, and NON-DOMINANT otherwise. While finer grained classes can be defined, we argue that these are enough to give insight in the behavior of elastic cloud systems.

6.5 Evaluation

In this section we analyze the elasticity space and pathway of the DaaS system presented in Chapter 3. In this scenario, the DaaS provider wants to implement a 2.5$ monthly subscription for each system client, i.e., per sensor. Using various tools, the system developer describes the DaaS and its elasticity requirements. Then, the developer deploys the system, which is then automatically controlled at run-time. MELA has two important roles in this evaluation. First, it provides to the developer the ability to monitor and analyze the system's elasticity behavior (via metrics) at each system level. This is used by the developer to understand the elasticity of the system, and verify if such a pricing scheme is sustainable. Secondly, it provides structured and enriched monitoring information to be used by elasticity controllers in managing elastic systems during run-time.

As elasticity is triggered by performance/cost requirements, we simulate a load from real sensor data, following a Gaussian distribution of M2M sensors connecting to the DaaS, starting from 50 sensors per second, increasing to 350, and then decreasing again, each request requiring between 1 and 10 operations. The system VMs were deployed on our OpenStack[37] cloud. In this evaluation we use the MELA Data Collector implemented for Ganglia[38]. Generic OS level and system specific monitoring data (E.g., clients/h, throughput, response time) are retrieved using provided and custom Ganglia plug-ins.

6.5.1 Prototype

We extend MELA[39] with services for managing and retrieving the elasticity space and pathway of cloud systems (Figure 6.2). Elasticity space and pathway functions are managed through the *Elasticity Functions Management API*. The elasticity space is determined by the *Elasticity Space Analysis* MELA component, from which the elasticity pathway is determined by the *Elasticity Pathway Analysis* MELA component. Composite monitoring snapshots together with determined elasticity space are stored in a *Monitoring*

[37]`http://www.openstack.org/`

[38]`http://ganglia.sourceforge.net/`

[39]Prototype and supplement materials at `http://tuwiendsg.github.io/MELA/spaceAndPathwayAnalysisService.html`

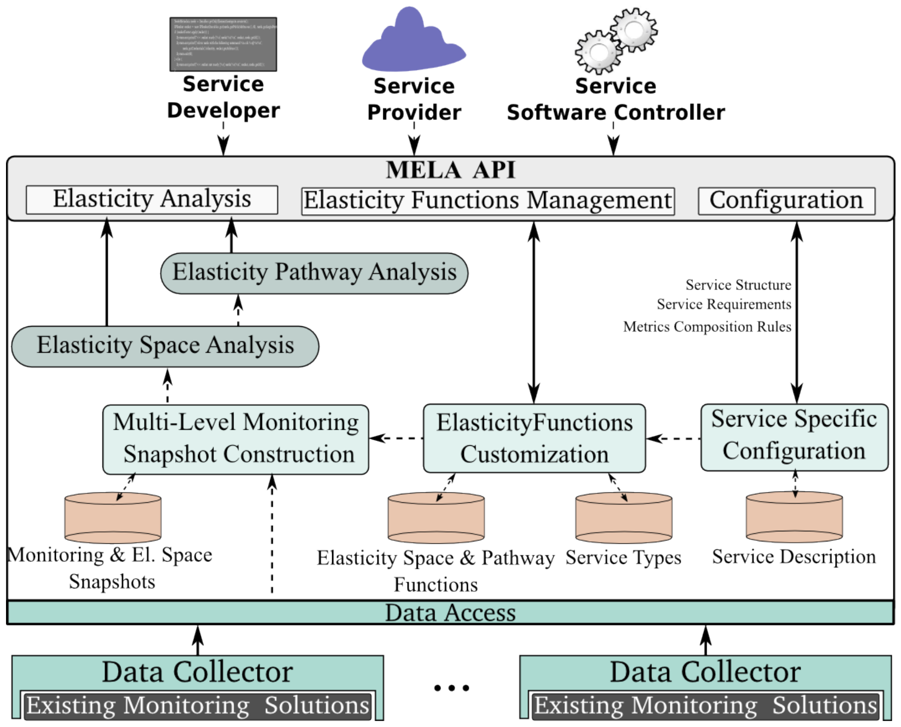

Service Developer
Service Provider
Service Software Controller
MELA API
Elasticity Analysis
Elasticity Functions Management
Configuration
Elasticity Pathway Analysis
Service Structure
Service Requirements
Metrics Composition Rules
Elasticity Space Analysis
Multi-Level Monitoring Snapshot Construction
ElasticityFunctions Customization
Service Specific Configuration
Monitoring & El. Space Snapshots
Elasticity Space & Pathway Functions
Service Types
Service Description
Data Access
Data Collector
Existing Monitoring Solutions
...
Data Collector
Existing Monitoring Solutions

Figure 6.2: MELA extended with services for analyzing elasticity space and pathway

& Elasticity Space Snapshots repository. Using the *Elasticity Analysis API*, the MELA user can retrieve the elasticity space and pathway for the whole system, or specific monitored elements.

6.5.2 Analyzing elastic cloud systems using Elasticity Space and Pathway

The elasticity space is used to determine what are the behavioral boundaries in which the system fulfills supplied requirements. The elasticity pathway function prototype

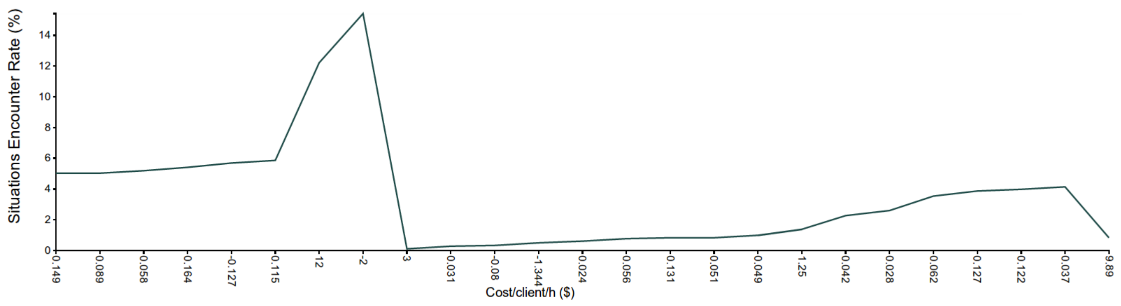


Figure 6.3: Cloud system level elasticity pathway

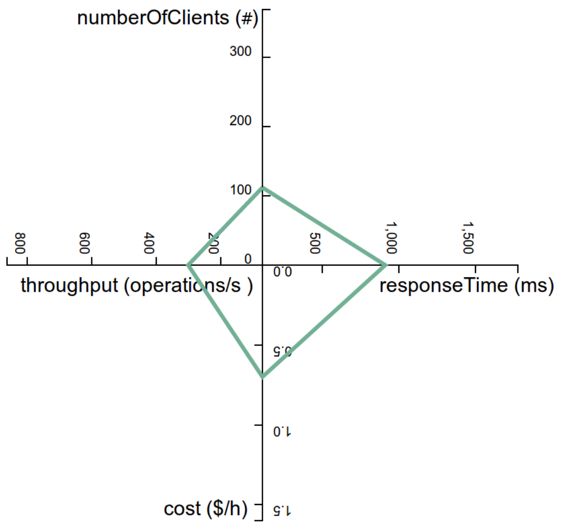


Figure 6.4: Event Processing topology elasticity space snapshot

groups combination of different metric values as DOMINANT, NON-DOMINANT, and RARE, according to their encounter rate in the monitoring data. Thus, it determines the usual behavior of the system and the correlations between the analyzed metrics. In our scenario, assuming a month of 30 days, the elasticity requirement for the system is a maximum cost of 0.0034$ per served client per hour. Using MELA, the user determines what are the elasticity boundaries for all the system topologies and units in which the cost requirement is respected. The user also determines using MELA the correlations between the analyzed metrics. Using this information, the MELA user can validate and refine the system's elasticity requirements used to control the elasticity of the system.

As the `cost` boundaries are set by the MELA user, the elasticity pathway of the system is inspected (Figure 6.3), revealing that the `cost/client/h` is less than 0.0034$ only in approximate 72% of the encountered situations. This leads to the conclusion that a pricing scheme of 2.5$ per month per client is not fully sustainable. To find the reason for this situation, the provider uses the MELA multi-level analysis feature to focus on the Event Processing Topology.

Figure 6.4 presents a snapshot of the elasticity space for the Event Processing Topology containing the `numberOfClients/h`, `responseTime`, `throughput`, and `cost/h` metrics. The complete space is depicted in Figure 6.5 and thick lines mark the elasticity space boundaries, i.e. minimum and maximum acceptable values. For the `numberOfClients/h` elasticity space dimension (upper left corner), the determined lower elasticity space boundary is approximately 150, understandable given that the system uses at least 4 VMs at 0.12$/h (150 multiplied by 0.0034, gives a 0.51$/h). To learn more about the system topology behavior and its boundaries, the user focuses on the `responseTime` dimension (lower right corner), for which MELA determines both a minimum and maximum boundary. It might seem counter-intuitive to also determine a minimum boundary, as the user would expect to always want minimum response time.

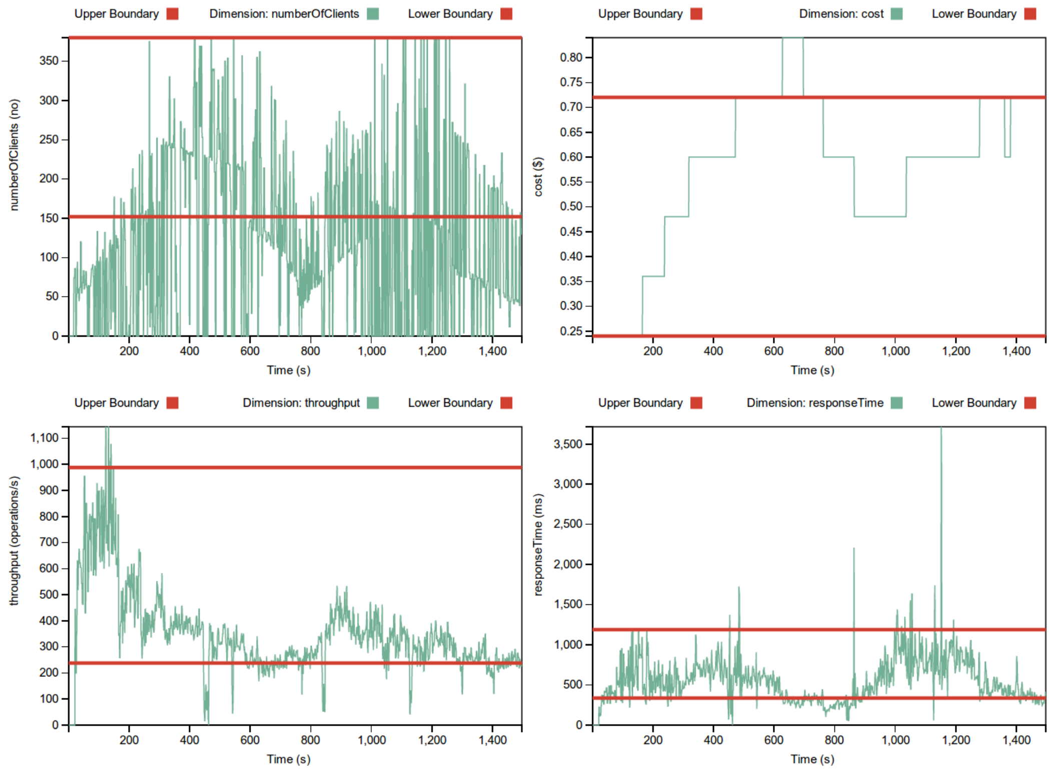


Figure 6.5: Event Processing topology elasticity space

However, in this case, this might be as result of underusing the virtual machines, which is not cost effective. Similarly, a lower boundary was determined for the `throughout` dimension (lower left corner). The points of low throughput are consistent with those of low `responseTime`, thus strengthening the previous conclusion. By investigating the elasticity pathway of the Event Processing Topology (Figure 6.6), and summing up the behavior situations, the user can determine that in approximately 20-30% of the situations, there is high response time with low number of clients. This might indicate a potential bottleneck in the system, and a system developer can use this information to improve the system's behavior.

The MELA user focuses next on the system units belonging to the *DataEnd* topology. The user examines the elasticity space of the *DataController* system unit by capturing its `writeLatency` and `cpuUsage` (Figure 6.7). For the `writeLatency` elasticity space dimension (left side), MELA determined a higher elasticity space boundary, indicating the maximum latency recorded in which the system respected the cost requirement. For the `cpuUsage` dimension there is only a lower elasticity boundary. Investigating the elasticity pathway of the *DataController* system unit (Figure 6.8), the MELA user can determine that in approximately 30% of the encountered situations the `cpuUsage` was over 95%, and in around 50% over 90%, indicating a potential bottleneck. However, a further analysis is needed to understand the complete elasticity behavior of the DaaS

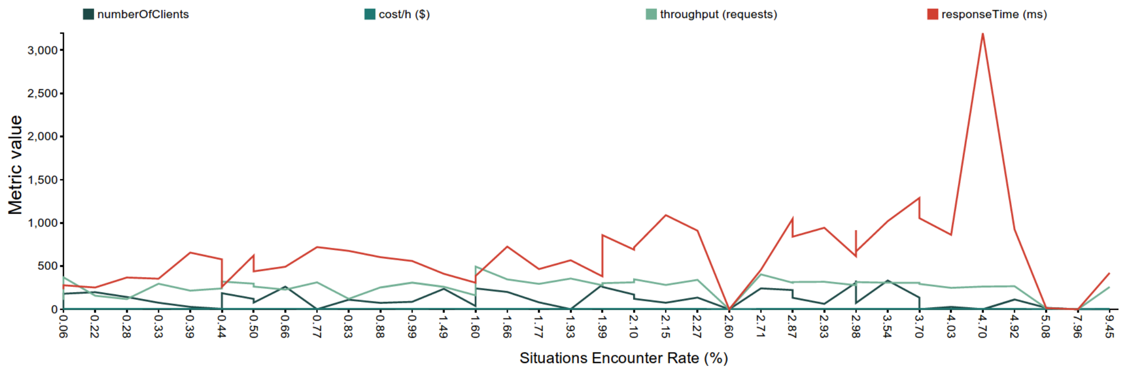


Figure 6.6: Event Processing topology elasticity pathway

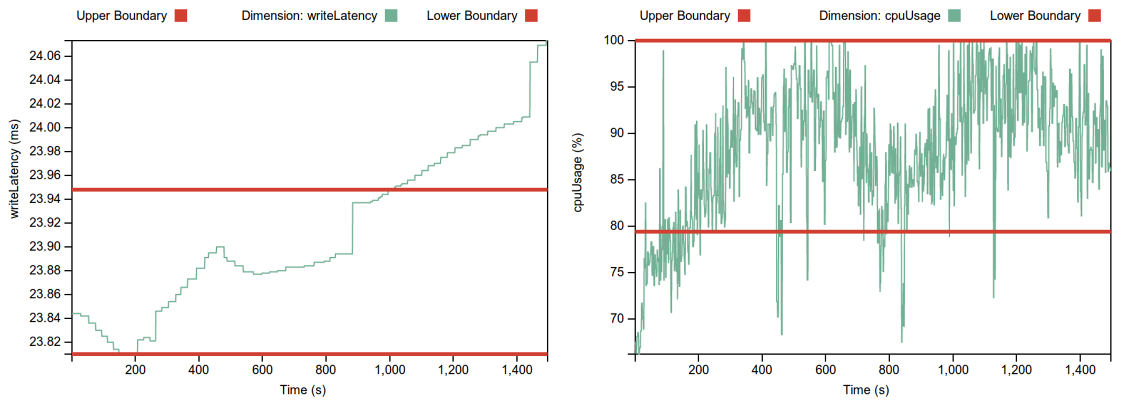


Figure 6.7: Data Controller unit elasticity space

system, by investigating the elasticity space and pathway for the other system elements, selecting more metrics to be analyzed, and/or refining the elasticity requirements for the elasticity controller.

The above-mentioned experiments have highlighted the importance and challenges of monitoring and analyzing the behavior of elastic systems at multiple levels. The elasticity space is crucial in understanding the elasticity boundaries of cloud systems. In these experiments, the user can obtain insights regarding what other boundaries the control mechanism must enforce at different levels in the cloud system. E.g., given the control mechanism and workload used, the MELA user learned that that it needs at least 148 clients to fulfill the overall cost requirement. Applying the elasticity pathway function, the user can learn what is the overall behavior of the cloud system. This brings insight in how does the system behave and what are the dependencies between the system's metrics, e.g., `responseTime`, `throughput` and `numberOfUsers`. With this insight, the user can refine the system's requirements and resume the analysis process, iteratively improving the system's elasticity.

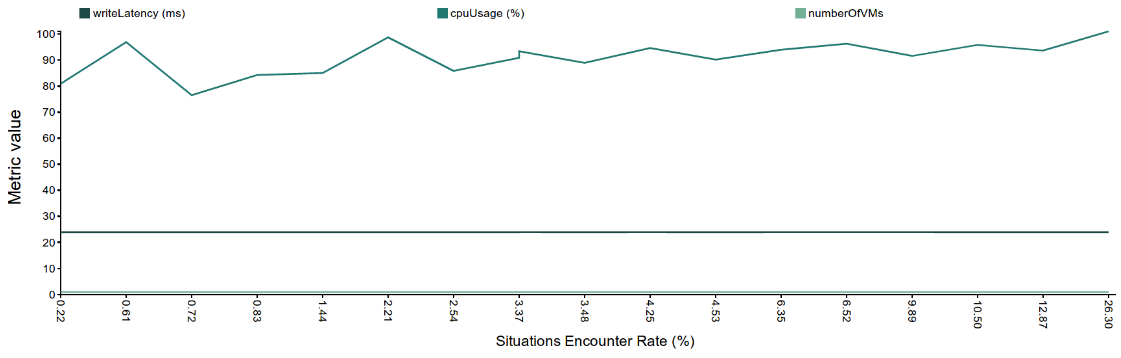


Figure 6.8: Data Controller unit elasticity pathway

6.6 Conclusions

Elasticity analysis is crucial for cloud system developers and owners in understanding the behavior of their systems at multiple levels, from individual units to the whole system. This information can be further used towards developing smarter mechanisms for controlling their elasticity. This chapter introduced concepts and techniques for monitoring and analyzing the elasticity of cloud systems. Our approach determines elasticity boundaries for all system's elements, evaluating the elasticity space based on the cross-level metric composition mechanism and user requirements. This brings insight in the behavior of the system, providing information on how such a system should be controlled. Applying the elasticity pathway over the elasticity space, a mechanism for classifying the elasticity behavior of cloud systems was defined, providing insight in the system's behavior evolution and acting as a base for predicting it.

CHAPTER 7

Analyzing Elasticity Relationships in Elastic Systems

In this chapter we focus on determining the elasticity relationships influencing the behavior of elastic systems, based on their elasticity space and monitoring information. Such relationships can be present among system units and topologies, and understanding them can improve the system's elasticity control.

7.1 Introduction

Due to the increasing number of available technologies for developing cloud systems, from hypervisors and virtual containers to platforms, cloud system are becoming more and more complex. System developers are able to run system units on top of virtual containers (E.g., Docker[40]), distributed among virtual machines in different clouds. However, in such cloud systems, individual system units are typically not behaving independently. Instead, due to communication dependencies (E.g., unit A sends/retrieves data from unit B) or run-time control dependencies (E.g., data end re-balancing after scaling), there exist different relationships between system units, influencing their run-time behavior.

We will refer to such relationships, which affect the run-time elasticity of the system, as *elasticity relationships*. Particular relationships can be of interest for particular stakeholders, including system owners, developers, and elasticity controllers. For example, a relationship between performance and resource usage could be used by a developer to estimate the maximum achievable performance before the resource becomes a bottleneck. Another relationship between cost and performance could indicate how much a system

[39]The contributions from this chapter where originally presented in [34].

[39]In the original content ([34]) we refer to cloud systems as cloud services. For consistency, in this chapter we maintain the terminology used up to this point in the thesis.

[40]`https://www.docker.com/`

owner is expected to pay for certain performance. We have seen that existing tools for analyzing elasticity of cloud systems focus on individual performance metrics [75]. However, relying only on information provided by these tools, system developers are unable to discover hidden design issues with future system elasticity, which can be captured by relationships between apparently unrelated system units. Moreover, current elasticity controllers can evaluate only the impact of their decisions on individual units, and are unable to understand how enforcing one elasticity capability on one unit affects the other units in the system. Thus we analyze and understand if relationships exist between system units, towards assisting the development and refinement of elastic cloud systems and controllers.

However, analyzing such relationships is challenging. First, due to the potential complexity of the system's software stack, each software layer can introduce different relationships. Second, due to possible multi-cloud system deployments, the relationships can vary between different cloud providers. Thus, there is a need to investigate new concepts and techniques for determining and analyzing elasticity relationships in complex multi-cloud elastic systems.

To this end, we focus on determining relationships between any of the system's performance, cost, and resource usage. For this, we characterize *elasticity relationships* of elastic cloud systems, and introduce algorithms to determine them.

In this chapter we make the following contributions:

- we formally define elasticity relationships of cloud systems
- we introduce a mechanism and algorithms for analyzing elasticity relationships based on system monitoring information
- we implement our approach as a platform for run-time analysis of elasticity relationships of cloud systems

7.2 Roadmap

The rest of this chapter is structured as follows. Section 7.3 presents the motivation and approach behind analyzing elasticity relationships of cloud systems. Section 7.4 introduces the concept of *elasticity relationships* and our approach for analyzing them. Section 7.5 presents our prototype and evaluation. Section 7.6 concludes the chapter.

7.3 Motivation

Let us consider the Data-as-a-Service (DaaS) elastic cloud system from Chapter 3. At run-time, an elasticity controller scales the service using these capabilities, according to elasticity requirements defined over various monitored metrics, e.g., `response time` $\leq$ `100 ms` for *Event Processing* unit, and `cpu usage` $\leq$ `90 %` for *Data Node* units. The DaaS provides data storage and exchange services for Machine-to-Machine (M2M) gateways, such as smart cities or vehicle fleets. Due to data privacy concerns, the DaaS

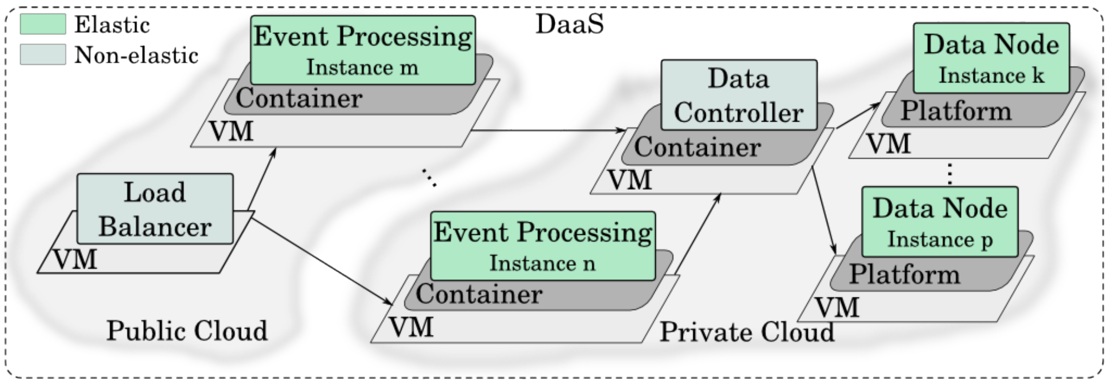


Figure 7.1: Elastic multi-cloud data-as-a-service (DaaS)

data end units are hosted in a private cloud, while event processing instances can run both in private and public cloud providers.

To be able to leverage the DaaS's elasticity capabilities for achieving user-defined requirements, we must understand how the behavior of one system unit is affected by the other units. Thus, we need to determine the elasticity relationships between individual system units and topologies, to understand how enforcing an elasticity capability on one unit/topology affects the elasticity of the other units and topologies.

For example, if the event processing end of our system is too loaded, a controller will scale it out. After the scaling, the data end might not be able to handle the increased number of requests coming from event processing instances. Thus, the data end could become in turn a performance bottleneck. Knowing this relationship beforehand would enable an elasticity controller to preemptively scale the data end, obtaining better system performance.

From the DaaS's structure (Figure 7.1) we can assume that elasticity relationships should exist between units which communicate directly, such as *Load Balancer* and *Event Processing*. However, other relationships might not be so obvious, being generated by indirect communication. For example, a relationship could exist between *Load Balancer* and *Data Node*, such as the load on the *Load Balancer* influencing the performance of the *Data Node*. Depending on the system, the relationship's interpretation can also differ, a relationship between metrics belonging to the same individual unit being potentially less important than relationships determined between different units.

Elasticity relationships are determined by analyzing the system's monitoring information. For exemplifying potential elasticity relationships we consider the DaaS's monitoring information structured using our approach in Chapter 5 and depicted in Figure 7.2. Due to communication dependencies, the DaaS can have several elasticity relationships (Table 7.1). A relationship could be present between monitored *cpuUsage* on the *Data Node* units, and *responseTime* of the *Event Processing* units, indicating if the data end is a bottleneck or not. Another relationship could exist between *throughput* on *Event Processing* unit, and *cpuUsage* on *Data Controller*, indicating what is the maximum achievable *throughput* before *cpuUsage* is too high. While the previous rela-

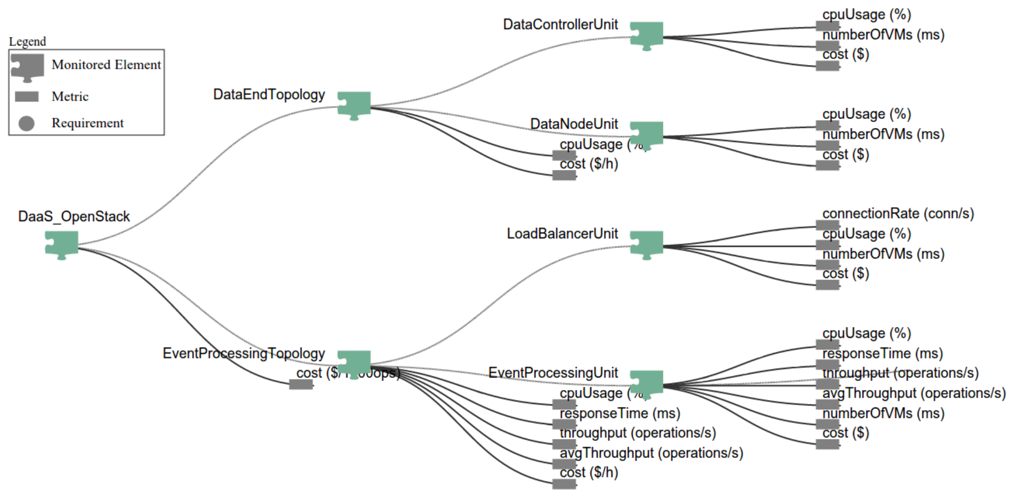


Figure 7.2: DaaS with structured monitoring information

Elasticity relationship (*element:metric* → *element:metric*)
DataNode: cpuUsage → EventProcessing: responseTime
EventProcessing: throughput → DataController: cpuUsage
LoadBalancer: connectionRate → EventProcessing: throughput
EventProcessing: throughput → DataNode: cpuUsage
DaaS:cost → DataNode: cpuUsage & EventProcessing: responseTime

Table 7.1: Potential DaaS elasticity relationships

tionships are direct, we can also have indirect relationships, such as the *connectionRate* on *Load Balancer* influencing *throughput* on *Event Processing*, which in turn influences *cpuUsage* on *Data Node* units. Finally, beside one-to-one relationships, we can also have many-to-one relationships. For example, if the previous requirements are used to scale the DaaS, the overall DaaS *cost* could depend on both requirements' metrics, *cpuUsage* on *Data Node*, and *responseTime* of the *Event Processing* units.

When discussing about elastic cloud systems it is important to consider the different stakeholders involved, and their perspectives on elasticity (7.3). Moreover, each of these stakeholders can have different *elasticity boundaries* over their perspectives, i.e. different elasticity requirements. Starting from system level perspective, a *System Administrator* would have an *Infrastructure-level* perspective, viewing elasticity as the allocation/deallocation of virtual resources (VMs, virtual networks), and would have as elasticity boundaries infrastructure-level requirements, such as *User Quotas* or *Infrastructure Limits*. A *System Developer* would have a *Platform-level* perspective, viewing elasticity as reconfiguring, allocating and deallocating system units, and would consider as elasticity boundaries system-level requirements, such as *Requests Queue Size* or *Data Processing Units Pool Size*. The *System Owner* in turn would have a *Business-level* perspective guided by *Budget Constrains*, and would consider as elasticity the possibility of altering business contracts such as *Pricing Schemes*, *Cloud Services Reservation*

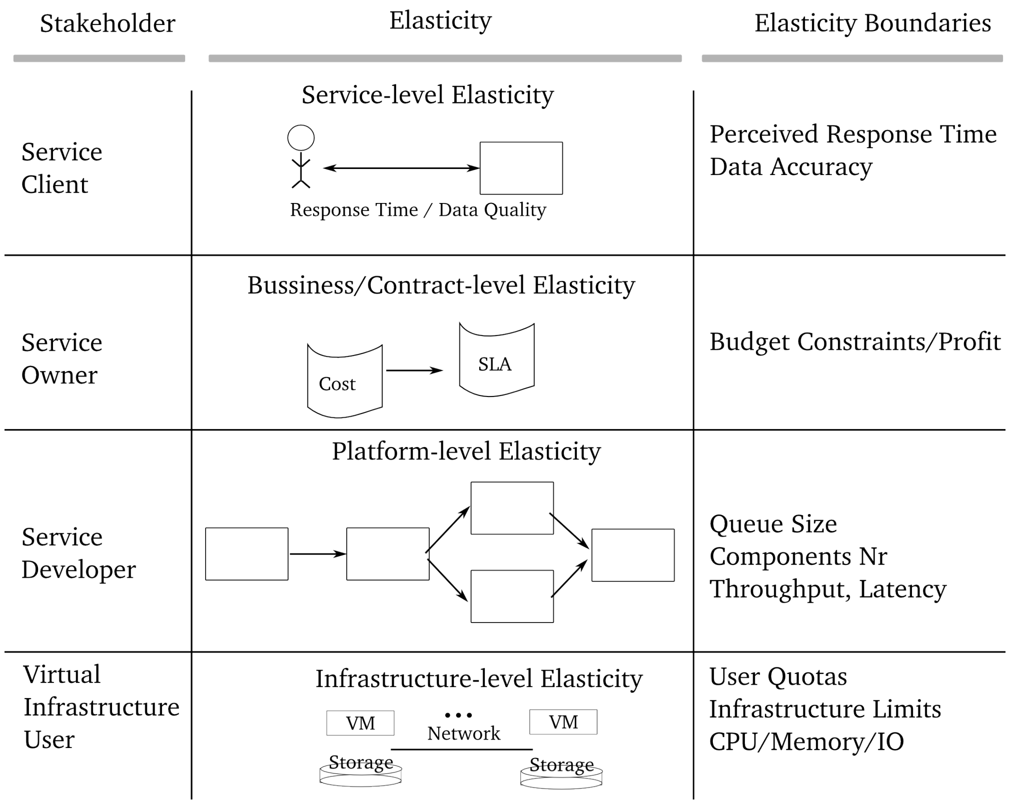


Figure 7.3: Elasticity stakeholders and perspectives

Schemes, or *Agreed SLAs*. Finally, the *System Client* would consider a system elastic if it maintains its elasticity boundaries with respect to user-perceived system performance, such as *Perceived Response Time* or *Data Quality*.

Thus, specific stakeholders could be interested in specific relationships. For example, a DaaS provider might be interested in cost relationships, to better plan their business. System developers might be interested in performance relationships, which they can use to adjust the system to eliminate bottlenecks or reduce resource underutilization. Various parameters of elasticity relationships can further be interpreted by software controllers, ensuring better automated control decisions. For example, an elasticity controller would benefit from understanding that after event processing scale-out, the DaaS's data end might not be able to handle the increasing number of requests, and thus, will become in turn a performance bottleneck.

This potential relationship complexity highlights the need of automatically determining elasticity relationships, and using them to improve the elasticity of complex single and multi-cloud systems. To be able to leverage the elasticity capabilities of elastic cloud systems in controlling the system's elasticity, we must understand how the elasticity of one system unit is affected by the elasticity of the other system units. It is crucial to understand how the system elasticity at one perspective influences the elasticity of the other perspectives, as to ensure the elasticity boundaries imposed by all stakeholders. To this end, we need to determine the elasticity relationships between individual system units and topologies, to understand how enforcing an elasticity capability on one topology affects the elasticity of the other units and topologies.

Current tools such as JCatascopia [68] or vPerfGuard [76] can show metrics related to performance, cost, or resource usage of individual system units, or give indicators about the future evolution of such metrics [77]. However, they do now answer the following questions crucial for developing elasticity controllers:

- what metrics are involved in elasticity relationships
- what are the functions describing the relationships
- how are the relationships affected by different clouds

To this end, we develop a mechanism for analyzing elasticity relationships of cloud systems based on the system's monitored behavior abstracted w.r.t its elasticity requirements.

7.4 Analyzing elasticity relationships of cloud systems

In this section we introduce our approach for analyzing elasticity relationships of cloud systems, towards supporting the development and control of elasticity controllers which better fulfill user-specified elasticity requirements. Before analyzing such relationships, we must first understand what an elasticity relationship is. For clarity, the concepts introduces in this section will be accompanied by examples targeting the DaaS presented in Chapter 3. We assume a set of elasticity requirements stating a `response time` $\leq$ `100 ms` requirement for the Event Processing, and a `CPU usage` $\leq$ `90 %` requirement for the Data Node instances.

7.4.1 Classifying elasticity relationships

Depending on the system's software stack and cloud deployment, various elasticity relationships can exist at different software layers between system units. Thus, we must be able to analyze multiple system types, from simple single-cloud systems, to complex systems running multiple units in virtual containers distributed among virtual machines hosted in different clouds. To this end, we use as input the model for representing elastic cloud systems introduced by Copil et al. [5]. The model describes a cloud system as composed of system units (i.e. functional blocks) logically grouped in system topologies (Figure 7.4). Any of the units, topologies, or whole system is considered an *Elastic Element*, and each element can have elasticity metrics, requirements, and capabilities.

As we aim to determine elasticity relationships, we expect that "elasticity" means different things for different cloud systems and units, according to specific requirements. To this end, we denote with *Elasticity Metric* any monitored system metric which can be used to determine if the system is elastic or not, i.e., has associated elasticity requirements.

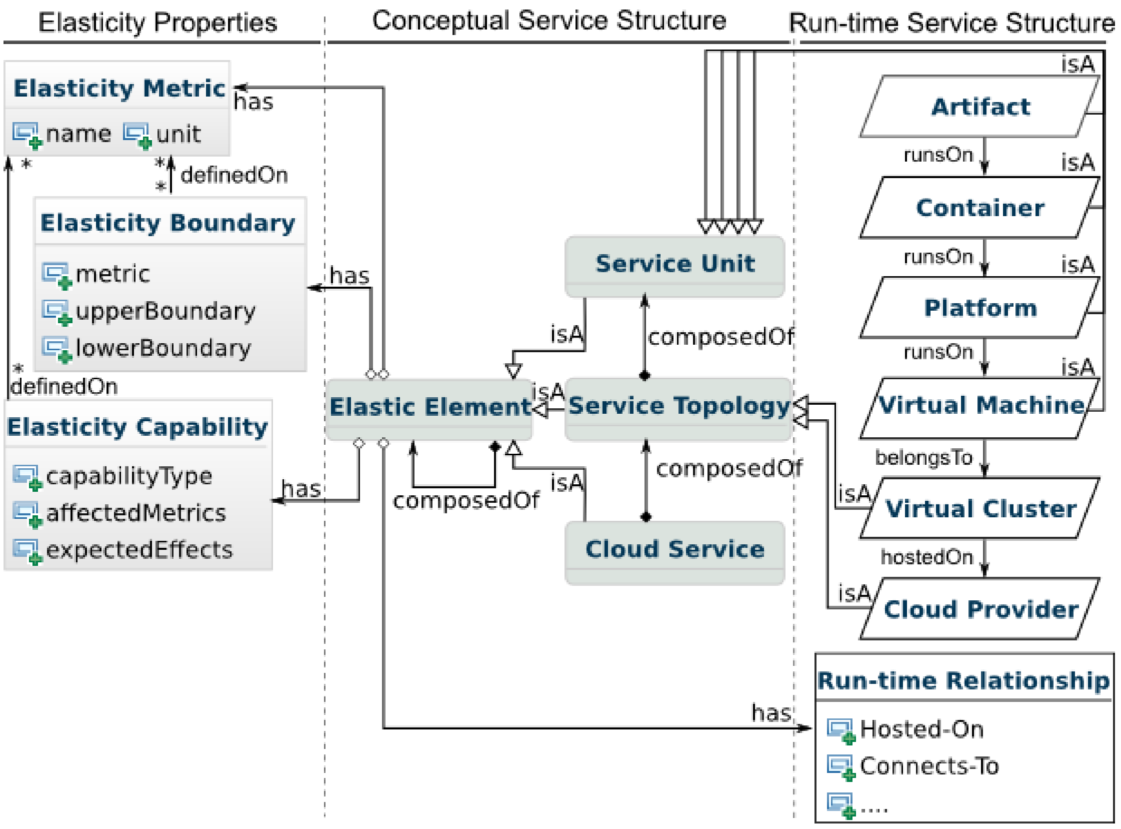


Figure 7.4: Elastic cloud system

Following the multi-dimensional principle of elasticity introduced by Dustdar et al. [15], an elasticity metric belongs to one of the following elasticity dimensions: *Cost*, *Quality*, or *Resources*. Different systems and their units could have different elasticity metrics, such as response time for an elastic web system, or data access latency for a data repository.

Elastic systems have elasticity requirements associated to elasticity metrics, describing their desired behavior. Thus, in determining elasticity relationships, we consider such requirements. We further use the concept of *Elasticity Boundary*, introduced in Chapter 6, for representing requirements that bound the values of one or more elasticity metrics. An *Elasticity Boundary* has the form $ElBoundary(m) = \langle m^u, m^l)\rangle$, where m^u and m^l denote the upper and lower bound over the allowed values of metric m.

Usually, elasticity of cloud systems is driven by quality, cost, resource usage, or a combination of the three. Moreover, system owners usually view their systems from a perspective driven by cost, quality, or both. As different elasticity relationships can exist between different perspectives, we classify relationships after the two fundamental business dimensions, *Cost* and *Quality*, and the three elasticity dimensions, *Quality*, *Cost*, and *Resources* (Table 7.2). The relationship category is given by the type of monitoring information used to determine it, different categories being potentially of interest to

Category	Relationship	Interested stakeholder	Usefulness
Quality dependency	*Quality*→*Quality*	Developer, Controller	Indicates potential quality/performance bottlenecks
Benefit-Cost	*Quality*→*Cost*	Owner, Developer, Controller	Indicates potential resource bottlenecks
Resource quality	*Quality*→*Resource*	Owner, Developer	Describes expected quality/performance when using certain resources
Cost effectiveness	*Cost*→*Quality*	Owner, Developer	Describes expected quality/performance under certain cost scheme
Cost composition	*Cost*→*Cost*	Owner, Developer	Describes the cost elements contributing to overall system's cost, indicating potential cost hot spots
Cost utility	*Cost*→*Resource*	Owner, Developer	Indicates potential resource bottlenecks under certain cost schemes

Table 7.2: Elasticity relationships

different stakeholders. System developers and elasticity controllers might be interested in *Quality dependency* or *Resource quality* relationships, which they can use to eliminate bottlenecks or reduce resource underutilization. A DaaS owner might be interested in cost-related relationships, such as *Cost effectiveness*, *Benefit-Cost dependency*, or *Cost composition.* Various parameters of elasticity relationships can further be interpreted by intelligent software controllers, ensuring better control decisions.

7.4.2 Elasticity relationship

Elasticity of cloud systems is driven by elasticity requirements, which specify boundaries over the system's metrics. To fulfill these requirements, elastic systems change their structure and used virtual resources at run-time through reconfiguration actions. Due to this reconfiguration, we should not determine relationships based on absolute monitored values, as such relationships might not hold after a reconfiguration. Instead, we determine relationships based on the system's behavior with respect to its elasticity boundaries, which, by abstracting from the absolute monitored values, can be used to describe the system behavior under different configurations.

Thus, based on the previous model and the elasticity boundary concept, we define an elasticity relationship, as follows:

Definition 4 *An* Elasticity relationship *between one elastic element and a set of other elements describes the change in the behavior of the first element w.r.t. its elasticity boundaries, triggered by a change in the behavior of the other elements.*

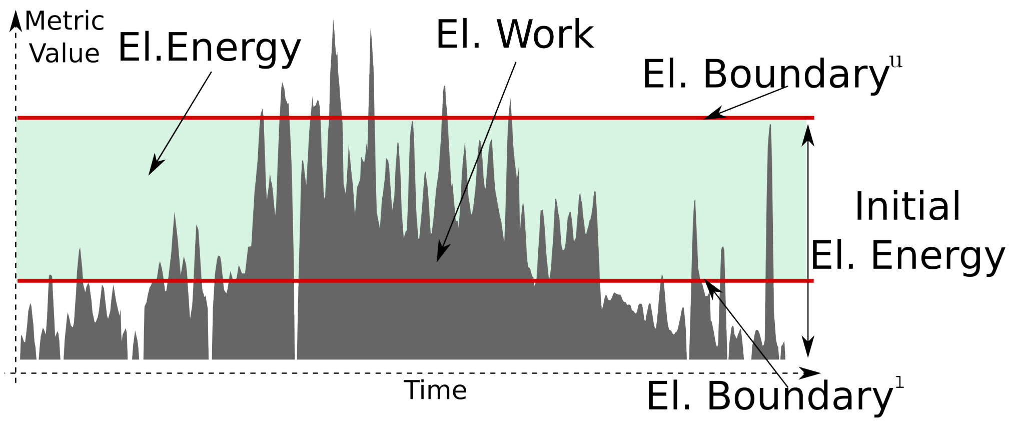


Figure 7.5: Elasticity Boundary, Work and Energy concepts

The most important for a relationship is determining the change function. The function describes how much the value of the elasticity metrics of one element change. The change of interest is w.r.t., metric's boundaries, when the values of the metrics monitored on other elements change. According to the internal processes of each element, the change function might be might be observed at run-time with a certain delay, and could attenuate over time.

Considering these issues, we capture an elasticity relationship *ElRelationship* between one or more elastic elements from a set, *ElasticElements*, as a tuple of functions: $ChangeFct$, $DelayFct$ and $AttenuationFct$ as follows:

$$ElRelationship : ElasticElements \rightarrow (ChangeFct, DelayFct, AttenuationFct) \quad (7.1)$$

where $ChangeFct$ is the function describing the change in the metrics of the related elements as a result of the relationship; $DelayFct$ is the delay with which the $ChangeFct$ is observed at run-time; $AttenuationFct$ the attenuation function which diminishes the effect of $ChangeFct$ over time.

We characterize the change function, $ChangeFct$, as taking for input a set of elastic elements *ElasticElements*, and having as output the estimated values for the elasticity behavior *Elasticity* of elastic element e:

$$ChangeFct_e \; : \; ElasticElements \times ... \times ElasticElements \; \rightarrow \; Elasticity(e) \quad (7.2)$$

Relying on the change function, users can estimate the behavior of each element. Quality, cost problems, or resource bottlenecks could be predicted using the relationships. Then, actions could be taken to address the predicted problems, and improve the overall elasticity of the system.

7.4.3 Elasticity relationships analysis

Elasticity of cloud systems is evaluated based on boundaries over the system's metrics defined by elasticity requirements. For determining the $ChangeFct$ (Eq. 7.2), we need to abstract, from concrete monitored values, the behavior of elastic systems with respect to their requirements (i.e., elasticity boundaries). To this end we define the concept of *Elasticity Work* of a cloud system as the current load on the system with respect to its elasticity boundaries. To the difference between the current elasticity work and the elasticity boundaries, we define the *Elasticity Energy* as the difference between the current and maximum acceptable load (upper boundary). Using these concepts illustrated for a single metric in Figure 7.5, we can determine relationships between the elasticity energy of two systems, and not individual metric values

First, for determining the $ChangeFct$ (Eq. 7.2), we quantify the absolute distance between the upper and lower elasticity boundaries for each elasticity metric of an elastic element using the *Initial Elasticity Energy* ($IElEnergy$):

$$\begin{aligned} IElEnergy(e) = \{ &\|ElBoundary(m)^u - ElBoundary(m)^l\| \\ &\mid m \in elasticity\ metrics \in \ \{Cost, Quality, Resource\}\} \end{aligned} \tag{7.3}$$

where $ElBoundary(m)^u$ and $ElBoundary(m)^l$ denote the upper and lower bound of elasticity metric m belonging to any of elasticity dimensions $Cost$, $Quality$, $Resource$.

Using the $IElEnergy$ we quantify the load monitored on the elasticity metrics of an elastic element w.r.t. its initial elasticity energy using the *Load Unit*, defined as a unit of usage over the energy of a metric in a time frame. We capture the load on the elasticity metrics of an element using the *Elasticity Work*, $ElWork$, as the percentage of energy used relative to the initial energy of the element over its metrics:

$$ElWork(m) = \frac{x * LoadUnit(m)}{IElEnergy(m)} \tag{7.4}$$

where x is the number of load units used per 1 time unit over which the load is measured, from the initial energy $IElEnergy$ of metric m.

$ElWork$ is a result of system's load, or resource usage while idle, based on which we can compute the instant elasticity energy, `ElEnergy`, of an element `e`. `ElEnergy` is computed as the difference between the element's initial energy, normalized to 100, and the sum of the work done in idle ($ElWork_{idle}$), and in load ($ElWork_{load}$), as follows :

$$ElEnergy(e) = 100 - ElWork_{idle}(e) - ElWork_{load}(e) \tag{7.5}$$

$ElEnergy$ is used to describe the behavior of the system, a zero energy indicating it violates its requirements, while one close to the initial energy indicates that it is underused. Using $ElEnergy$ for representing the co-domain of the $ChangeFct$ from Eq. 7.2 ($Elasticity$), from an elasticity relationship between one element e_i and a set of other elements $e_k, ..., e_n$, we can compute the expected values of e_i's elasticity energy at time t. To this end we apply the relationship's $ChangeFct$ over the elasticity energy of metrics belonging to each related element, as $ChangeFct^t_{e_i}(ElEnergy(e_k)^t, ..., ElEnergy(e_n)^t)$, considering the $DelayFct$, and $AttenuationFct$ functions. Estimating the energy values is useful for both system developers and controllers in estimating the system's behavior.

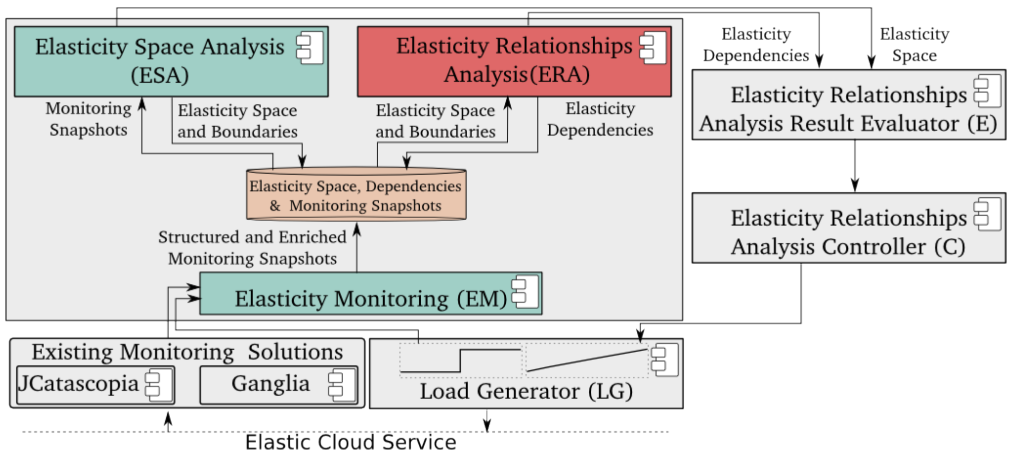
We develop Algorithm 6 for determining elasticity relationships. Depending on the type of relationships we want to determine, we must be able to investigate from only a subset of metrics, to all collected metrics for all system's elements. To this end, our algorithm can be applied for determining for any metric of interest, the elasticity relationships it has with another set of monitored metrics. For each analyzed elastic metric, the *ComputeElEnergy* function (Lines 11-18) applies Eq. 7.5 to compute the elasticity energy over each metric monitored value, considering the monitored value as indicator of complete elasticity work, *elWork*. By applying *ComputeElEnergy* over all analyzed metrics (Lines 2-6), we obtain for each metric a time series of elasticity energy values. The elasticity energy is determined based on the initial elasticity energy, which can change at run-time due to scaling actions, and energy work, which changes according to the system load. The energy time series provides us with information about the elasticity behavior of the analyzed elements over each analyzed element's metric in time. Over the elasticity time series, various analysis techniques can be applied (Line 8), depending on the type of analyzed relationship.

Algorithm 6 Determining elasticity relationships between system metrics

Input: *m* - elastic metric to discover relationships for
Input: *metrics* - elasticity metrics potentially related to *m*
Output: *relationship*

```
function AnalyzeELRelationships(m, metrics)
    mElEnergy = ComputeElEnergy(m)
    metricsElEnergy
    for $m_i$ in metrics do
        metricsElEnergy.add(ComputeElEnergy($m_i$))
    end for
return TimeSeriesAnalysis(mElEnergy, metricsElEnergy)
end function

function ComputeElEnergy(metric)
    elEnergyInTime = []
    iELEnergy = $|| \, m.Boundary^u - m.Boundary^l ||$
    for elWork in metric.monitoredValues do
        elEnergyInTime.add(100 - $\frac{elWork}{iElEnergy}$)
    end for
return elEnergyInTime
end function
```



Elasticity Space Analysis (ESA)
Elasticity Relationships Analysis(ERA)
Elasticity Dependencies
Elasticity Space
Elasticity Relationships Analysis Result Evaluator (E)
Monitoring Snapshots
Elasticity Space and Boundaries
Elasticity Space and Boundaries
Elasticity Dependencies
Elasticity Space, Dependencies & Monitoring Snapshots
Elasticity Relationships Analysis Controller (C)
Structured and Enriched Monitoring Snapshots
Elasticity Monitoring (EM)
Existing Monitoring Solutions
JCatascopia
Ganglia
Load Generator (LG)
Elastic Cloud Service

Figure 7.6: MELA extended with elasticity relationships analysis service

7.5 Evaluation

7.5.1 Prototype

Architecture

For applying our approach from Section 7.4, we extend MELA, a platform for monitoring and analyzing elastic services, introduced in Chapter 5.4.4. We add a new *Elasticity Relationship Analysis* service[41] implementing our techniques for analyzing elasticity relationships (Figure 7.6). MELA already provides an *Elasticity Monitoring* system which collects monitoring data, structures and enriches it, and an *Elasticity Space Analysis* service which uses this data to determine the system's elasticity space and boundaries.

While for the determined relationships we require elasticity boundaries over all cloud system's metrics for computing the system's elasticity energy, they might not be always known. Thus, we use MELA's *Elasticity Space Analysis* service for determining the *Elasticity Space* of the target system from supplied elasticity requirements and collected monitoring information. The elasticity space contains the *Elasticity Boundaries* determined for all elasticity metrics. Based on the determined boundaries and monitoring information, our *Elasticity Relationships Analysis* service uses an array of functions and techniques to determine the system's elasticity relationships. The elasticity relationships' analysis result is evaluated by a *Result Evaluator*, and an *Elasticity Relationships Analysis Controller* orchestrates all components.

[41]Prototype and supplement materials at `http://tuwiendsg.github.io/MELA/elasticityRelationshipsService.html`

Functions for determining elasticity relationship

For determining the elasticity relationships' coefficient functions, i.e., energy change, delay, and attenuation, we apply R[42] functions. To use R, we compute from monitoring information the elasticity energy of each metric at each monitoring interval. Thus we obtain for each elasticity metric an elasticity energy time series over which we apply R analysis functions. To obtain a clear view over the usual behavior of the system, we apply a preprocessing step over the time series and remove outliers determined by R *mbox* function. We further determine the delay function $DelayFct$ of an elasticity relationship by computing the *lag* between the evaluated energy time series using the cross-covariance estimation function ccf.

The change function of each relationship is determined using a linear regression approach, computing the linear correlations between two energy time series with the linear models fitting function *lm* available in R. The change function is extracted under the form $ChangeFunction(m_{dependent}) = constant + coeff_i * m_i + ... + coeff_n * m_n$, where $m_{dependent}$ is the metric from the relationship whose values can be computed from the values of the other metrics in the relationship, by adding to the $constant$, the values of metrics m_x multiplied by their corresponding coefficients, $coeff_x$.

For each determined coefficient of the linear relationship, we check if the estimation error is one order of magnitude smaller than the coefficient, and if not, we discard the relationship as inaccurate. Finally, we obtain the change function, with the associated *Adjusted r* coefficient of determination, an indicator on how well the extracted relationship fits the original data, from 0% (no fitting), to 100% (maximum fitting). As linear model fitting is used to estimate the values of the $m_{dependent}$ metric based on the related metrics, we evaluate the quality of the estimation. We compute the standard, average, maximum, and minimum absolute variance based on the absolute difference between the metric's estimated values based on the relationship, and the monitored values. The deviation is reported in terms of concrete values, not percentages, to make it easier to evaluate it with respect to monitored values.

7.5.2 Experiments

Setup

To evaluate the proposed approach, we deploy the DaaS in both single and multi-cloud configurations, on our private OpenStack[43] cloud, and Flexiant[44], a public commercial cloud, using virtual machines of similar types (1 CPU with 1 GB of RAM). The DaaS is structured in two logical topologies, (i) *Event Processing* topology, containing instances of *Event Processing* units and a *Load Balancer*, and (ii) a *Data End* topology containing instances of *Data Node* units and a *Data Controller* acting as data load balancer. We implemented a software controllable *Load Generator* for applying stepwise increasing/de-

[42]http://www.r-project.org/
[43]http://www.openstack.org/
[44]http://www.flexiant.com/

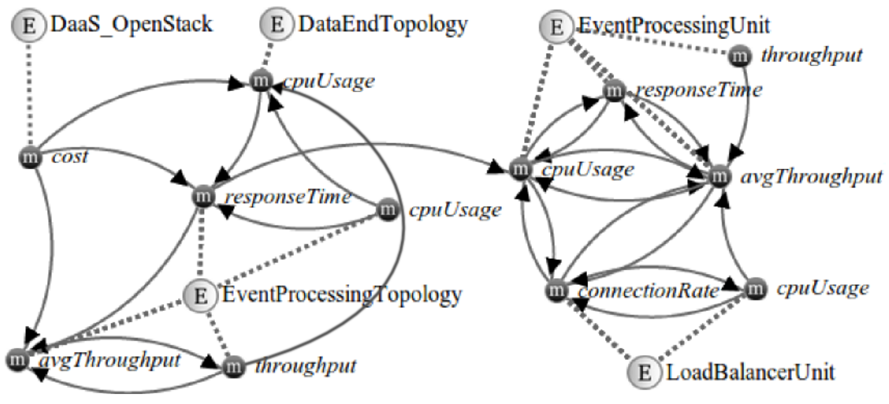


Figure 7.7: DaaS on private cloud - Determined elasticity relationships graph

creasing load over the DaaS, simulating sensors which connect and send data to the DaaS.

DaaS deployed on private cloud

First, monitoring information is structured using MELA (Figure 7.2), the metrics considered important being propagated and associated to each unit and topology. In this case *throughput*, *averageThroughput*, and *responseTime* for *Event Processing* units, and *cpuUsage* for all units are computed. The metrics are obtained applying an *average* or *sum* operation on the values monitored for each unit instance running in a virtual machine. Then, the metrics are further propagated and associated to the each unit's system topology. From the *Load Balancer*, *connectionRate* is also collected. Cost per system unit is computed by multiplying the assumed virtual machine cost with the number of machines running instances of each unit.

First, the DaaS is deployed in our private OpenStack-based cloud, with one VM for each system unit. The *Load Generator* is used to apply a workload starting with 30 sensors, increasing to 90 in steps of 30, and decreasing to 30 again, according to expected DaaS usage. Each load step takes around 5 minutes, providing enough monitoring information during the same load to enable relationship analysis. As a system developer, we want to understand if there exists any *Quality* relationship between the throughput on the *Event Processing* and the CPU usage of *Data End*, as to understand if data end CPU usage could be a bottleneck.

Using our prototype, relationships are determined as linear functions, expressed under the form $metric(t) : element = constant + coeff_i * metric_i(t) : element_i + ...$ To understand resource quality, from the determined relationships (Figure 7.7), we focus on the relationship between the *cpuUsage* of the *Data End* and *throughput* of the *Event Processing*. The determined relationship (Figure 7.8) is a linear equation in which the elasticity energy of *cpuUsage* at time t can be estimated by multiplying the *throughput*'s energy value at time t with 0.29, and adding 2.86 to the result. Converting the abstract

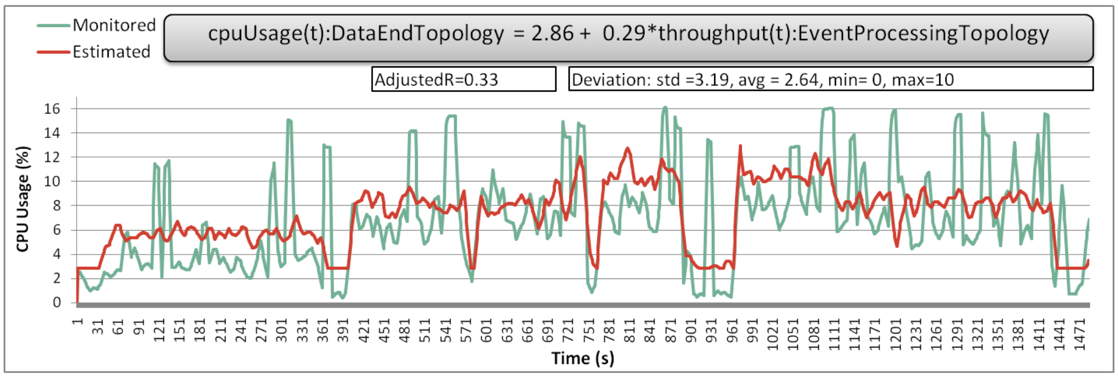


Figure 7.8: DaaS on private cloud - Determined Quality relationship

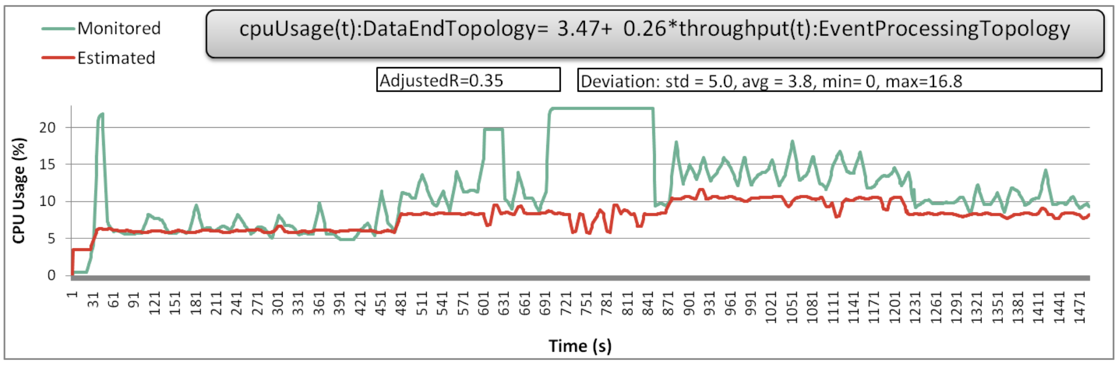


Figure 7.9: DaaS on public cloud - Determined Quality relationship

energy to concrete values with respect to the current elasticity boundaries of the system, we can estimate *cpuUsage* based on the *throughput*'s monitored values. From the relationship, we estimate that, using this deployment structure, with maximum accepted CPU usage (from elasticity requirements) of around 90%, the maximum achievable throughput is (90-2.86)/0.29 ≈ 300 sensors per second. From the relationship's quality indicators, i.e., standard deviation (std.) of 3.19, average (avg.) of 2.64 and maximum (max.) of 10, this relationship is trustworthy. Thus, when more than 300 sensors are connecting to the DaaS, the data end should be scaled out.

DaaS deployed on public cloud

We are further interested if the same relationship holds on the public cloud, and analyze the DaaS deployed on Flexiant public cloud, with same load and number of VMs. Although the DaaS behavior on the public cloud differs in terms of CPU usage pattern (Figure 7.9), the same relationship type is detected, `cpuUsage(t)=3.47+0.26*throughput(t)`, with minor differences both in its coefficients, and quality indicators, increasing the confidence that the determined relationship is not generated by particular cloud infrastructures, but instead is present in the system design, and thus, must be considered when controlling

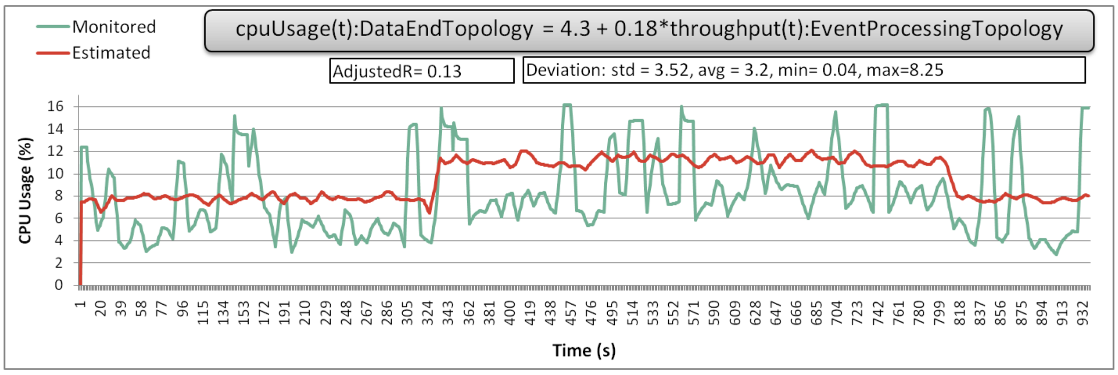


Figure 7.10: DaaS on multi cloud - Determined Quality relationship

the system's elasticity on any cloud.

DaaS deployed on multi-cloud

As both evaluations returned similar relationships, we further want to evaluate if the same relationship holds when the DaaS "bursts" into a public cloud due to elasticity requirements. To this end, we deploy the DaaS in a multi cloud configuration, with 2 *Event Processing* instances on each cloud, and the data end deployed on the private cloud. The load on the DaaS is doubled, as the system is expected to burst in public clouds only during high load periods.

From the same relationship determined for the multi-cloud scenario (Figure 7.10), we notice that the coefficient for *throughput*, `0.18`, is smaller than in the single cloud scenario, and thus, it has less influence on the overall CPU usage. This might indicate that other relationships between other metrics are influencing the DaaS, and we would need to investigate also the other determined relationships in order to understand the DaaS's behavior.

DaaS deployed on private cloud with elasticity controller

A developer might further want to determine if the previous relationships also hold during run-time elasticity control. Thus, we deploy the DaaS in the private cloud with an attached elasticity controller, rSYBL [5]. Due to requirements of *responseTime* on *Event Processing* topology $\leq$ 100 ms and *cpuUsage* on *Data End* $\leq$ 90%, at run-time, the controller adds/removes unit instances.

In this scenario, the determined relationship (Figure 7.11) also includes *responseTime*, indicating that, during elasticity control, *responseTime* has a contribution on *cpuUsage*, even if small. From the relationship, we notice that the estimated *cpuUsage* goes over 100% between certain time frames, indicating potential bottlenecks. The estimated bottlenecks are not encountered in the monitored *cpuUsage*, due to the controller scaling out the data end. From the computed quality indicators, i.e., std. deviation of 55, avg. of 40 and max. of 173.6, we notice that due to enforcing elasticity actions, the determined

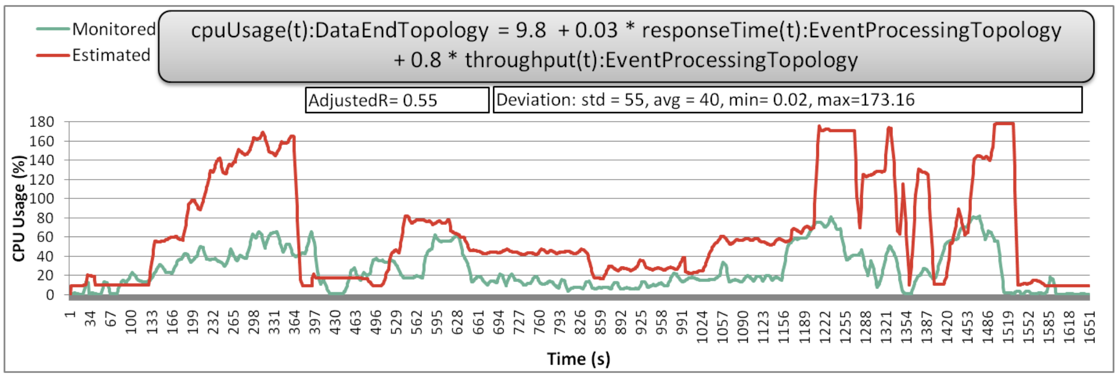


Figure 7.11: DaaS with controller - Determined Quality relationship

Determined Elasticity Relationships			Relationship quality statistics				
Category	No.	Linear relationship function	Adjusted r	Deviation			
				Std	Max	Min	Avg
Resource quality	1	*cpuUsage(t):EventProcessingTopology* = 11.7 + 0.87**cpuUsage(t):DataEndTopology*	0.55	20.2	51.5	0	17
	2	*responseTime(t):EventProcessingTopology* = 11.5 + 2.35**cpuUsage(t):DataEndTopology*	0.19	122.9	347.9	0.1	93.2
	3	*cpuUsage(t):LoadBalancerUnit* = 4.9 + 0.53**connectionRate(t):LoadBalancerUnit*	0.66	63.7	137.2	0.05	52.6
	4	*cpuUsage(t):EventProcessingUnit* = 16.9 + 0.71**connectionRate(t):LoadBalancerUnit*	0.4	68.7	187.2	0.01	54
Quality dependency	5	*throughput(t):EventProcessingUnit* = 2.5 + 0.56* *connectionRate(t):LoadBalancerUnit*	0.54	41.9	161	0.00	30.6

Table 7.3: DaaS during run-time control - Determined relationships

relationship is not trustworthy anymore, as an average error of 40% in *cpuUsage* is too high. From this evaluation we can conclude that applying elasticity control on cloud systems changes their internal elasticity relationships. Thus, one should analyze the elasticity relationships introduced by each controller, to understand if the controller introduces additional performance or quality problems, or if it removes problematic relationships such as bottlenecks.

Thus, we investigate other relationships, captured in Table 7.3. The first determined *Resource quality* relationships indicates that *cpuUsage* on the data end influences both the *cpuUsage* on the event processing topology (1), and the *responseTime* (2). This means *cpuUsage* must still be considered as a metric influencing the elasticity of the system. Relationship 3 between the *connectionRate* reported by the *Load balancer* and its *cpuUsage* can be used by the elasticity controller to decide if and when the load balancer should be scaled vertically, depending on the number of connected DaaS users. From the *Quality dependency* relationship 5, we notice that the achieved *throughput* can be estimated to 60% of the *connectionRate* monitored on the load balancer, indicating potential performance problems.

Based on the above elasticity relationships, the DaaS's elasticity could be improved by removing indicated potential bottlenecks, and its elasticity controller redesigned to enforce elasticity actions preemptively, based on estimated values.

7.6 Conclusions

In this chapter we focused on analyzing elasticity relationships in cloud systems, enabling different stakeholders to understand the behavioral relationships governing the run-time behavior of complex cloud systems. To this end, we have defined the elasticity relationships, and developed a mechanism for determining elasticity relationships of cloud systems. Our approach can be applied to a large array of system configurations, from single cloud to multi-cloud systems with complex software stacks.

We evaluated our approach on an elastic cloud system in single and multi-cloud configurations, on both private and public clouds, with and without an elasticity controller. We highlighted that different controllers and platforms can generate different relationships, which influence their system's run-time elasticity in different ways. We have shown that using our approach, a user can easily discover relationships crucial for understanding how system units and topologies influence each other at run-time.

CHAPTER 8

Cost-aware scalability of elastic systems in public clouds

In the previous chapters we have focused on selecting the appropriate cloud services, and monitoring and analyzing the behavior of elastic systems running in cloud environments. Through the experiments done in the previous chapters we have noticed that cost of elastic systems running in public clouds can be very complex. Cost is the third elasticity dimension [15], and understanding it is crucial for achieving cost-efficient elastic systems.

In this chapter we address the issue of monitoring costs and analyzing cost efficiency of elastic systems running in public clouds. To this end, we introduce a model for capturing the pricing schemes of cloud services. We define algorithms for evaluating costs of elastic systems, and their cost efficiency. We further analyze which system units are more cost efficient to scale-in and when, and provide cost-aware scale-in recommendations.

8.1 Introduction

Run-time costs evaluation is required for understanding and controlling elastic systems running in public clouds [78]. Currently, distributed systems deployed in public clouds can be built from a combination of proprietary software units, and public cloud services. Such services span from virtual infrastructure, to image storage, monitoring, and platform services such as message queues. Many such systems are elastic, capable of automatic/manual run-time reconfiguration according to particular requirements [15]. Depending on requirements, such reconfiguration can be through scale-out actions can allocate additional cloud services to run new instances of systems units. In turn, scale-in actions reduce the number of used cloud services, reducing the systems' running costs.

[44]The contributions from this chapter where originally presented in [35].

[44]In the original content ([35] we refer to cloud systems as cloud applications. For consistency, in this chapter we maintain the terminology used up to this point in the thesis.

While systems are scaled-out due to performance requirements, cost is the main driver for system scale-in [26, 27]. Cost-aware scalability controllers consider the costs of different types of cloud services used by the systems, rather than just manipulate the number of used services [28]. However, cost of elastic systems is complex, some services having multiple cost elements. For example, a VM service could be billed both every hour, and separately per each GB of generated I/O. Certain costs can be static, such as costs for reserving a cloud service [29, 30, 31]. Other costs can be dynamic, such as modifying the price of using a cloud service depending on the usage. Costs of cloud services can also depend on particular service combinations. For example, the reservation cost of an Amazon EC2[45] VM service depends both on its type, and its storage configuration, a VM optimized for high I/O costing more than a regular one. Additionally, public cloud services are usually billed over certain time and/or usage intervals. This means it might not be cost efficient to deallocate such services at any moment in time, such as deallocating a cloud service used for 10 minutes but billed for an entire hour.

Due to this cost complexity, developers of elastic systems need support in analyzing the systems' costs when running in public clouds. Even more, developing cost-aware scalability controllers requires detailed costs analysis, for improving the cost efficiency of the controlled systems. For cost efficiency, one must understand how much a system has used from a billed usage quota given by its cloud provider. For example, a cloud service billed per GB of I/O, which has generated 1.5 GB of I/O, can generate another 0.5 GB at no additional cost. The usage over cloud services employed by different elastic unit instances can also diverge in time, e.g., after several scaling operations. This can make different unit instances more/less cost-efficient to deallocate during scale-in operations. Such cost analysis information is required in cost-aware scalability to avoid deallocating unused but paid for cloud services, or by the systems, in improving their internal behavior to take advantage of the unused services.

8.2 Roadmap

The rest of the chapter is structured as follows. Section 8.3 presents the motivation and approach. Section 8.4 introduces our approach for analyzing cost and cost efficiency of cloud systems. Section 8.5 presents our prototype and experiments. Section 8.8 concludes the chapter and summarizes our contributions.

8.3 Motivation and approach

8.3.1 Motivation

It is not cost efficient to scale-in systems running in public clouds at any moment in time. In private clouds, services can be deallocated when they are no longer needed [79], as no billing is involved. However, public cloud providers usually bill services over crisp

[45]`http://aws.amazon.com/ec2/`

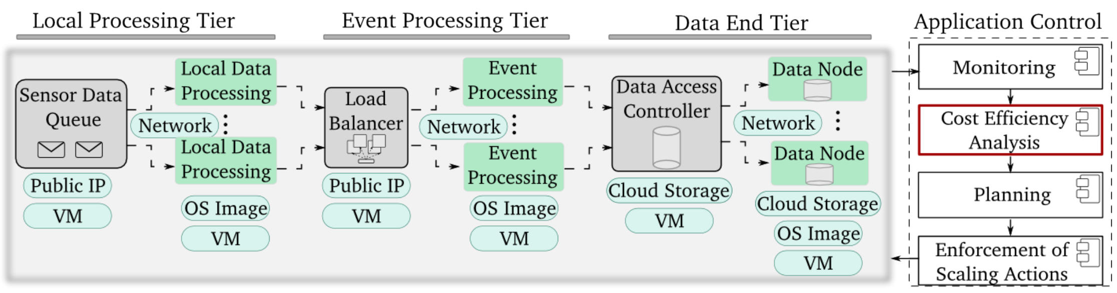


Figure 8.1: Data-as-a-Service elastic system (Chapter 3) with used cloud services

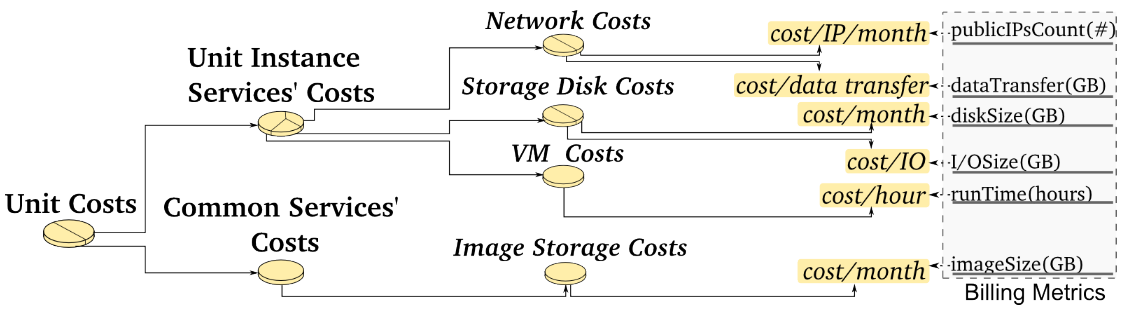
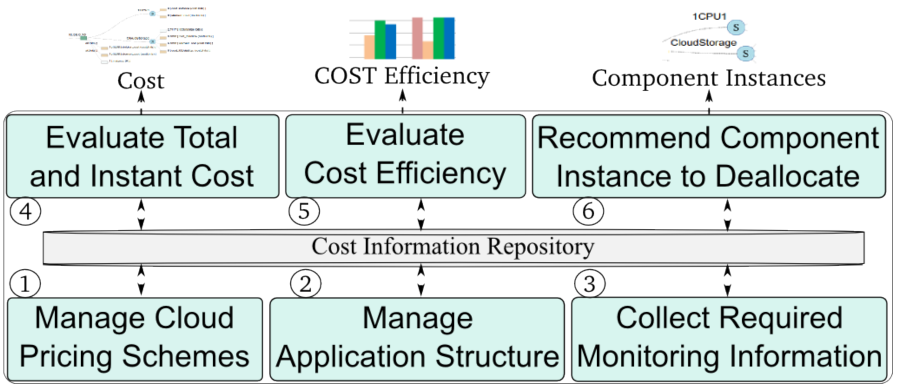


Figure 8.2: Example of scalable unit costs complexity

time and/or usage intervals, rounding up the service usage. For example, let us consider two VMs billed per hour and per GB of I/O. If allocated at the same time, and one generates 0.5 GB, and the second 0.9 GB, it is cost efficient to deallocate the second one, as the provider would bill each for one GB of I/O. If not allocated at the same time, their run-time costs must also be considered. Thus, for cost-aware scalability in public clouds, all costs must be analyzed, ensuring unused but paid for services are not deallocated.

Let us consider the Data-as-a-Service (DaaS) elastic cloud system from Chapter 3. Sensors send data to the system for processing, and are enabled/disabled depending on requirements (E.g., expected data frequency). Thus, the DaaS is designed as an elastic system. Its units are horizontally scaled by allocating/deallocating cloud services at run-time, to cope with varying demand, and reduce operating costs. The `Local Data Processing`, `Event Processing`, and `Data Node` units are horizontally scalable, around which a control mechanism is designed considering system operation costs and performance. At run-time, each of the system units use cloud services for achieving desired functionality.

All system's units use `Virtual Machine (VM)` and `Network` cloud services (Figure 8.1). The scalable units (depicted in pairs in Figure 8.1) use `OS Image` services for storing custom OS images used in allocating new units' instances. The `Data End` tier's units also use high performance `Cloud Storage` services. Finally, the `Sensor Data Queue` and `Load Balancer` use `Public IP` services for exposing their functionality to users running outside the cloud. The system's control mechanism enforces performance and costs requirements. It analyzes the system's state, plans and executes scaling actions



Cost
COST Efficiency
1CPU1
CloudStorage
Component Instances
Evaluate Total and Instant Cost
Evaluate Cost Efficiency
Recommend Component Instance to Deallocate
4
5
6
Cost Information Repository
1
2
3
Manage Cloud Pricing Schemes
Manage Application Structure
Collect Required Monitoring Information

Figure 8.3: Approach for cost-aware elasticity in public clouds

by adding/removing instances of the system's scalable units, relying on monitoring and costs information.

However, costs of elastic systems deployed in public clouds can be very complex. The costs of a scalable unit, e.g., the `Data Node` (Figure 8.2), can be composed of: (i) the cost for each `Common` service shared by all instances of the unit, e.g., OS image; (ii) and costs of all the cloud services used by each unit instance, e.g., `VM`, `Storage` or `Network`. Each cost element can be billed over different cost metrics.

For cost-aware scalability of systems in public clouds, developers and scalability controllers must understand the system's costs, posing several research questions:

- "How does each system unit contribute to the overall system's costs?", i.e., which unit is more expensive, and over which cost elements.
- "What are the instant system costs?", i.e., the rate at which system units are spending money.
- "What is the system's cost efficiency w.r.t., billed costs versus actual system usage?", allowing controllers to avoid deallocating services paid in full but underused (E.g., billed per hour, but used only 10 minutes).

8.3.2 Approach

We focus on aiding developers of elastic systems for public clouds to monitor their costs, and develop cost-aware scalability controllers. In our work cloud providers are black boxes, providing APIs for allocating/deallocating services on-demand and querying their pricing schemes. Developers are using such cloud services to run their systems, and want to maximize their usage for same or lower costs. The developers have no access to the inner workings of the clouds they use, interacting only with their user APIs.

For achieving cost-aware scalability in public clouds, in this chapter we develop a platform for monitoring costs and analyzing cost efficiency of elastic systems (Figure 8.3), providing:

- A model and mechanism for describing and managing complex pricing schemes of cloud services (①).
- A mechanism for describing elastic cloud systems and their used cloud services (②), leveraging the model defined in [5].
- A mechanism for collecting cost billing metrics from instances of system units and associating them to the system structure (③). The mechanism relies on our monitoring approach presented in Chapter 5.
- A service evaluating the total and instant costs for the system, its units, and used cloud services (④).
- A service evaluating cost efficiency of unit instances, providing insight in how much was used from what it has been paid for (⑤), to be used in improving the cost efficiency of elastic systems.
- A service recommending which system unit instance to be deallocated during scale-in operations based on desired cost efficiency, providing support for cost-aware scalability in public clouds (⑥).

8.4 Evaluating cost and cost efficiency of elastic systems

We first capture the pricing schemes of cloud providers. We then determine and evaluate the system's costs, depending on the cloud services used by the system.

8.4.1 Capturing complex cloud pricing schemes

Cloud providers offer multiple types of services. Each of these services can have configuration options offered under different costs. Depending on its type, a cloud service can have several cost elements, such as cost per reserving the service and cost per using it, expressed over time and usage. Thus, for cost analysis, we need to manage complex pricing schemes. To this end, based on previous work [32], we define a cost representation model centered on the *Cost Element* concept (Figure 8.4). Each *Cost Element* has a *type*, either service *Reservation* flat rate (E.g., hourly, monthly), or per *Usage*. A *Cost Element* is defined and computed per billing cycle, over its reservation time (E.g., used hours) or usage *Unit* (E.g., 1 GB I/O), over a service *Billing Metric* (E.g., VM uptime, I/O). Additionally, cost can be specified in intervals, captured using a *costFunction* specifying cost over intervals of measured *Units* over the billing *Metric*. Finally, a *Cost Element* is applicable if the service is used in a particular configuration, marked by an *applicableIf-ServiceHas* property, specifying which concrete *Resource* and *Quality* properties, or other *Cloud Services* the service should have. Using this model, we are able to describe any cost function, from simple fixed cost per used cloud service, to cost per service if used in conjunction with other services, considering both the service usage and reservation time.

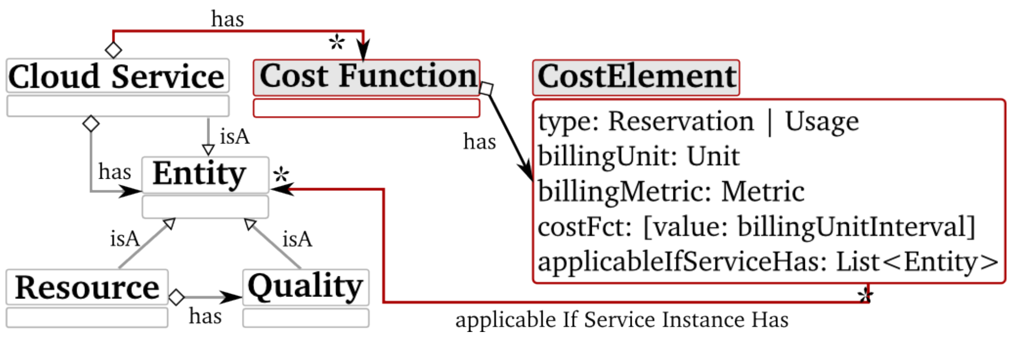

has
Cloud Service
Cost Function
CostElement
type: Reservation | Usage
billingUnit: Unit
billingMetric: Metric
costFct: [value: billingUnitInterval]
applicableIfServiceHas: List<Entity>
isA
has
Entity
has
isA
isA
Resource
Quality
has
applicable If Service Instance Has

Figure 8.4: Cost model of cloud offered services

Algorithm 7 Determining applicable costs

Input: $el : Unit|Topology|System,\ providers : cloud - providers$
Output: $aC : applicable\ cost\ scheme$

```
 1: function DetermineApplicableCost($el, providers$)
 2:     for s in el.usedServices do
 3:         service=$GetServiceCostDescription$(s,providers)
 4:         for costF in service.costFunctions do
 5:             for costEl in cf.costElements do
 6:                 costR = costEl.applicableIfHasResources
 7:                 costQ = costEl.applicableHasIfQuality
 8:                 costS = costEl.applicableIfUsesServices
 9:                 if el.$HasAll$(costR) && el.$HasAll$(costQ)&& el.$UsesAll$(costS) then
10:                         aC.$Put$(el,s,costEl)
11:                 end if
12:             end for
13:         end for
14:     end for
15:  return aC
16: end function
```

8.4.2 Determine applicable costs for current system configuration

During system run-time, if a unit is horizontally elastic, it will be instantiated more times, each instance using cloud offered services, potentially with different configurations. However, due to elasticity control, the system structure and associated cloud services can change at run-time [80, 28, 81]. To this end, after any enforced elasticity control, we use Algorithm 7 to analyze what cost elements are no longer applicable to the new system structure, and what new cost elements must be considered in cost evaluation. Depending on the concrete system configuration, we determine the applicable *Cost Elements* depending on the system's concrete configuration. For each cloud offered service used by each element (Line 2), we search in the managed cloud providers for the complete

Billing type	Billing function	Required monitoring information for computing service cost
per Reservation	Fixed	- none
	per Interval	- service reservation time (allocation/deallocation) - number of allocated service instances
per Usage	Fixed	- instant value of the cost billing metric
	per Interval	- instant value of the cost billing metric - summed historical values of the billing metric

Table 8.1: Service billing types in public clouds

offered service description, and possibly applicable cost functions (Line 3). Then, for each cost element, we verify its applicability conditions, by evaluating if the element uses the Resources, Quality, or associated Cloud Offered Services specified in the cost element's description. If a cost element is applicable in the current system configuration, it is used in cost evaluation.

8.4.3 Evaluating total and instant system cost

When evaluating cost of elastic systems, the information required by different cost billing types must be considered. We analyze main cloud providers (E.g., Amazon EC2[46], Flexiant[47], Azure[48], IBM Cloud[49], Rackspace[50]), and classify their cost billing types in Table 8.1. Cost can be billed as a fixed rate (E.g., α\$ per hour), or over an interval (E.g., first x GB of I/O free, next β\$). For computing *Fixed Reservation* cost, we retrieve from the cloud provider the cost for reserving the service. For *Interval Reservation* cost we monitor how many service instances where reserved and for how long, depending if the cost is billed over instance count or time. For *Fixed Usage* cost, we must monitor the billing metric, while for *Interval Usage* cost we need to record the historical usage over the billing metric, to determine the applicable cost interval.

Cost can be defined over intervals of time and usage (E.g., first GB free, second α\$). To use the correct pricing interval in cost evaluation, we must maintain an accurate view over the usage and lifetime of the cloud services employed by the system, for which we define Algorithm 8. Starting from the applicable cost previously determined with Algorithm 7, for each applicable cost element, we retrieve the billing metric (E.g., I/O size, disk size) specified in pricing scheme (Line 4). If the cost element specifies a *Reservation* billing for the service, we compute how many billing cycles have passed between the last two monitoring intervals (Line 7), and add it in the system usage. If the cost is per *Usage*, we need to record the total usage over the billing metric (Line 13). Some monitored metrics can be *cumulative*, i.e., never reset, continuously increasing by adding new monitoring values to the previous ones. With such metrics, we use their value for the updated system usage (Line 15). Otherwise, we add the newly collected value to

[46]https://aws.amazon.com/ec2/
[47]https://www.flexiant.com/
[48]https://azure.microsoft.com/
[49]http://www.ibm.com/cloud-computing/
[50]http://www.rackspace.com/

Algorithm 8 Determining usage of cloud services

Input: $el : Unit|Topology|System$, p : *previousely determined usage*
Input: aC : *applicable cost scheme*, m : *current monitoring snapshot*
Output: u : *updated element usage snapshot*

```
1: function UpdateElementUsage($el, p, m, aC$)
2:     for s in el.usedServices do
3:         for ce in aC.GetApplicableCost(el, s) do
4:             metric = ce.billingMetric
5:             if ce.type == Reservation then
6:                 t=GetTimeBetween(m.timestamp,p.timestamp)
7:                 billingCycles=GetBillingCycles(metric, t)
8:                 if p.Contains(s) then
9:                     billingCycles += p.GetLifetime(s,el)
10:                end if
11:                u.UpdateLifetime(s,el,billingCycles)
12:            else if ce.type == Usage then
13:                currVal = m.GetValue(metric,s,el)
14:                if metric.type == Cumulative then
15:                    u.SetValue(s,el,metric,currVal)
16:                else
17:                    prevVal = 0
18:                    if p.Contains(s) then
19:                        prevVal = p.GetValue(metric,s,el)
20:                    end if
21:                    eVal = EstimateMissing(p, metric, prevVal, currVal,t)
22:                    usage = prevVal + currVal + eVal
23:                    u.SetValue(s,el,metric, usage)
24:                end if
25:            end if
26:        end for
27:    end for
28:    return u
29: end function
```

the previous values, and compute the total (Lines 17-23). At run-time, monitoring data might not be always available in time, or it might be missing. To solve such issues, missing values could be estimated (Line 21), e.g., through interpolation.

For cost-based elasticity control, system developers and controllers require information about the system's instant cost, i.e., the rate at cost units are billed for the current system configuration. Such information is crucial in evaluating if the current cost of the system and its units is too high, and determining appropriate elasticity control. Thus, based on latest monitoring information and the total system usage previously evaluated with

Algorithm 9 Determining system instant cost

```
Input: $el : Unit|Topology|System$, $sU$ : total system usage snapshot
Input: $aC$ : applicable cost scheme, $m$ : current monitoring snapshot
Output: $iC$ : instant element cost
 1: function EvalInstantCost($el, sU, aC, m$)
 2:     elementCost=0
 3:                                     ▷ compute instant cost rate for each used service
 4:     for s in el.usedServices do
 5:         for ce in aC.Get(el, s) do
 6:             metric = ce.billingMetric
 7:             if ce.type == Reservation then
 8:                 lifetime = sU.GetLifetime(s,el)
 9:                 cmICost = ce.GetCostForValue(lifetime)
10:                 iC.SetValue(metric, s, el, cmICost)
11:             else if ce.type == Usage then
12:                 metricVal = m.GetValue(metric, s, el)
13:                 cPerUnit=ce.GetCostForValue(metricVal)
14:                 if metric.type == Cumulative then
15:                     iC.SetValue(metric, s, el, cPerUnit)
16:                 else
17:                     cmICost = metricVal * cPerUnit
18:                     iC.SetValue(metric, s, el, cmICost)
19:                 end if
20:             end if
21:             elementCost += cmICost
22:         end for
23:                                     ▷ compute children cost and overall element cost
24:         for childEl in el.children do
25:             cChild = EvalInstantCost(childEl, sU, aC, m)
26:             iC.AddCost(cChild)
27:             elementCost += iC.GetElementCost(childEl)
28:         end for
29:     end for
30:     iC.SetElementCost(elementCost)
31:   return u
32: end function
```

Algorithm 8, we apply Algorithm 9 to evaluate the instant cost of an elastic system. For each of the cloud services used by the system units (Line 4), we analyze each applicable cost element (Line 5). For cost per *Reservation*, we compute the applicable cost value based on the element's cost intervals, w.r.t. the lifetime of the used service (Line 9), and store the value directly in the instant cost. For cost per *Usage*, we determine based on the system usage so far the applicable cost value (Line 13). If the billing metric is cumulative, we add the applicable cost rate to the instant cost (Line 15). Otherwise, the latest monitored metric value is multiplied with the applicable cost value, and added in the instant cost (Lines 17-18).

Developers or system controllers might be interested only in certain cost information. For example, they could monitor overall system's cost until it reaches a certain threshold, after which they would be interested in which of the system's units and topologies are most expensive, to enforce appropriate cost-based elasticity control. Thus, to understand the contribution of each cost element to the overall system cost, for each system unit, we compute its overall instant cost by recursively computing the cost of all its used cloud services. Further, for each system topology and overall system we compute its cost from the cost of its children units and topologies, and cloud services used directly by the system or topology (Lines 24-28). Computing composite cost of each system unit and topology from the cost of its children and its used services provides a composite hierarchical view over system's cost, enabling developers or controllers to extract from the cost snapshot only the cost information of interest.

We apply a similar algorithm for computing total system cost since deployment, for total cost multiplying the total system usage (obtained using Algorithm 8) with the cost element's values, for each of the element's cost intervals.

8.4.4 Evaluating cost efficiency of elastic systems

We define a function evaluating the cost efficiency of deallocating system unit instances, based on the pricing scheme of the cloud services used by the system.

After multiple allocations/deallocations of cloud services during scaling, the billing cycles of different instances of the cloud services used by different system units can become desynchronized, i.e., billing occurs at different points in time, depending on when each instance was allocated, and on how much it was used (Figure 8.5). Thus, for increasing the system's cost efficiency, developers and controllers need to understand which particular instance of a service is more cost efficient to deallocate, and when. This is important as the system gains nothing if a service is deallocated without using all what it

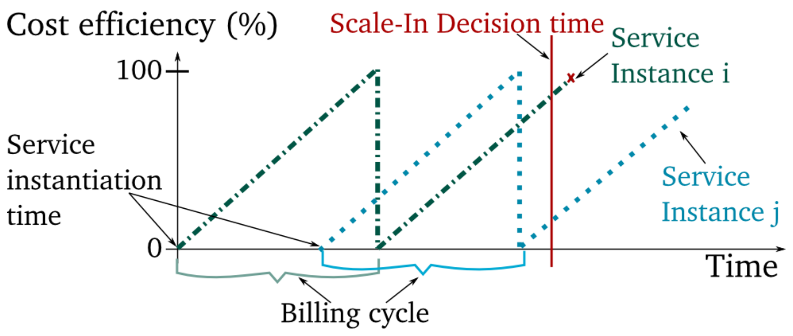


Figure 8.5: Service instance cost efficiency w.r.t. billing cycle

was paid for. Moreover, if the system's load is highly fluctuating, opportunistic scaling-in can decrease cost efficiency by deallocating underused services, and then allocating new ones to cope with rising demand.

When scaling-in an elastic system by deallocating cloud services, to maximize cost efficiency, the developer/controller should deallocate the service instance closest to its billing cycle for all its cost elements (E.g., *Instance i*). Ideally, a cost efficient system would deallocate a service instance with a usage of 100% on all its cost elements, w.r.t., the billing cycle of each element. For example, if a service is billed per hour and per GB of data, efficient from the cost perspective is to deallocate a service instance which has been running for an integer number of hours, and which has generated an integer number of GBs of data. To this end, based on the system total usage determined using Algorithm 8, we define a function E for evaluating the cost efficiency of deallocating a particular cloud service instance s_i, by computing a cost-weighted sum of the instance usage over all its cost elements, both over reservation time and monitored usage, reported to the overall billed cost units, as follows:

When scaling a cloud system, a cost-aware controller should deallocate the unit having its services closest to their billing cycle for all cost elements (E.g., *Instance i* in Figure 8.5), to maximize cost efficiency. Ideally, a cost-aware controller would deallocate a unit instance with a usage of 100% over all its cost elements, w.r.t., the billing cycle of each element. For example, if a unit uses cloud services billed per hour and per GB of data, it is cost efficient to deallocate it when it has run for an integer number of hours, and has generated an integer number of GBs. To this end, we define a function E for evaluating the cost efficiency of deallocating a system unit instance i, based on the system total usage obtained with Algorithm 8. Function E computes a cost-weighted sum of the instance usage over all its cost elements, both over reservation time and monitored usage, reported to the overall billed cost units:

$$E(i) = \frac{\sum\limits_{s \in i.serv} \sum\limits_{c \in s.cEl} c_{.value} * (s_{.usage}(c_{.metric}) \bmod c_{.cycle}))}{\sum\limits_{s \in i.serv} \sum\limits_{c \in s.cEl} c_{.value}} \tag{8.1}$$

, where $i.serv$ are the cloud services used by the unit instance i; $s.cEl$ are the applicable cost elements for used cloud service s; $c_{.value}$ is the cost value in units of the cost element c (E.g., I/O cost) for one billing cycle $c_{.cycle}$ (E.g., 1 GB, 1 hour); $s_{.usage}(c_{.metric})$ is the current usage interval over the metric $c_{.metric}$ over which cost is billed.

Weighting the instance usage with the cost value ensures that each cost element has a contribution proportional to its cost to the cost efficiency. Thus, a cost efficiency E of 1 means that the instance usage is the same as the total billed usage, and by deallocating it, the cost efficiency does not decrease. Reversely, an efficiency < 1 means we pay 1-efficiency% more than what we use. Depending on individual system requirements, other custom cost efficiency functions can be defined based on the same system total usage and cloud pricing schemes. For example, another function might only analyze cost efficiency of cost elements reported per service reservation.

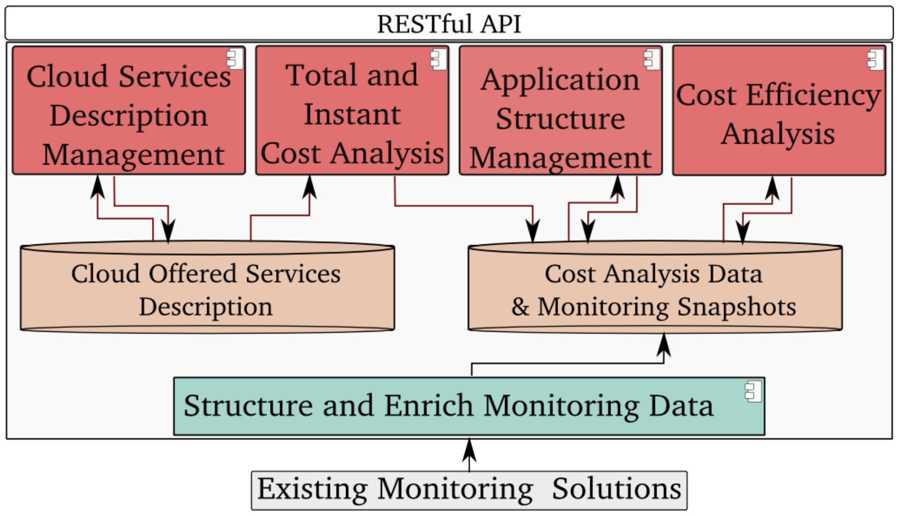

RESTful API
Cloud Services Description Management
Total and Instant Cost Analysis
Application Structure Management
Cost Efficiency Analysis
Cloud Offered Services Description
Cost Analysis Data & Monitoring Snapshots
Structure and Enrich Monitoring Data
Existing Monitoring Solutions

Figure 8.6: Cost analysis platform

Depending on the desired cost efficiency, a developer of controller can decide to instantly deallocate the most cost efficient instance, or wait until one of the instances reaches the desired cost efficiency before deallocating it.

8.5 Cost analysis platform prototype

Applying our cost efficiency analysis approach from Section 8.4, we have implemented a cost analysis platform[51] (Figure 8.6) by extending MELA, a platform for monitoring and analyzing elastic cloud systems, first introduced in Chapter 5.4.4.

8.5.1 Managing pricing schemes of cloud providers

Cloud providers have different mechanisms for exposing their services' pricing schemes, from proprietary APIs to plain HTML descriptions. Thus, the first concern the platform addresses is to provide an easy to use mechanism for describing the cloud services and their cost elements from any cloud provider. To this end we extended MELA with a unit for managing the description of cloud services, relying on an XML-based representation of the pricing model defined in Section 8.4.1. As manually managing XML descriptions can be difficult, we provide a Java-based Fluent API (Listing 8.1) for generating them. Thus, adapters for retrieving the pricing scheme of custom cloud providers can be built to use the fluent API to generate the cloud's XML description, and submit it to our platform using a specific RESTful service.

[51]Prototype and supplement materials at http://tuwiendsg.github.io/MELA/costEvaluationService.html

Listing 8.1: Cloud pricing scheme description fluent API

```
CloudService service = new CloudService()
  .withUuid(UUID).withName("Name")
  .withCategory("Name").withSubcategory("Name")
  .withCostFunction(new CostFunction()
    .withAppliedIfServiceInstanceUses(List<Unit>)
    .withCostElement(new CostElement()
      .withCostMetric(new Metric("name", "unit/time",Type))
      .withBillingInterval(new MetricValue(v), costUnits)
      .withBillingInterval(...
  ).withCostElement(...
```

8.5.2 Managing the structure of elastic cloud systems

Elastic cloud systems change their structure at run-time by allocating/deallocating cloud services. Thus, we provide a mechanism for updating the system's structure and the used cloud services during run-time, relying on the XML representation of the model introduced in [33]. Using this model, a cloud system is represented as a cascading set of `Monitored Elements` representing the system's units, which in turn are grouped in topologies. We extend the model to allow specification of the used cloud services, and provide a fluent API for describing the system's structure (Listing 8.2). The used cloud service instances are identified by the unique identifier (UUID) of the cloud provider offering the service, the UUID of the service in the provider's service's list, and the UUID of the service's instance. Further, if the same service has different cost for specific configurations, the concrete configuration is specified in terms of resource/quality properties. The description must be updated after each scaling action, to ensure cost analysis consistency. The platform exposes the description to interested stakeholders in XML or JSON format, and as a tree-based visualization.

Listing 8.2: Cloud system structure description fluent API

```
 MonitoredElement vm = new MonitoredElement("UUID")
    .withCloudOfferedService(
          new UsedCloudOfferedService()
             .withCloudProviderID("UUID"))
             .withCloudProviderName("Provider")
             .withId("UUID").withInstanceUUID("UUID")
             .withName("ServiceName")
             .withResourceProperties(Map<Property, Value>)
             .withQualityProperties(Map<Property, Value>)
    ).withCloudOfferedService(...
```

8.5.3 Collecting and enriching monitoring information

Due to the dynamic structure of elastic systems, as VMs are created/destroyed dynamically at run-time, if monitoring information is associated only with each VM, it will be lost during scale-in operations. Addressing this, we use MELA Data Service, which provides a mechanism for describing metric composition rules, enriching and associating monitoring information collected from existing monitoring systems, to the current system structure.

For example, associating a CPU usage metric collected from a VM to the instance of the system unit hosted on the VM. Moreover, monitoring data can be enriched by applying aggregation operations over collected metrics (E.g., average CPU usage over all instances of a unit), or injecting information using a `set` operation. The metric composition rules can be described directly in XML, or using a fluent API.

8.5.4 Evaluating and providing cost information

Using the enriched monitoring information, the system structure, and the cloud service's description, the algorithms from Section 8.4 are applied to evaluate the total and instant cost, and cost efficiency of cloud systems. This information is exposed through RESTful services, and the complete cost decomposition is also provided in comma-separated-values (CSV) format. Services are implemented for evaluating the cost efficiency of scaling-in one or more instances of a system unit, and for recommending which unit instance to deallocate w.r.t., to desired cost efficiency. For aiding human users, we provide web-based visualizations for the composite cost of elastic systems, both using tree and pie charts, implemented in D3.js[52]. The visualizations enable them to monitor and understand the cost of cloud systems. To ensure platform performance, after each evaluation of total and instant cost, we cache the total computed usage and lifetime for each used cloud service. This ensures a roughly constant cost evaluation time for each new monitoring snapshot, in evaluation obtaining a cost analysis time of under 1 second for each monitoring snapshot.

8.6 Experimental scenarios

We evaluate our platform on the elastic cloud data center for IoT from Section 8.3 (Figure 8.1), deployed in Flexiant[53] public cloud. The system developer deploys our platform on a standalone VM, and uses it to understand the system's cost under expected load. To this end, a load is applied starting with 300 sensors, increasing to 900 in steps of 300, and decreasing again. While the used system has a shared-nothing architecture, each unit using its own cloud services, our approach is also applicable on shared architectures, with multiple units sharing a cloud service. In this case finer-grained monitoring is required to differentiate the service usage generated by different units.

8.6.1 Configuring the platform for the target system

First, the system developer submits to our platform the pricing scheme of the cloud services offered by Flexiant[54](Table 8.2). While some services have monthly cost rates, the billing is done per hour, and thus the pricing scheme is specified in hourly rates. The offered services are classified using our API in `IaaS` (Infrastructure as a Service),

[52]`http://d3js.org/`

[53]We use Flexiant as an illustrative example in this chapter, but similar issues apply to other public cloud providers

[54]`www.flexiant.com/2010/04/14/flexiscale-2-0-pricing/`

Service			Cost Element				
Category	Subcategory	Service	metric	Billing unit	Billing cycle	Billing interval	Cost units
IaaS	VM	1CPU, 0.5GB RAM	instance	#	hour	0→Inf.	2
		1CPU, 1GB RAM					3
		1CPU, 2GB RAM					4
		2CPU, 4GB RAM					10
		3CPU, 6GB RAM					15
		4CPU, 8GB RAM					20
	Disk	VM-attached cloud storage	disk size	GB	month	0→Inf.	5
			I/O	GB			2
	Network	Public	data transfer	GB		0→Inf.	5
		Private	instance	#	month		0
		VLAN	instance	#	month	0→1	0
						2→Inf.	1000
		IP	instance	#	month	0→5	0
						6→Inf.	100
		Firewall	instance	#	month	0→Inf.	1
MaaS	Image	Custom operating system image	snapshot size	GB	month	0→Inf.	5

Table 8.2: Flexiant pricing scheme

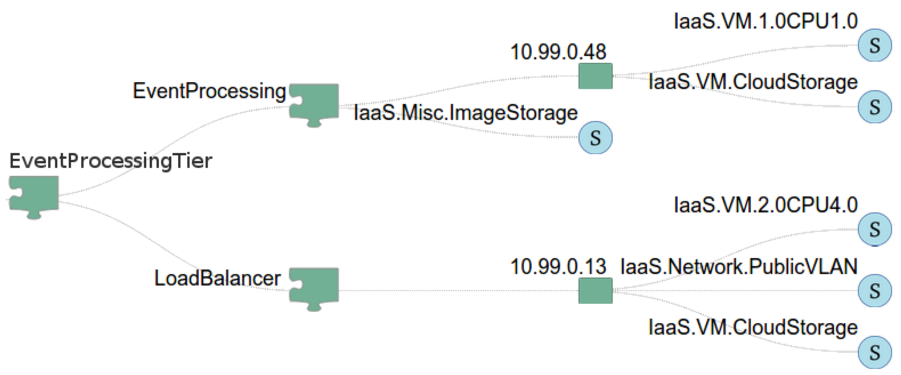


Figure 8.7: Event Processing topology structure and used cloud services

offering `VMs`, `Network` and `Disk`, as a service, and `MaaS` (Management as a Service) offering `OS Image Storage` services.

Next, the developer describes the system structure with the used cloud services. As Flexiant provides separate billing for `VM` and `Cloud Storage` services, separate instances are specified for each unit. units accessible from outside the cloud use `Public VLAN` services providing public IPs (E.g., `Load Balancer`) and the elastic units use `OS Image Storage` services in instantiating new unit instances on top of new `VMs` (E.g., `Event Processing`). The developer visualizes the described system's structure, Figure 8.7 depicting the `Event Processing` topology's structure after the initial cloud deployment (before scaling), the used cloud services being represented with Ⓢ.

Next, the system developer must ensure the necessary monitoring information can be collected, i.e., exposing the billing metrics specified by the cost elements captured in Table 8.2. Thus, the system is configured to expose the required information through

Ganglia[55], by implementing the required plug-ins, such as to collect disk I/O, and then structure it using MELA [33]. Metric resolution for associating the metrics targeted by cost elements to the ones collected from the elastic service is done based on metric name, and measurement unit. Thus, care must be taken to ensure that we will have in the monitoring information the metrics targeted by the cost elements. This can be done either by implementing data collection plug-ins returning the metrics with the same name and unit, or use MELA metric composition rules to rename and convert monitoring information, and even create aggregated complex metrics based on collected ones.

Listing 8.3: Fluent API for describing Cloud Pricing Scheme

```
CloudService service = new CloudService()
  .withUuid(UUID).withName("Name")
  .withCategory("Name").withSubcategory("Name")
  .withCostFunction(new CostFunction()
    .withAppliedIfServiceInstanceUses(List<Unit>)
    .withCostElement(new CostElement()
      .withCostMetric(new Metric("name", "unit/time",Type))
      .withBillingInterval(new MetricValue(v), costUnits)
      .withBillingInterval(...
  ).withCostElement(...
```

8.7 Evaluation

8.7.1 Evaluating cost of black-box systems

In our first experiment we apply our cost analysis approach to discover interesting cost aspects of black-box systems, over which there is minimum knowledge about their internal behavior. The system is a Spark[56] cluster consisting of one master and one worker, used for stream processing of IoT sensor data, which could be used for the DaaS's Data End. In this scenario, the system developer has a candidate for the DaaS Data End, and wants to understand how changing the system load on Spark influences the system's cost. The system is deployed in Flexiant[57] public cloud, both Spark units using instances of the same type of `1CPU 1 GB RAM` VM and `Cloud Storage` services, and the master additionally using a `Public VLAN` providing a public IP to which data is streamed.

Initially, a developer or cloud provider uses our fluent API (Listing 8.3) for describing the pricing scheme of cloud services offered by Flexiant[58](Table 8.2). While the pricing scheme specifies monthly cost rates for several services, the billing is done per hour, and thus the pricing scheme is specified in hourly rates. The offered services are classified in `IaaS` (Infrastructure as a Service), offering `VMs`, `Network` and `Disk`, as a service, and `MaaS` (Management as a Service) offering OS `Image Storage` services. The system

[55]`http://ganglia.sourceforge.net/`

[56]`https://spark.apache.org/`

[57]We use Flexiant as an illustrative example in this chapter, but similar issues apply to other public cloud providers

[58]`www.flexiant.com/2010/04/14/flexiscale-2-0-pricing/`

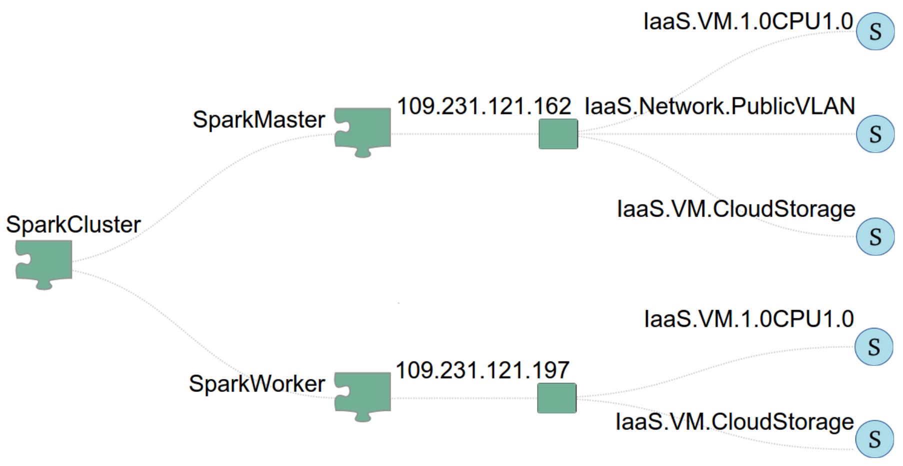


Figure 8.8: Spark cluster structure and used cloud services

configuration with the used cloud offered services attached to the system units and topologies is visualized using our platform in Figure 8.8 (used cloud services are depicted with Ⓢ). As Flexiant provides separate billing for VMs and cloud storage, separate instances of `VM` and `Cloud Storage` services are specified for each unit (E.g., `Spark Master`).

Next, the system developer must ensure the necessary monitoring information can be collected, i.e., exposing the billing metrics specified by the cost elements captured in Table 8.2. Thus, the system is configured to expose the required information through Ganglia[59], by implementing the required plug-ins, such as to collect disk I/O, and then structure it using MELA [33]. Metric resolution for associating the metrics targeted by cost elements to the ones collected from the elastic service is done based on metric name, and measurement unit. Thus, care must be taken to ensure that we will have in the monitoring information the metrics targeted by the cost elements. This can be done either by implementing data collection plug-ins returning the metrics with the same name and unit, or use MELA metric composition rules to rename and convert monitoring information, and even create aggregated complex metrics based on collected ones.

What is the relationship between system cost and load? is the first question answered in this experiment. As Spark is used for stream processing, the owner expects the impact to be only on network data transfer cost. Thus, four separate Spark instances are deployed and run for 5 hours, each processing data sent every second from 0, 1000, 5000, and 10.000 sensors. By analyzing retrieved total cost for each system every hour (Figure 8.9), the owner understands that running the system 1 hour for 1000 sensors/second costs the same as running the system with no load. However, if the system is run more hours, cost for serving 1000 sensors increases with approx. 30%. Compared to serving 1000 sensors, serving 5000 for 5 hours costs with approx. 80% more, while serving 10.000 sensors costs with 40% more than serving 5000. From this information, the developer understands

[59]`http://ganglia.sourceforge.net/`

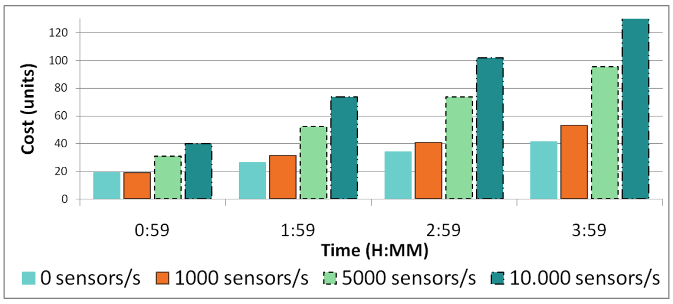


Figure 8.9: Spark system cost for different load

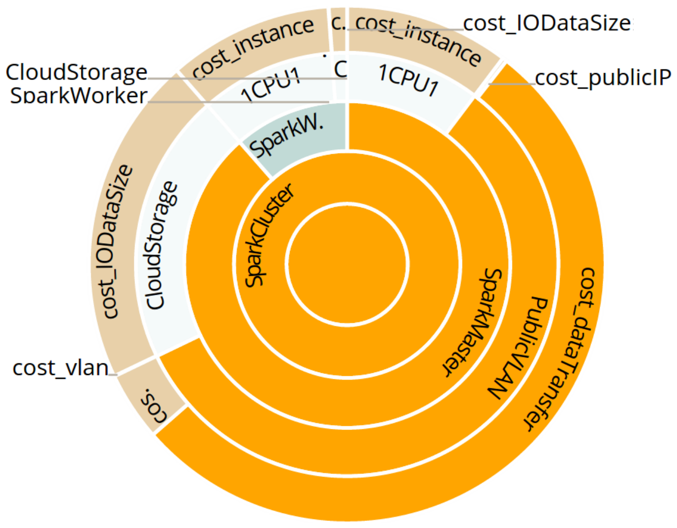
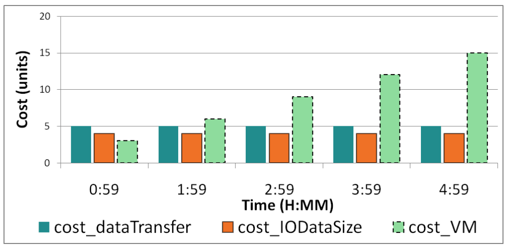


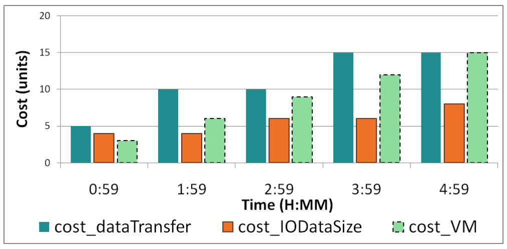
Figure 8.10: Composite cost of system serving 10.000 sensors/s

that the system's cost grows roughly with a factor of less than half the increase in the sensors' number. Thus, if the system is to be exposed as a service, the developer can predict its cost and understand its cost efficiency for the expected load.

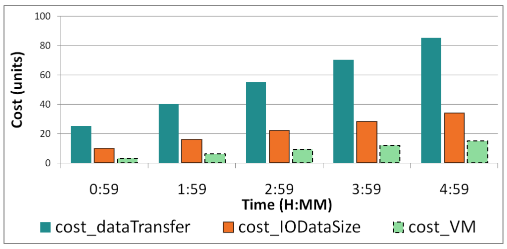
Which is the most expensive system unit? can be determined from the contribution to system cost of each system unit, the owner understanding that when the load is high, the `Spark Master` is the most expensive unit (Figure 8.10). Thus, the owner retrieves the decomposed cost data, to understand how different system load influence the cost elements applicable to the `Spark Master`, from which the most important where determined cost per used VM, network data transfer, and I/O. For the system processing no data, the cost is dominated by the cost per used VM (Figure 8.11a). When the system load is increased to 1000 sensors (Figure 8.11b), the data transfer cost becomes at least as important as cost per VM. However, when the system is serving 10.000 sensors/second (Figure 8.11c), cost of data transferred over the network becomes the dominant cost unit. With this information, depending on the envisioned load, system developers can determine where cost can be reduced. For example, if the system is expected to serve



(a) no load



(b) serving 1000 sensors/s



(c) serving 10.000 sensors/s

Figure 8.11: Spark Master cost elements decomposition

over 1000 sensors/s, it is worthwhile to consider compression of sensor data received by the system to reduce data transfer cost.

What are the most important cost elements? is answered from the cost decomposition for each load (E.g., Figure 8.11c). While the developer expected the system load to influence only data transfer cost, disk I/O cost emerged as the second most important cost element. Moreover, cost per disk I/O tends to be equal to roughly 40% of data transfer cost when the system is under load. Using this information, a developer could reduce overall system cost by reducing the number of I/O bound operations, such as logging.

Thus, even with a black-box system, using our approach, one can obtain crucial insight into what are the dominant cost elements, information useful in predicting system cost under different loads, and in reducing system cost by understanding what cost aspects needs to be analyzed/improved.

8.7.2 Improving cost efficiency of elastic systems

In our second experiment, we analyze cost and improve cost efficiency of a complex elastic system. In this scenario, the developer of the elastic cloud data center for IoT from Chapter 3 (Figure 8.1) is interested in understanding the cost of each unit, topology, and whole system. The developer further wants to design a cost-efficient elasticity controller, for which s/he needs to understand how different cost elements influence the system's cost efficiency.

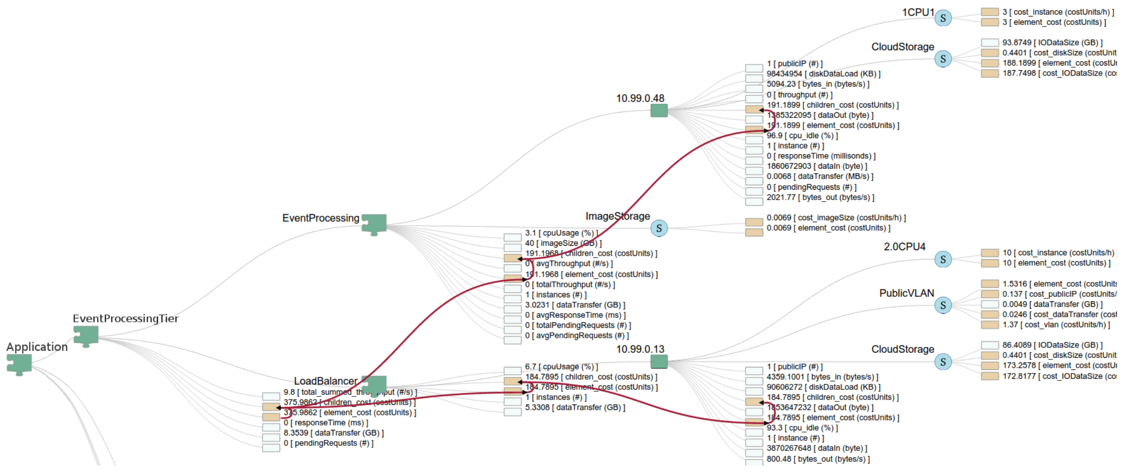

Application
EventProcessingTier
EventProcessing
LoadBalancer
9.8 [total_summed_throughput (#/s)]
375.9862 [children_cost (costUnits)]
375.9862 [element_cost (costUnits)]
0 [responseTime (ms)]
8.3539 [dataTransfer (GB)]
0 [pendingRequests (#)]
3.1 [cpuUsage (%)]
40 [imageSize (GB)]
191.1968 [children_cost (costUnits)]
0 [avgThroughput (#/s)]
191.1968 [element_cost (costUnits)]
0 [totalThroughput (#/s)]
1 [instances (#)]
3.0231 [dataTransfer (GB)]
0 [avgResponseTime (ms)]
0 [totalPendingRequests (#)]
0 [avgPendingRequests (#)]
6.7 [cpuUsage (%)]
184.7895 [children_cost (costUnits)]
184.7895 [element_cost (costUnits)]
1 [instances (#)]
5.3308 [dataTransfer (GB)]
ImageStorage
0.0069 [cost_imageSize (costUnits/h)]
0.0069 [element_cost (costUnits)]
10.99.0.48
1 [publicIP (#)]
98434954 [diskDataLoad (KB)]
5094.23 [bytes_in (bytes/s)]
0 [throughput (#)]
191.1899 [children_cost (costUnits)]
1385322095 [dataOut (byte)]
191.1899 [element_cost (costUnits)]
96.9 [cpu_idle (%)]
1 [instance (#)]
0 [responseTime (millisonds)]
1860672903 [dataIn (byte)]
0.0068 [dataTransfer (MB/s)]
0 [pendingRequests (#)]
2021.77 [bytes_out (bytes/s)]
1CPU1
3 [cost_instance (costUnits/h)]
3 [element_cost (costUnits)]
CloudStorage
93.8749 [IODataSize (GB)]
0.4401 [cost_diskSize (costUnit
188.1899 [element_cost (costUn
187.7498 [cost_IODataSize (co
10.99.0.13
1 [publicIP (#)]
4359.1001 [bytes_in (bytes/s)]
90606272 [diskDataLoad (KB)]
184.7895 [children_cost (costUnits)]
1853647232 [dataOut (byte)]
184.7895 [element_cost (costUnits)]
93.3 [cpu_idle (%)]
1 [instance (#)]
3870267648 [dataIn (byte)]
800.48 [bytes_out (bytes/s)]
2.0CPU4
10 [cost_instance (costUnits/h)
10 [element_cost (costUnits)]
PublicVLAN
1.5316 [element_cost (costUnits
0.137 [cost_publicIP (costUnits/
0.0049 [dataTransfer (GB)]
0.0246 [cost_dataTransfer (cost
1.37 [cost_vlan (costUnits/h)]
CloudStorage
86.4089 [IODataSize (GB)]
0.4401 [cost_diskSize (costUnit
173.2578 [element_cost (costUn
172.8177 [cost_IODataSize (co
S

Figure 8.12: System partial snapshot with composite instant cost associated to monitoring information

Evaluating cost of elastic systems

As previously, the system configuration with used cloud services is described. Beside `VM` and `Cloud Storage` services, units which must be accessible from outside the cloud also use and pay for `Public VLAN` services providing public IPs (E.g., `Load Balancer`). As several system units are elastic, `OS Image Storage` services are used for speeding up the creation and instantiation of new unit instances on top of new `VM` instances (E.g., `Event Processing Unit`). To evaluate the cost of the used `OS Image Storage` service, information hard to monitor directly is required, such as the size of each used image. While this could be obtained from specific cloud provider APIs, in this experiment the developer defines additional metric composition rules that create the required image size metrics and associates them to the elastic system units using images. In this way the same image is monitored only once per system unit, and not once per each individual VM. As the system owner is interested in understanding the system's cost under expected load, a system load is applied according to expected usage, starting with 300 sensors, increasing to 900 in steps of 300, and decreasing to 300 again.

What is the contribution to total cost of each system unit and topology? is visualized using our platform in Figure 8.13. The system cost is composed of the cost of the system children topologies. In turn, the topologies' cost is composed of the cost of their children units. The cost of individual units is composed of the cost of used instances of `VM` services, the cost of the `Cloud Storage`, reported both in cost/disk size, and per size of read/written data, and the cost of additional used services, such as the `Public VLAN`. From this cost decomposition the owner/developer understands (from the cost proportion of each cost element) which system units have low cost impact (E.g., `Local Processing Unit`), and which are costly (E.g., `Data Node`). Further, the owner/developer sees that, under the given system load, the `Cloud Storage` cost is roughly 1/3 of the `Event Processing Unit` cost. Further, the biggest contributor to

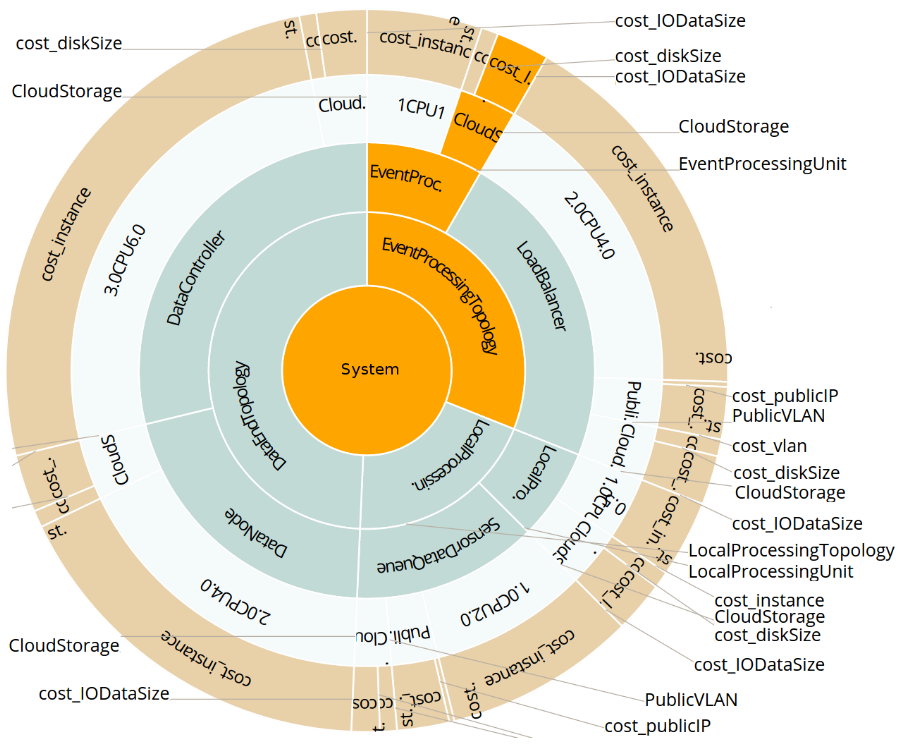


Figure 8.13: System snapshot of total cost composition

storage cost is IO DataSize, i.e., cost for read/written data. This is not the case for all units, such as Data Controller, for which the cost of using the 3 CPU 6GB RAM VM is much higher than the Cloud Storage cost. Using this information, a system developer or software elasticity controller can reduce system's cost by changing I/O rate for the Event Processing Unit, choosing services with less I/O cost for running the unit's instances, or migrating to a cloud with lower I/O cost.

System developers and software elasticity controllers traditionally enforce elasticity decisions with respect to collected monitoring information. Thus, for supporting real-time cost-based elasticity control, our platform exposes evaluated instant and total cost along monitoring information. Figure 8.12 depicts such a monitoring snapshot visualized by our platform, evaluated cost being represented by darker colored rectangles. Furthermore, our platform aggregates cost of individual unit instances in overall cost for each unit, topology, and the overall service, under an element_cost metric (highlighted with thick line). In this snapshot, cost of used cloud offered services (E.g., Cloud Storage) is associated to the unit instances running inside virtual machines. The cost of each unit instance is further composed of the cost of each unit (E.g., Event Processing Unit), which in turn is composed in the cost of the parent topologies, up to the cost of the entire system. Thus, a developer or software elasticity controller, based on the desired cost level (such as only topology, unit, or overall system), can retrieve cost at the required level as any other monitored information.

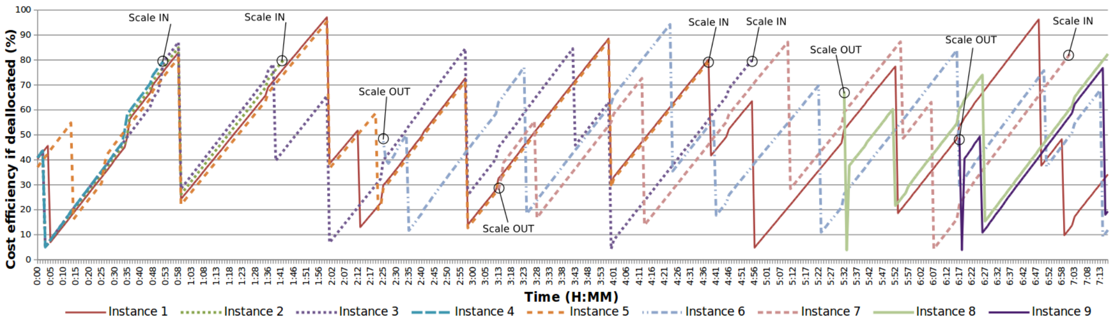


Figure 8.14: Cost efficiency variation between Event Processing Unit's instances after scale-in/out actions

Strategy type	Deallocating	Strategy description
Cost-aware	w.r.t. Reservation time	- deallocates the component instance if it has run over 90% of its reservation cycle
	w.r.t. Cost efficiency	- deallocates the unit instance with cost efficiency over 80%
Cost-agnostic	Last allocated	- deallocates the last allocated unit instance
	First allocated	- deallocates the first allocated unit instance

Table 8.3: Evaluated scalability control strategies

Evaluating cost efficiency of elastic systems

In this scenario, we apply our approach from Section 8.4.4 for improving the cost efficiency of the elastic system described in Section 8.7.2, by providing cost efficiency information and scale-in recommendations to its elasticity controller. While the presented elastic system has three horizontally elastic topologies, in the following we focus on controlling through horizontal scaling the `Event Processing` unit. Focusing on the `Event Processing Unit` both fully covers the issues present in cost efficiency of elastic systems, and increases the readability of the evaluation scenario.

For evaluating the system's cost efficiency improvement when elasticity controllers consider cost efficiency, an elasticity controller is implemented supporting different scale-in strategies (Table 8.3): two which do not consider cost efficiency, and two relying on our cost efficiency analysis. The two cost-efficiency agnostic strategies deallocate the `Last allocated` and `First allocated` system unit instance. The first cost-based strategy deallocates a unit it has run over 90% of its reservation billing cycle, i.e., run over 54 minutes. The second cost-based strategy deallocates a system unit instance only when its cost efficiency is over 80%. Due to the difference in billing cycles for the two cost elements, as visible in Figure 8.14, 90% cost efficiency is not always achievable in this scenario, and using such a limit would mean we might increase cost by running instances more billing cycles. We deploy and control the elastic `Event Processing Topology` for each strategy for over 12 hours. We apply a fixed system load of 450 sensors per

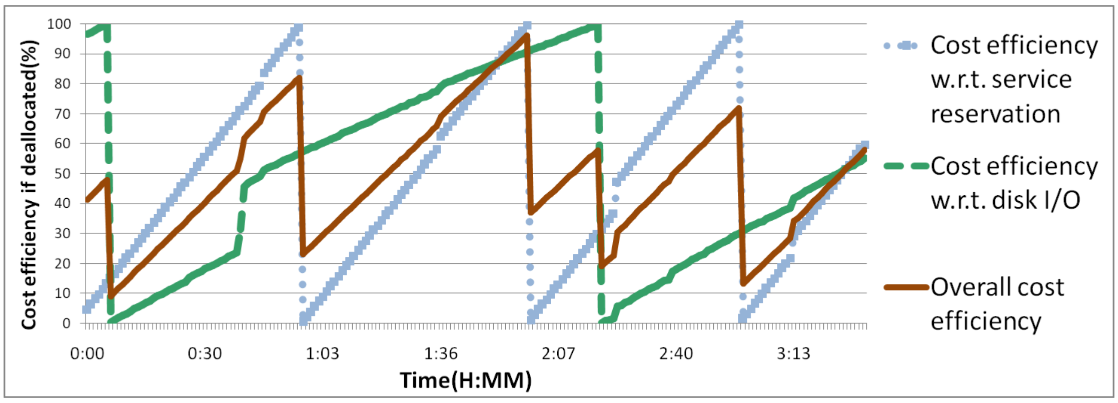


Figure 8.15: Decomposed cost efficiency of VM cloud service

second and the `Load Balancer` distributes requests in a round-robin manner, to reduce the variables influencing our results, enabling us to better evaluate and compare cost efficiency of different controllers. As the elastic unit is the `Event Processing Unit`, we continuously evaluate the cost efficiency of its instances. We start with five unit instances, and repeatedly enforce 2 scale-out followed by 2 scale-in actions, each action being enforced every 45 minutes. By performing the actions in a time interval less than 1 hour (the Reservation billing cycle of the used cloud services), we mimic normal behavior of elastic systems which can add and remove services instances anytime. At the same time we maintain an action periodicity allowing us to compare the efficiency of the scaling strategies.

What is the system's cost efficiency evolution in time? is answered by our platform by analyzing cost efficiency of separate unit instances (Figure 8.14). The analysis highlights the need for analyzing cost efficiency when scaling-in elastic systems. Initially, all 5 instances have similar cost efficiency. However, as time progresses and scale-in/out actions are enforced, the cost efficiency of the unit instances differ between each other, due to the different billing cycles of their cost elements. For improving the system's cost efficiency, it is crucial to understand how the cost efficiency of an individual system unit instance is influenced by the cost elements applicable to the cloud offered services it employs. Figure 8.15 shows for a single service unit instance its (i) VM reservation cost efficiency, (ii) disk I/O cost efficiency, and (iii) overall cost efficiency. From the figure we can notice that the disk I/O and the VM reservation cost efficiencies have different cycles, as it roughly takes 2 reservation periods (1 hour each) to complete a disk cost cycle, i.e., to generate 1GB of I/O. As cloud systems can have complex behavior, and can employ a large array of cloud services, each with its own pricing scheme, this analysis highlights the need to evaluate and consider the cost efficiency of elastic systems during their control, to increase the system's cost efficiency.

What is the system's cost efficiency under different control strategies? is answered by using our platform to evaluate the system's cost efficiency, and using it in comparing the cost efficiency of each scale-in operation for every control strategy. By integrating the cost information provided by our platform in the system's control mechanism, the developer maintains the service's cost efficiency at over 80% during enforcement of elasticity actions (Figure 8.16). From the figure we can notice that except one action, the strategy deallocating units when their cost efficiency is over 80% obtains better cost efficiency even compared to the one deallocating units at over 90% of their reservation billing cycle. This highlights that all cost elements need to be considered when scaling elastic systems, as relying only on reservation time can lead to less cost-efficient solutions. Furthermore, employing cost-agnostic strategies can lead to deallocating services which are almost unused, but paid in full, decreasing the system's cost efficiency, and increasing its cost.

Improving cost efficiency of elastic systems

While the above scenario highlights the improvement in cost efficiency of elastic systems by using our cost efficiency analysis, the employed uniform periodic control does not fully capture the dynamism of elastic systems. If the system load is highly fluctuating, by opportunistic deallocation of cloud services, controllers can decrease cost efficiency by deallocating services ahead of time, and then allocating new ones to cope with rising demand. To evaluate such scenarios, we compare the best cost agnostic (deallocating last added instance), and the best cost-based (deallocating w.r.t. cost efficiency) strategies from the previous scenario. We implement an elasticity controller which tries to enforce randomly between 1 and 3 scale-in/scale-out actions at time intervals between 30 and 60 minutes, mimicking the behavior of real systems which might scaled at different points in time, depending on requirements. Under the cost-aware strategy, when a scale-in is possible, the controller waits until a service instance has reached 80% cost efficiency, while under the cost-agnostic strategy it deallocates services as soon as possible. If after a scale-in request, a scale-out is received, the cost-aware controller will check if it's waiting to reach a certain cost efficiency to scale-in the system, and just cancel the scale-in action, instead of allocating a new service.

What is the benefit of considering cost efficiency in elasticity control? is noticeable

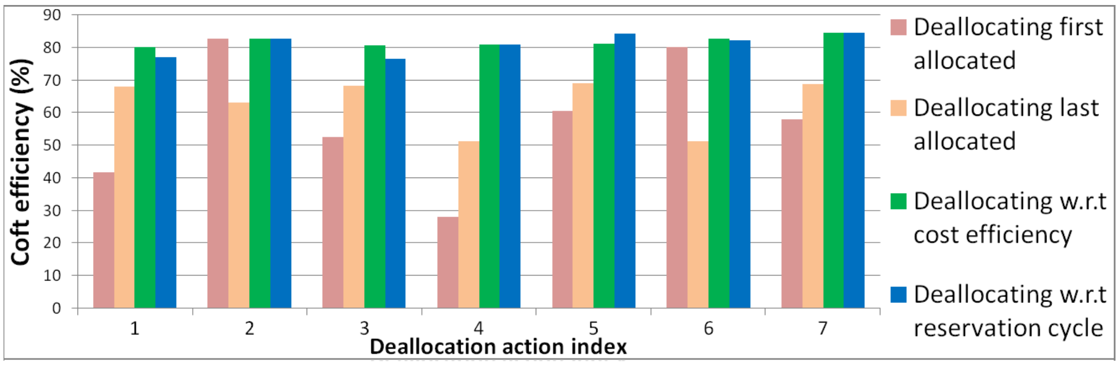


Figure 8.16: Cost efficiency of unit instance deallocated by different strategies

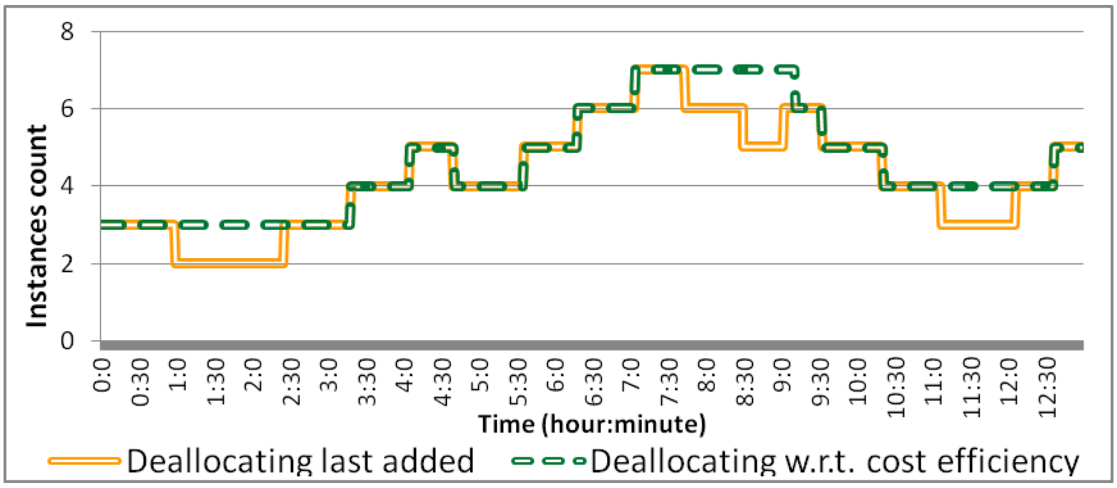


Figure 8.17: Event processing instances no. under cost-aware and cost-agnostic scalability

from Figure 8.17, depicting the number of `Event Processing` unit instances under each evaluated control strategy. We have run the scenario over 12 hours, and in three cases, the cost-based strategy avoided unnecessary scale-ins, while the cost-agnostic one deallocated and allocated back service instances. For example, between 7:30 and 9:30, the cost-based strategy scaled in just once, instead of two scale-in and one scale-out performed by the cost-agnostic one. Thus, considering cost efficiency in scaling-in elastic systems can reduce control oscillation. The improvements obtained by the controller deallocating services considering their cost efficiency are summarized in Table 8.4. Overall, the controller performed with approx. 37% less scale-in/out actions, left 8% more running VMs, all with a cost reduction of 7%. Thus, the system is overall more stable, as any scaling can introduce instability. By running more unit instances, the system has more spare resources to accommodate unexpected spikes in load. Additionally, considering the cost proportions obtained in Section 8.7.2, the cost reduction obtained in about 13 hours of operation time translates to money which can be spent to run one VM for over 3 hours and generate around 1.5 GB of I/O.

This scenario highlights that by considering cost efficiency, one can design better elasticity control mechanisms which can, at the same time, increase the performance and stability of the system, while reducing its operation cost.

8.7.3 Lessons and impact for elasticity control

We ran the evaluation scenarios repeatedly over a period of several weeks, to understand the cost issues affecting elastic systems, and learned several important lessons.

Cost behavior of elastic systems is unpredictable even in uniform conditions. In the last scenario, even with constant system load, the system's behavior influencing cost (E.g., VM lifetime, disk I/O) was not uniform. Due to cloud internal processes, some VMs took longer to allocate/deallocate than others. Additionally, due to unknown factors, the

Evaluation criteria	**Scalability control**		**Cost-aware scalability**
	cost-aware	**cost-agnostic**	**vs. cost-agnostic control**
Scaling actions	10 (6 out, 4 in)	16 (9 out, 7 in)	**37% less actions** = increased system stability
Average no. of instances	4.59	4.24	**8% more instances** = spare resources for load bursts
Total cost (units)	143	153	**7% cost savings** = 3.3 VM hours + 1.5 GB I/O

Table 8.4: Cost-aware scalability control compared to cost-agnostic scalability

VMs behavior (E.g., disk I/O) was not uniform, such as for `Instance 5` in Figure 8.14 having significantly higher cost efficiency in the first 15 minutes. To address this issue, we imposed multiple logical structures over the same virtual infrastructure, starting with the same VMs for each strategy, and only logically scaling-in the system units. Due to this behavioral uncertainty, load prediction might be an unreliable for cost evaluation. Thus, controllers should use real-time cost efficiency information in their decision processes.

Cost reduction is not the only benefit of using cost efficiency during system control. From the last scenario, we have discovered that even using simple heuristics such as waiting for the desired efficiency to deallocate services can reduce the system's cost, while also improving its performance and stability (Table 8.4).

Cost efficiency information could be used to improve the system's usage. From Figure fig:costEval:costEffectivenessVsLifetime one can see that after time 2:00, overall cost efficiency remains rather low. In such situations, knowing the actual usage from what was paid on each cost element, the system load could be re-routed as to increase usage on certain cost metrics on some unit instances, and decrease on others. Thus, the system cost efficiency would be increased by increasing the cost efficiency of each unit.

Cost information could also be used in system scale-out. Knowing the dominant cost elements (E.g., Figure 8.11) could be useful during system scale up/out for selecting cloud services with less cost over those elements, reducing if possible the cost increase while increasing performance.

8.8 Conclusions

In this chapter we have focused on aiding developers of elastic systems for public clouds to monitor their costs, and develop cost-aware scalability controllers. We have introduced a model for capturing complex pricing schemes of cloud providers, from fixed service costs to costs per multiple services. We have defined algorithms for determining the applicable costs depending on the system's configuration, and for maintaining an updated view over the usage and costs of its cloud services. We have defined a function for evaluating cost efficiency of cloud systems, used in cost-aware scalability, analyzing which system unit is cost efficient to deallocate and when. We have implemented an open source costs analysis platform for monitoring costs and analyzing cost efficiency of elastic systems.

We have evaluated our platform on a elastic cloud-based data center for IoT, deployed in Flexiant, one of the leading European public cloud providers. The evaluation has shown that cost-aware scalability can increase the performance and stability of cloud systems, while reducing their operation costs. The evaluation further highlighted that cost-agnostic scaling strategies can lead to deallocating unused services, but paid in full, actually increasing cost.

CHAPTER 9

Related work

This thesis provides support for developers and software controllers of elastic systems in designing, monitoring, analyzing, and controlling the system's elasticity. Due to the particularities of each of these phases, related work spans multiple areas, from service selection, to distributed systems monitoring, analysis, and control.

9.1 Building elastic systems

In the area of building elastic systems, which we targeted in Chapter 4, related work spans from initial system design, to modeling the cloud providers whose services will be used in building the systems, and the actual selection of cloud offered services.

Focusing on the design and cloud deployment of elastic systems, Slipstream[60] is a cloud provisioning tool enabling users to describe their systems in terms of required resource, and deploy and run them on a variety of cloud providers. Another commercial tool for deployment of cloud systems is Azure's Octopus Deploy[61], a .NET[62] oriented tool enabling developers to deploy their systems on both private or public cloud running Microsoft's Azure[63] cloud platform. Going further than single-cloud systems, Di Nitto et al. introduce the ModaClouds approach [23] for the development and operation of multi-cloud systems, providing a set of tools enabling users to describe systems using both PaaS and IaaS cloud services from multiple cloud providers. Kopp et al. introduce Winery[22], a TOSCA[82] based modeling tool for cloud systems. Winery is a tool offering an HTML5-based environment for graph-based modeling of systems, allowing the definition of reusable components and relationships between system units. Winery allows a user to specify for the system unit both software and virtual infrastructure resources

[59]The content from this chapter was partially presented in [32, 33, 1, 34, 35]
[60]`http://sixsq.com/products/slipstream.html`
[61]`http://octopusdeploy.com`
[62]`http://www.microsoft.com/net`
[63]`https://azure.microsoft.com/`

required for the unit to run. However, all the above tools require from the user a complete specification of the required cloud services. Completely describing cloud systems requires the user to have absolute knowledge over all the cloud services which can be used from multiple cloud providers. Our approach differs as we focus on reducing the complexity of this task by providing recommendations for the system configuration, based on the required system elasticity capabilities. As we focus on the service selection aspect, our approach can be integrated or used in conjunction with existing system modeling tools, as we have showcased in Chapter 4.6.2.

While the above tools enable the description of cloud systems, towards automating the process selecting cloud services suitable for individual elastic systems, one must capture information about the cloud services offered by different cloud providers. To this end, several approaches exist focusing on modeling cloud providers towards cloud services provisioning. Goncalves et al. [67] define CloudML, a modeling language for describing resources, services, and service requests in the context of multi-cloud systems. The objective of CloudML is to be vendor-neutral, providing abstractions for describing the resources and functional capabilities of cloud services, hiding their particular representation provided by each cloud provider. This provides to system developers a uniform baseline to compare cloud services offered by different cloud providers, much more difficult to do when each cloud provider exposes information in different formats. Enabling users to describe their systems and the associated cloud services, Andrikopoulos et at. [83] introduce GENTL, a topology language for describing cloud systems. Understanding that there already exist many languages for specifying cloud systems, using GENTL, the authors allow the mapping from these languages into a common model, which supports different types of annotations for enriching the topology model with additional information. Focusing on efficient policies for selecting cloud services, Villegas et al. [29] model cloud providers based on Amazon EC2, a popular commercial IaaS. Moreover, the authors assume that allocating/deallocating any cloud service incurs a certain delay, and that each such action has a certain cost associated to it. Stepping into the standards domain, there are several standards aimed at describing cloud services and their capabilities, such as the Open Cloud Computing Interface (OCCI)[64]. The OCCI core cloud representation model classifies cloud services in categories, further split in sub-classed by Kind and Mixin, which can be associated to define any Entity subclass, such as Resource or Link. Entity properties are marked as mutable or immutable, indicating if they can be changed or not. Action instances describe actions for instantiating/destroying Entity instances or changing their properties. OCCI introduces two extension mechanisms for their model, first by sub-classing Resource, Link and Action, and second, by defining custom Category, Kind or Mixin instances. An infrastructure extension is also introduced which provides Resource and Link subclasses for IaaS clouds. Another standard aiming to standardize cloud representation and interaction is the Cloud Infrastructure Management Interface (CIMI)[65]. The aim of CIMI is to provide a reference implementation model for cloud management interfaces, strictly defining any action necessary for deallocating

[64]http://occi-wg.org/

[65]http://www.iso.org/iso/catalogue_detail.htm?csnumber=66296

and allocating cloud services. However, these approaches focus on representing cloud services mostly providing virtual infrastructure services, from the system deployment perspective. We differ, as we adopt the run-time elasticity perspective, and view the service selection as an intermediary step towards ensuring the elasticity of the constructed system. To this end, we model cloud services capturing information about their available elasticity capabilities, and analyze how each service influences the system's elasticity during run-time.

Focusing on cloud service selection, Zhang et al. [61] introduce a recommender system for selecting cloud infrastructure services based on their functionality and QoS parameters. The authors employ and ontology-based mechanism for modeling cloud services, which can be manipulated through both regular expressions and SQL. The introduced approach captures a repository of available infrastructure services from different providers including compute, storage and network services, based on which a semi-automatic selection mechanism is implemented. A mathematical formulation of the cloud service provider selection problem towards maximizing selection benefits within a given budget is introduced by Chang et al. [62], focusing on probability-based selection of cloud storage services. Assuming that cloud services might fail in time, the authors introduce a solution for choosing a subset of cloud offered services from multiple cloud providers, to replicate data across multiple cloud sites, while maintaining a fix budget. Acknowledging that human decision makers are crucial in selecting the cloud services appropriate for building individual elastic systems, Wittern et al. [63] capture properties of cloud services and requirements using variability modeling, and integrate human decision-makers, towards filtering cloud services for constructing cloud systems. Cloud services are described using feature models capturing the commonalities and differences between cloud services. Considering the dynamics of cloud services, the authors use features to capture information about both the control and management capabilities exposed by cloud systems, and the particular life-cycle phases of different service types. Dastjerdi et al. [84] address the issue of service selection from a negotiation perspective, in which the cloud provider and cloud user negotiate over the service's properties. Users want lowest price with the highest availability, while cloud providers would like to sell services with highest cost and lowest performance. The authors consider that users need during their system's run-time to allocate new cloud services, to cope with spikes in demand. Thus, users are considered to have time requirements influencing their negotiation, as they need to acquire the required services within a certain time limit, after which the system needing the extra services starts to violate its operating requirements. Moving from selecting computing resources based on concrete requirements, Patiniotakis et al. [64] ranks and selects cloud services suitable for building cloud systems using a fuzzy quantification approach. The authors argue that humans are better inclined in using relative requirements, such as requiring something cheap, or expensive, instead of exact requirements over crisp concrete service cost. Thus, the authors describe the properties of quantifiable key performance indicators, which they use to sort cloud services according to specified fuzzy requirements. Moving into the platform as a service (PaaS) cloud model, Kamateri et al. [65] semantically interconnect heterogeneous PaaS offerings across different cloud providers. The authors aim to address the issue of the diversity and heterogeneity of today's existing PaaS offerings, providing support for users to migrate between PaaS providers. Increasing the level of abstraction even further, Demchenko et al. [85] leverage the everything as a service principle (XaaS), and consider that cloud systems should be composed of services. To this end, the authors introduce GEMBus, an automated services composition platform providing federated network access to distributed systems and resources towards creating service oriented architectures. Compared to the mentioned approaches, we do not focus only on initial system construction and deployment. Instead we capture and use in our service selection process information regarding the elasticity capabilities of selected services, and how these capabilities influence the run-time elasticity of the system using them.

9.2 Monitoring elastic systems

Monitoring of elastic systems, which we targeted in Chapter 5, is crucial for human and software controllers, which require relevant information to maintain their systems within certain operating parameters. As cloud computing is a branch of distributed computing, many existing approaches for monitoring distributed systems can also be applied for cloud computing scenarios.

While arguing that virtualization brings benefits in autonomic resource management, Lu et al. [86] highlight that profiling physical resource utilization information of VMs when consolidated on a single server is crucial for understanding if this collocation creates system performance issues. To this end, the authors provide a platform outputting estimates of physical resource utilization on individual VMs, to be used in smart distribution of system load. Focusing on managing large scale data centers and clouds, Wang et al. [72] introduce Monalytics, a scalable platform for data collection and aggregation. For effective management of large distributed systems (running on top of either physical or virtual computing resources), the authors collect monitoring data, and analyze it in real-time, as to identify problematic system states. Highlighting the complexity of monitoring cloud systems, the Shao et al. [87] introduce a model for run-time monitoring of cloud systems, focusing on common monitoring concerns. The authors focus on four monitoring levels, starting from infrastructure level monitoring, followed by middleware monitoring, system-level monitoring, and monitoring the systems interactions. Depending on the requirements of interested stakeholders, each stakeholder can retrieve the data captured by one of these four monitoring perspectives. Taking the system-level monitoring approach, Singh et al. [71] focus on monitoring systems distributed across different servers in a data center. Such systems consist of tiers, which in turn may also be replicated via clustering. The authors monitor the workload at each system component and tier, and use an aggregator to obtain the overall tier behavior in the case tiers are replicated, and use this information in dynamically adding/removing replicas of each tier to match the system workload. Dhingra et al. [88] argue that monitoring is crucial in building systems which must adhere to certain QoS guarantees, monitoring enabling them to know when to request for more resources, and what proportion of various physical resources are appropriate for each type of systems. To this end, the authors introduce a distributed monitoring framework, in which monitoring agents run on individual physical or virtual machines, and report monitoring information to a central collection point. Katsaros et al. [89] introduce a self-adaptive hierarchical monitoring mechanism for clouds, providing on-the-fly self-configuration in terms of monitoring frequency and monitored metrics. Further, the authors measure both QoS at system level, and resource usage of the underlying infrastructure, triggering events if monitored values exceed certain thresholds. In the context of service clouds, i.e., cloud providers offering a large array of services, Clayman et al. [90] raise the issue of monitoring when such clouds are federated. Moreover, systems running on top of such federated clouds need to properly monitored, the authors proposing Lattice, a framework for collecting, processing, and disseminating information about the network and computing resources used by such systems. An elastic monitoring framework for cloud infrastructures is presented in Konig et al.[91], based on a peer-to-peer architecture enabling the authors to monitor a diverse set of entities and metrics, spanning across all layers of a cloud stack. Furthermore, the authors consider that elastic systems can change their structure at run-time, and design an extensible query mechanism for retrieving monitored data, allowing the dynamic specification of data sources. Also dealing with dynamic infrastructures, Trihinas et al. [68] introduce JCatascopia, a tool for monitoring elastic systems, employing dynamic probe addition and removal to cope with infrastructure dynamicity. Moreover, to provide support for monitoring elasticity, monitoring probes can be activated/deactivated dynamically during system run-time, if required by elasticity controllers. Moving further into system-level monitoring, Leitner et al. [92] apply complex event processing techniques to extract system-specific performance information from system-level metrics. To this end, monitored data is expressed as event streams, the authors determining the system state using rules targeting sequences of detected events. Highlighting that in cloud environments one cannot assume the existence of online distributed monitoring nodes and reliable inter-node communication, Shicong et al. [73] present an adaptive cloud monitoring system providing information about monitoring message delay and loss. Further, the authors provide estimations on monitoring accuracy, and capture uncertainties introduced by messaging problems. Matthew et al. [93] introduce Ganglia, a scalable distributed monitoring system designed initially for high performance computing. Ganglia was designed to provide scalable monitoring of distributed systems, and has the ability of dynamically adding and removing monitored machines. This makes Ganglia applicable in cloud computing, for collecting monitoring data either from the physical cloud infrastructure, or from virtual machines. Ward et al. [94] introduce Varanus, a similar monitoring tool, designed to accommodate rapid cloud elasticity. Varanus is designed to maintain performance even during periods of high elasticity, in which resources to be monitored are added or removed at a high rate. Quoc et al. [95] monitor multi-cloud systems, introducing an approach for near real-time resource utilization monitoring, including CPU, memory, disk activity and network traffic. Alhamazani [96] highlight the need to monitor cloud systems at multiple levels of both the system and cloud provider software stacks. To this end, the authors integrate benchmarking and motoring in a framework for collecting baseline performance indicators and monitoring information about cloud systems. Cianciaruso [97] address the issue of monitoring multi-cloud systems by introducing a model-based monitoring approach. Depending on the monitored system model, the monitoring infrastructure should dynamically adapt, both in terms of monitoring frequency, and in monitored resources. Acknowledging that different cloud users can have individual monitoring concerns, Nguyen et al. [98] introduce a role-based monitoring approach. Depending on their goals, users use monitoring templates to describe their monitoring concerns. While most cloud monitoring tools focus on generic monitoring tools which can be applied to any distributed system, there also exist cloud platforms for high performance computing. These introduce particular monitoring concerns, addressed by Agelastos et al.[99]. The authors introduce a lightweight distributed metric collection service, which can be run continuously across HPC platforms, obtaining insight in the platform resource usage due to particular HPC job placements. Cloud monitoring is also the focus of many industry tools such as Nagios[66], Zabbix[67], OpenNMS[68], or Hyperic[69].

Such tools mostly focus on gathering data from the physical and virtual infrastructure, and distributing it, without correlating it with the systems running on it. We differ as we do not focus on monitoring. Instead, we rely on data from existing monitoring solutions, aggregate it according to the system's structure and enrich it. We further use the data to provide a multi-level integrated view over the behavior of individual system unit instances, and the overall behavior of system units, topologies, and the complete system. Moreover, existing work does not correlate collected data from multiple levels, and especially does not correlate data collected from multiple instances of the same system unit. Instead, we aggregate according to the system's structure the monitoring information collected from multiple levels and data sources, and enrich it with custom derived metrics. Moreover, we adopt the cloud infrastructure user perspective, assuming there is no access to the inner workings of the cloud provider. This enables our approach to be used both in private clouds, and public clouds where the cloud user only has access to his virtual infrastructure, i.e., the cloud services he requested and paid for.

9.3 Analyzing the behavior of elastic systems

Analysis of elastic systems, which we targeted in Chapter 6 and Chapter 7, is approached from two perspectives in current research: (i) system monitoring and identification of abnormal events, and (ii) determining relationships among different monitored metrics.

Dean et al. [74] use self-organizing maps (SOMs) to predict abnormal virtual machine behaviors, classifying abnormal situations by their neighborhood distance. Doelitzscher et al. [100] analyze the behavior of cloud users to discover security breaches leading to VMs being overtaken. To this end, the authors introduce an anomaly detection system for Infrastructure as a Service (IaaS) clouds, relying on neural networks to analyze

[66]`http://www.nagios.org/`
[67]`http://www.zabbix.com/`
[68]`http://www.opennms.org/`
[69]`http://www.hyperic.com/`

and learn the normal usage behavior systems running in the cloud. The introduced approach provides a means of understanding if the behavior of particular instances of system units falls within the expected behavior for those units. Analysis of cloud system behavior towards detecting security issues is also addressed by He et al. [75], the authors analyzing system monitoring using statistical anomaly detection for determining abnormal behavior system behavior. To this end, the authors capture not only VM level metrics, but also monitor and analyze the normal system behavior at the software level, e.g., resource usage for normal database operation. Moving away from security, Venzano et al. [101] evaluate the performance impact virtualization has on the network performance of systems migrated from data centers to virtualized environments, such as private clouds. In the introduced approach, the authors study and evaluate the performance of cloud systems under certain traffic patterns, determining the impact of the virtualization layer, and especially of VM collocation, on the system's network performance. Focusing on energy efficiency of cloud infrastructures, Yang et al. [102] analyze the correlations between different cloud workload patterns and the infrastructure's energy efficiency. As co-located workloads in virtualized environments have to compete for resources, and thus computation power is wasted with resource sharing mechanisms, the authors analyze different workload types influence the amount of work per Watt consumed. Moving from the virtual infrastructure view to the system view, Gullhav et al. [103] focus on evaluating distributed multi-tier systems in terms of performance and dependability. As the authors consider that multi-tier systems replicate their units according to requirements, focus on estimating the relationships between the system response time and the number of such replicas, information to be used in improving system control. As in virtualized environments the performance of individual VMs is influenced by the number of VMs located on the same physical machine, and their usage patterns, Lloyd et al. [104] focus on determining in cloud infrastructures the relationships between physical and virtual machine resource utilization, and use this information to predict the performance of systems running on top of those infrastructures. Adopting the same infrastructure provider perspective, Yiduo et al. [105] analyze the network I/O performance of cloud systems co-located on the same physical resources. The authors evaluate the impact of idle virtual machines on the virtual machines that are executing load and are running on the same physical host. Feifei et al. [106] correlate the performance of cloud systems with the energy consumption of the used computing infrastructure, enabling cloud providers to optimize the virtual machine placement for reducing overall energy consumption. Kim et al. [107] perform multi-dimensional analysis of cloud systems, determining relationships between system performance, availability, and used VM types. The relationships are integrated in a mechanism for managing elastic scientific applications running on public IaaS clouds. Panneerselvam et al. [108] focus on classifying cloud workloads depending on the types of cloud systems and their particular operations. The workload classification is applied in predicting usage of cloud environments, depending on the running systems. Due to the complexity of cloud systems, Mdhaffar et al. [109] argue that complex event processing should be used in determining if such systems behave properly or not. Addressing the dynamicity and changing requirements in cloud systems, the authors introduce a dynamic complex event processing platform, in which platform components are activated, deactivated according to requirements. Noticing that modern cloud systems have from a few components, to hundreds of interacting software components running on top of or using different types of cloud services, Singh et al. [77] focus on estimating the behavior of distributed systems when any change in the system's structure or workload takes place. To this end, the authors commonly available monitoring information, and provide a mechanism for executing workload-based "what-if" queries about the system's behavior. As scale of cloud systems continues to grow, exhibiting ever more complicated interactions between system components, Ding et al. [110] aim to help cloud providers in determining, without any prior knowledge, the number of systems and the dependency relationships between system components, information to be used in cloud management, such as system update. Relying on passive network monitoring data, the authors analyze the cloud network traffic matrix, and determine the connectivity graph used by the systems running in the cloud. Noticing that cloud systems produce a large amount of monitoring data, Xiong et al. [76] introduce vPerfGuard, a framework for system performance diagnosis which automatically discovers metrics which are most descriptive of system performance. To this end, the authors introduce a mechanism relying on statistical analysis for learning the system performance model, and analyzing the model's accuracy as the system evolves in time. Gambi et al. [111] use Kriging models to capture and predict the performance of highly dynamic elastic cloud systems, automatically adapting the model to changing system workload conditions.

With respect to the above approaches, we differ as we analyze the elasticity behavior of the system with respect to its performance and cost requirements, and not absolute metric values. This is crucial in analyzing elastic systems which scale up/down, potentially bursting in different clouds. We further analyze both direct and indirect relationships between different units and topologies, avoiding to focus on a single system unit, thus obtaining a complete view over the system's dependencies. Moreover, our work is not focusing on any specific system type, and can be applied to analyze any elastic system, providing information relevant for various types of stakeholders.

9.4 Evaluating cost and cost efficiency of elastic systems

Analyzing costs of complex elastic systems, which we targeted in Chapter 8, is required from their design, to deployment and run-time control. When considering hybrid cloud systems, understanding the cost of running them on public vs. private clouds is crucial for their deployment and run-time control. This is highlighted by Kaviani et al. [112] in system partitioning for hybrid clouds, helping developers trading off performance and cost in a hybrid cloud deployment of cloud systems, and Andrikopoulos et al. [113] focus on optimally distributing system components across cloud services in a cost efficient and flexible manner. The rationale behind the cost of Amazon EC2 spot instances is analyzed by Agmon et al. [114], analyzing the spot price histories to reverse engineer Amazon's pricing function. Sangho et al. [115] discuss cost-effective strategies for using such instances, introducing adaptive check pointing schemes to improve completion time of long running jobs. A similar approach is introduced by Jangjaimon et al.[116], applying enhanced adaptive incremental check-pointing for multi-threaded processes running in Amazon EC2 on spot priced VMs. Imai et al. [117] introduce an optimization framework for deploying MapReduce/Hadoop over multi-cloud environments, considering the virtual machine and data transfer costs. Further, complex systems such as presented by Raghavan et al. [118] providing cloud storage on top of different cloud services, require detailed cost analysis of each used service towards run-time cost optimization. Franceschelli et al. [119] introduce a tool for aiding developers of cloud systems to select suitable services, enabling them to evaluate multiple system architecture, and how different cloud services impact their performance and cost. This is further highlighted by [78] in requirements for e-science clouds. Thus, cost-benefit evaluation is necessary for understanding the behavior of complex scientific processes, due to the plethora of available cloud services and configuration options.

Cost evaluation is crucial in controlling elastic cloud systems, such as underlined by Lorido-Botran et al. [27], almost all surveyed scaling techniques considering cost in their control processes. Further, Hwang et al. [26] survey cloud scaling strategies, and underline that cloud productivity is tied to performance/cost ratio. Sharma et al. [80] focus on cost-based optimization of elastic cloud systems, considering the cost of different types of virtual resources, and of transitioning the system between different configurations, and Truong et al. [120] analyze and estimate cost of running scientific applications in the cloud, considering different execution models and service dependencies. Douglas et al. [121] estimate the cost of running scientific simulations in public clouds, leveraging the Amazon API for retrieving cost information, while Lilienthal [30] computes the optimal resources for hybrid systems running in private-public clouds. Brighen et al. [31] estimate running cost of data intensive systems in clouds, considering data query processing time, while Tsai et al. [122] address the same problem by estimating expected costs under estimated system load, and Teregowda et al. [123] port the SeerSuite search engine to cloud as a scalable system. Zhang [124] analyze cost-efficient deployment solutions for running Hadoop solutions on public clouds. The authors consider only the virtual machine cost, and evaluate deployment strategies using the same type of VM, and using different VM types. While the previous authors highlight the cost complexity of elastic systems, they do not capture and evaluate all cost elements, and do not give insight in the cost efficiency of the analyzed systems.

In [28], Mao et al. allocate virtual machines to perform given tasks in a time deadline, shutting down VMs when approaching full hour operation. Pooyan et al. [81] provision cloud services fulfilling SLA and cost constraints, terminating a cloud service if it has been running a multiple number of hours. Silva et al. [125] introduce a framework for benchmarking and scaling cloud systems for understanding their cost/performance trade off. Hwang et al. [126] focus on cost effective provisioning of cloud services, considering reserved and on-demand cost, and three cost functions: upfront fee, usage charge, and on-demand cost. Guo et al. [127] focus on cost-aware bursting of systems running in private clouds to public clouds, by proactively replicating virtualized systems, lowering the bursting time. Fernandez et al. [79] focus on cost-aware scaling of web servers in heterogeneous cloud infrastructures, executing scale-in actions by releasing resources if the system load exceeds a lower threshold, to reduce cost. In our work we provide to such control approaches detailed information about both the cost, and cost efficiency of elastic cloud systems, providing a base towards improving their run-time control.

CHAPTER 10

Conclusions and future work

In this chapter we summarize the main results of this thesis. In Section 10.1 we focus on the main outcomes of the conducted work, detailing the advancements brought over the state of the art in monitoring and analyzing elastic systems. We revisit the research questions introduced in Section 10.2 and critically analyze them in Section 10.2. Finally, in Section 10.3 we discuss ongoing trends and directions where the contributions presented in this work can act as a base for future research.

10.1 Summary of Contributions

Throughout this thesis, we have elaborated novel techniques for analyzing the aspects influencing the run-time elasticity of cloud systems. We have adopted the cloud user perspective, and aimed to support the design and run-time management of elastic systems, by providing information crucial at each of these stages.

We have first focused on reducing the complexity of building elastic systems running in the cloud. We have introduced a novel approach for analyzing and quantifying the support cloud services offer for various forms of elasticity. We have integrated our approach in a platform for recommending cloud services suitable to system's requirements over resources, performance, cost, and the envisioned system elasticity. Targeting elastic systems during their run-time, we have focused on monitoring them, taking into consideration their structure volatility. We have introduced a model for representing the run-time structure of elastic cloud systems, and a mechanism for associating monitoring information collected from various sources to the logical system structure. Using our approach, system controllers can analyze the system's behavior at multiple levels, from simple virtual infrastructure metrics, to system-level metrics composed of other simple or composite metrics. We have introduced the concepts of elasticity space and pathway to characterize the behavior in time of elastic systems. Based on our monitoring mechanism we have defined techniques for characterizing the behavior of elastic systems

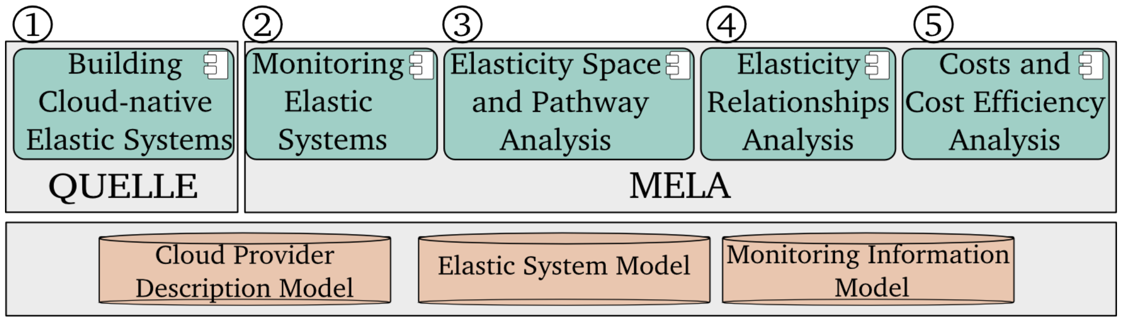

①
②
③
④
⑤
Building Cloud-native Elastic Systems
QUELLE
Monitoring Elastic Systems
Elasticity Space and Pathway Analysis
Elasticity Relationships Analysis
Costs and Cost Efficiency Analysis
MELA
Cloud Provider Description Model
Elastic System Model
Monitoring Information Model

Figure 10.1: Integrated platform for monitoring and analyzing elastic cloud systems

with respect to their requirements. Our approach provides information on how each elastic system component should be controlled, and offers insight in the system's behavior evolution, acting as a base for predicting it. We have further introduced a mechanism for determining behavioral relationships between system components, understanding how different components influence each other at run-time. Determining the relationship enables different stakeholders, from developers to elasticity controllers, to understand the run-time behavior of elastic cloud systems, and improve their control mechanisms. Providing support for cost-aware control of elastic systems, we have defined a mechanism for evaluating their cost efficiency. We have analyzed which system components are cost efficient to deallocate and when. To this end we have introduced a model for capturing complex pricing schemes of cloud providers, and defined algorithms for determining and evaluating the applicable costs depending on the system's configuration.

We have evaluated our approach in multiple scenarios involving the elastic system detailed in Section 3. We have integrated the research contributions brought by this thesis (Section 1.5) in a unified platform for monitoring and analyzing elastic cloud systems (Figure 10.1). Each research contribution was implemented on top of the existing platform version, extending it with additional functionality. The platform has two distinct components, relying on the same informational model. The first component is QUELLE (①), analyzing and quantifying the elasticity capabilities of cloud services, implementing the research contribution from Chapter 4. The second component is MELA, monitoring and analyzing elastic cloud systems. MELA provides to elasticity controllers and system developers information to be used in improving the run-time control of elastic systems. MELA contains components which work together for implementing the rest of the contributions made in this thesis. Contribution introduced in Chapter 5 is implemented by a *Monitoring* service (②). Using data obtained from the *Monitoring* service, an *Elasticity Space and Pathway Analysis* service (③) implements the contribution from Chapter 6. Based on the determined system *Elasticity Space*, an *Elasticity Relationships Analysis* service (④) implements the contribution from 7. Finally, a *Costs and Cost Efficiency Analysis* service (⑤) implements the contribution from Chapter 8 based on data obtained from the MELA *Monitoring* service. Through our evaluations, we have shown that using our approach, we reduce the complexity of selecting cloud services suitable for deploying elastic cloud systems. Using our monitoring approach, both system developers and elastic systems' controllers can obtain multi-level system behavioral information. We have further shown that our elasticity analysis approach reduces the complexity of controlling elastic systems at run-time, providing additional knowledge about the system behavior. Additionally, our cost efficiency analysis approach can increase the performance and stability of cloud systems, while reducing their operation costs.

10.2 Research Questions Revisited

The research questions introduced in Section 1.4 guided the work in this thesis. In this section, we revisit these questions and summarize how they have been answered within the context of our work, along with a discussion of the limitations of the presented solutions.

- **Question 1**: *How is the run-time elasticity of cloud systems influenced by the cloud services they use?*
  We have answered this question through our contributions from Chapter 4. In design and control of elastic systems is important to understand the support offered by cloud services for different types of elasticity. We have addressed this issue by quantifying the elasticity of cloud services. We have introduced a quantification function, and applied it to recommend cloud services based on their elasticity support. In the quantification function we have considered mandatory service configurations as reducing the overall elasticity of the cloud service, and optional configurations as increasing their elasticity. We have integrated the function in a platform for recommending suitable cloud services for building elastic systems, depending on their requirements and designed elasticity capabilities. The described quantification function only discriminates between two configuration categories: mandatory (i.e., service MUST use option), and optional (service CAN use option). However, for complex elastic systems more complex quantification functions might need to be defined, such as weighting the service's configuration options depending on system requirements. The implemented platform for recommending system configurations evaluates each system component/unit independently, and generates a set of suitable cloud services for it. This leads to a need to further process and aggregate the solutions for each system component, to achieve a deployable system. Further work is required on analyzing and processing the generated solutions, to generate a complete system description, such as understanding which recommended cloud services can be shared among system components.

- **Question 2**: *How can elastic systems be monitored and analyzed, considering their complexity and dynamic run-time structure?*
  We have answered this question through our contributions from Chapter 5. Monitoring elastic cloud systems is non-trivial due to both their software complexity, and dynamic run-time structure. We have addressed this issue by introducing a mechanism for linking the system's run-time structure and used cloud services

to its design-time structure. The mechanism structures monitoring information according to user requirements, in multiple levels of detail, from information about cloud services, to information describing system components, and the entire system behavior. However, our approach assumes that the system has a shared-nothing architecture, one cloud service being used by a single system component. In practice, while performance or security sensitive systems would adopt this shared-nothing approach, less sensitive systems might not. In this case, finer-grained monitoring is required to differentiate the cloud service usage done by each system component relying on it. As the focus of the approach was on structuring collected monitoring data, improving existent data collection mechanisms was out of the scope of this thesis.

- **Question 3**: *How can the behavior of elastic systems be characterized towards aiding in their run-time control?*
  We have answered this question through our contributions from Chapters 6 and 7. We have first focused on dealing with incomplete system requirements by determining missing requirements using novel concept of elasticity space and boundary. The space is designed to characterize the behavior of all system components when all system requirements are respected. In the provided prototype we record as space boundaries the minimum and maximum values for each system metric, when all system requirements are respected. However, we do not record the boundaries' history, losing data about their evolution in time. This can lead to isolated system behaviors, such as spikes, to have a greater influence on the system's elasticity boundaries than if we have used a history-aware function. We also provide a function and prototype for computing the elasticity pathway of cloud systems. In the provided prototype we compute behavioral clusters, but we do not capture the transition between one behavioral cluster to another, such as from low to high system performance. While the presented approach provides information crucial in reactively controlling elastic systems, addressing the previous issues would improve predictive control mechanisms even more. In Chapter 7 we have focused on determining behavioral relationships between components of elastic systems. Relationships describe the impact an elasticity action enforced on particular component or change in system load has on the rest of the system. To this end we define the concept of elasticity energy based on the system's behavior with respect to its elasticity boundaries. However, in our prototype we only determine linear relationships. While knowing such relationships enables predictive elasticity control, behavioral relationships can be much more complex. In this case, more work is required on mechanisms for determining other types of relationships, providing a more detailed view over the behavior of elastic systems.

- **Question 4**: *How can elastic systems running in public clouds be controlled in a cost efficient manner?*
  We have answered this question through our contributions from Chapter 8. Cloud services are provided under different pricing schemes and billing options. Understanding when it is cost efficient to deallocate such services is important for elasticity controllers. To address this issue, we have captured complex pricing schemes of cloud providers, and have used them to recommend to elasticity controllers when and what services can be deallocated. We have defined a cost-weighted function evaluating how much a cloud service has used from what was paid for. We have shown that our approach improves not only the cost efficiency of elastic systems, but also their performance and stability. In our prototype we evaluate the cost efficiency of deallocating cloud services based on desired cost efficiency. However, due to the complexity of cloud cost schemes and the behavior of elastic systems, required cost efficiency levels might be require an unattainable. This can lead to a controller using our system to wait indefinitely for the desired cost efficiency before deallocating services. To this end, more work is needed on analyzing and characterizing the historical evolution of the cost efficiency of cloud systems, understanding the possible efficiency values. As we have focused on providing a base platform for building cost-aware elasticity controllers, historical costs analysis for requirements validation was out of the scope of this thesis.

10.3 Future Work

In this thesis we have presented a set of concepts and techniques to enable and improve run-time elasticity control of elastic cloud systems. However, based on the discussion in Section 10.2, it is apparent that a number of important challenges were out of the scope of this thesis. In the following, we outline some challenges and possibilities for future research.

- Considering today's computing state of the art, a new research direction is monitoring and analyzing elastic cyber-physical systems. Such systems have both components deployed on physical entities in the real world, and components running in clouds or data centers. Cyber-physical systems would benefit from our monitoring approach, capable of collecting data from multiple sources, and aggregating it to obtain higher level behavioral information. However, the heterogeneity of cyber-physical systems implies different system components might be monitored using different mechanisms, at different rates. Future work should address this issue through adaptive monitoring, considering the particularities of each monitored system component. This raises new research questions on how to monitor and analyze cyber-physical system considering their particularities. Focus should be placed on designing adaptive monitoring solutions, considering the particular concerns of such systems, such as energy efficiency, security, or physical limitations.

- As elastic computing is a relatively new research area, there is no comprehensive taxonomy on elasticity capabilities, and no formal understanding when one capability is better than another. This opens a new research direction in analyzing elasticity capabilities of cloud systems, which we consider crucial for increasing the adoption of elastic systems. In Chapter 4 we have provided a base for aiding cloud users to select cloud services depending on their support for certain system elasticity, such as cost or performance elasticity. A logical continuation of this work is focusing more on the elastic system to be designed, and analyzing its particular elasticity capabilities. Developers of elastic systems would benefit from understanding if certain designed elasticity capabilities are useful or not for their system. Moreover, it is expected that certain capabilities might be applicable only on some cloud providers. To provide complete support for designing elastic systems, new approaches should be investigated for analyzing them and their elasticity capabilities at every stage of their development process.

- Analyzing health of elastic systems is a research direction crucial in system management and control. Health analysis should focus only on determining failures of the used hardware or software, but also on verifying system health after enforcement of elasticity capabilities. Further, health is not always a true/false property. It is expected that future work will analyze degraded behavior of individual system component instances and unstable performance triggered by enforcement of elasticity capabilities. In our approach we have focused on performance and cost analysis, not considering health aspects influencing elastic systems. The complexity of elastic cloud services, both in terms of software stack, and elasticity control, increases the chances of system failures or sub-optimal behavior. Future work must focus on better capturing the relationships in time between the behavior of system components over their health, performance, and cost.

Bibliography

[1] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, "Mela: elasticity analytics for cloud services," *International Journal of Big Data Intelligence*, vol. 2, no. 1, pp. 45–62, 2015.

[2] "Architecting for the cloud: Best practices," Tech. Rep., 2011. [Online]. Available: http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf

[3] *Patterns: Service-Oriented Architecture and Web Services.* IBM Redbooks, 2004. [Online]. Available: http://www.redbooks.ibm.com/abstracts/sg246303.html

[4] "An architectural blueprint for autonomic computing," IBM, Tech. Rep., Jun. 2005. [Online]. Available: http://www-03.ibm.com/autonomic/pdfs/AC%20Blueprint%20White%20Paper%20V7.pdf

[5] G. Copil, D. Moldovan, H.-L. Truong, and S. Dustdar, "Multi-level elasticity control of cloud services," in *Service-Oriented Computing*, ser. Lecture Notes in Computer Science, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds. Springer Berlin Heidelberg, 2013, vol. 8274, pp. 429–436.

[6] B. Suleiman, S. Sakr, R. Jeffery, and A. Liu, "On understanding the economics and elasticity challenges of deploying business applications on public cloud infrastructure," *Journal of Internet Services and Applications*, pp. 173–193, 2011.

[7] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the art of virtualization," in *ACM Symposium on Operating Systems Principles (SOSP).* New York, NY, USA: ACM, 2003, pp. 164–177.

[8] V. Soundararajan and J. M. Anderson, "The impact of management operations on the virtualized datacenter," in *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA).* New York, NY, USA: ACM, 2010, pp. 326–337.

[9] B. Gomolski, "U.s. it spending and staffing survey," Gartner Research, Tech. Rep., 2005. [Online]. Available: https://www.gartner.com/doc/486624/it-spending-staffing-survey-

[10] B. Childers, “Running ubuntu 9.10 under amazon’s elastic cloud,” *Linux J.*, vol. 2010, no. 191, Mar. 2010. [Online]. Available: http://www.linuxjournal.com/magazine/running-ubuntu-910-under-amazons-elastic-cloud

[11] P. Mell and T. Grance, “The nist definition of cloud computing,” *National Institute of Standards and Technology (NIST), US*, 2009.

[12] L. M. Vaquero, L. Rodero-Merino, J. Caceres, and M. Lindner, “A break in the clouds: Towards a cloud definition,” *SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Dec. 2008.

[13] “Cloud computing - the business perspective,” *Decision Support Systems*, vol. 51, no. 1, pp. 176 – 189, 2011.

[14] M. Klems, J. Nimis, and S. Tai, “Do clouds compute? a framework for estimating the value of cloud computing,” in *Designing E-Business Systems. Markets, Services, and Networks*, ser. Lecture Notes in Business Information Processing. Springer Berlin Heidelberg, 2009, vol. 22, pp. 110–123.

[15] S. Dustdar, Y. Guo, B. Satzger, and H. L. Truong, “Principles of elastic processes,” *IEEE Computing*, no. 5, pp. 66–71, 2011.

[16] I. Konstantinou, E. Angelou, D. Tsoumakos, C. Boumpouka, N. Koziris, and S. Sioutas, “Tiramola: elastic nosql provisioning through a cloud management platform,” in *International Conference on Management of Data (SIGMOD).* ACM, 2012, pp. 725–728.

[17] Z. Gong, X. Gu, and J. Wilkes, “Press: Predictive elastic resource scaling for cloud systems,” in *Network and Service Management (CNSM), 2010 International Conference on*, Oct 2010, pp. 9–16.

[18] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, “Dynamically scaling applications in the cloud,” *SIGCOMM Computer Communication review*, vol. 41, no. 1, pp. 45–52, Jan. 2011.

[19] J. Idziorek, “Discrete event simulation model for analysis of horizontal scaling in the cloud computing model,” in *Winter Simulation Conference (WSC)*, Dec 2010, pp. 3004–3014.

[20] D. Merkel, “Docker: Lightweight linux containers for consistent development and deployment,” *Linux J.*, vol. 2014, no. 239, Mar. 2014. [Online]. Available: http://dl.acm.org/citation.cfm?id=2600239.2600241

[21] D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, Sept 2014.

[22] O. Kopp, T. Binz, U. Breitenbücher, and F. Leymann, “Winery - a modeling tool for tosca-based cloud applications,” in *International Conference on Service Oriented Computing (ICSOC)*, S. Basu, C. Pautasso, L. Zhang, and X. Fu, Eds. Springer Berlin Heidelberg, 2013, vol. 8274, pp. 700–704.

[23] E. Di Nitto, M. Almeida da Silva, D. Ardagna, G. Casale, C. Dorin Craciun, N. Ferry, V. Muntes, and A. Solberg, “Supporting the development and operation of multi-cloud applications: The modaclouds approach,” in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, Sept 2013, pp. 417–423.

[24] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes, “Cloudscale: Elastic resource scaling for multi-tenant cloud systems,” in *ACM Symposium on Cloud Computing (SOCC)*. New York, NY, USA: ACM, 2011, pp. 5:1–5:14.

[25] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, “Automated control in cloud computing: Challenges and opportunities,” in *Workshop on Automated Control for Datacenters and Clouds 9ACDC)*. New York, NY, USA: ACM, 2009, pp. 13–18. [Online]. Available: http://doi.acm.org/10.1145/1555271.1555275

[26] K. Hwang, X. Bai, Y. Shi, M. Li, W. Chen, and Y. Wu, “Cloud performance modeling and benchmark evaluation of elastic scaling strategies,” *IEEE Transactions on Parallel and Distributed Systems*, vol. PP, no. 99, pp. 1–1, 2015.

[27] T. Lorido-Botran, J. Miguel-Alonso, and J. Lozano, “A review of auto-scaling techniques for elastic applications in cloud environments,” *Journal of Grid Computing*, vol. 12, no. 4, pp. 559–592, 2014.

[28] M. Mao, J. Li, and M. Humphrey, “Cloud auto-scaling with deadline and budget constraints,” in *International Conference on Grid Computing (GRID)*. IEEE/ACM, Oct 2010, pp. 41–48.

[29] D. Villegas, A. Antoniou, S. Sadjadi, and A. Iosup, “An analysis of provisioning and allocation policies for infrastructure-as-a-service clouds,” in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE/ACM, 2012, pp. 612–619.

[30] M. Lilienthal, “A decision support model for cloud bursting,” *Business & Information Systems Engineering*, vol. 5, no. 2, pp. 71–81, 2013.

[31] A. Brighen, L. Bellatreche, H. Slimani, and Z. Faget, “An economical query cost model in the cloud,” in *International Conference Database Systems for Advanced Applications (DASFAA)*. Springer Berlin Heidelberg, 2013, pp. 16–30.

[32] D. Moldovan, G. Copil, H.-L. Truong, and S. Dustdar, “Quelle - a framework for accelerating the development of elastic systems,” in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science, M. Villari, W. Zimmermann, and K.-K. Lau, Eds. Springer Berlin Heidelberg, 2014, vol. 8745, pp. 93–107.

[33] ——, “Mela: Monitoring and analyzing elasticity of cloud services,” in *International Conference on Cloud Computing Technology and Science (CloudCom).* IEEE, 2013, pp. 80–87.

[34] ——, “On analyzing elasticity relationships of cloud services,” in *International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2014, pp. 447–454.

[35] D. Moldovan, H.-L. Truong, and S. Dustdar, “Cost-aware scalability of applications in public clouds,” in *International Conference on Cloud Engineering (IC2E).* IEEE, 2016 accepted.

[36] K. A. Scarfone, M. P. Souppaya, and P. Hoffman, “Sp 800-125. guide to security for full virtualization technologies,” Gaithersburg, MD, United States, Tech. Rep., 2011. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-125/SP800-125-final.pdf

[37] “Cloud Computing Report, EU ,” Tech. Rep., 2010. [Online]. Available: http://cordis.europa.eu/fp7/ict/ssai/docs/cloud-report-final.pdf

[38] “Understanding full virtualization, paravirtualization, and hardware assist,” Tech. Rep. [Online]. Available: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf

[39] B. P. Tholeti, “Hypervisors, virtualization, and the cloud: Learn about hypervisors, system virtualization, and how it works in a cloud environment,” Tech. Rep., September 2011. [Online]. Available: http://www.ibm.com/developerworks/cloud/library/cl-hypervisorcompare/cl-hypervisorcompare-pdf.pdf

[40] B. Venners, *Inside the Java Virtual Machine.* New York, NY, USA: McGraw-Hill, Inc., 1996.

[41] A. N. Mian, A. Mamoon, R. Khan, and A. Anjum, “Effects of virtualization on network and processor performance using open vswitch and xen server,” in *International Conference on Utility and Cloud Computing (UCC).* Washington, DC, USA: IEEE Computer Society, 2014, pp. 762–767.

[42] P. Varman and J. Wang, “Storage and i/o virtualization, performance, energy, evaluation and dependability (speed08),” *SIGOPS Oper. Syst. Rev.*, vol. 42, no. 6, pp. 1–2, Oct. 2008.

[43] T. J. Brandon Chavis, “Running containers in the cloud,” Tech. Rep., 2015. [Online]. Available: http://d0.awsstatic.com/whitepapers/docker-on-aws.pdf

[44] K. Francis and P. Richardson, “Green maturity model for virtualization,” Microsoft, Tech. Rep., January 2009. [Online]. Available: http://msdn.microsoft.com/en-us/library/dd393310.aspx

[45] P. Mell and T. Grance, "The NIST Definition of Cloud Computing," http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, Tech. Rep., 2011.

[46] "Oracle jd edwards cloud computing: Choosing a deployment strategy that fits," Oracle, Tech. Rep., Oct. 2012. [Online]. Available: http://www.oracle.com/us/products/applications/jd-edwards-enterpriseone/jde-cloud-computing-wp-1851596.pdf

[47] M. D. Hill, "What is scalability?" *SIGARCH Comput. Archit. News*, vol. 18, no. 4, pp. 18–21, Dec. 1990. [Online]. Available: http://doi.acm.org/10.1145/121973.121975

[48] J. Hoskins, J. Young, and J. Fletcher, *Exploring IBM's New Age Mainframes*, 5th ed. Gulf Breeze, FL, USA: Maximum Press, 1997.

[49] M. Michael, J. Moreira, D. Shiloach, and R. Wisniewski, "Scale-up x scale-out: A case study using nutch/lucene," in *Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International*, March 2007, pp. 1–8.

[50] C. Fehling, F. Leymann, R. Retter, W. Schupeck, and P. Arbitter, "Cloud application architecture patterns," in *Cloud Computing Patterns.* Springer Vienna, 2014, pp. 151–238.

[51] L. Treloar, *The Physics of Rubber Elasticity.* Oxford: Clarendon Press, 1975.

[52] J. C. Cooper, "Price elasticity of demand for crude oil: estimates for 23 countries," *OPEC Review*, vol. 27, no. 1, pp. 1–8, 2003.

[53] "Scalability and elasticity for virtual application patterns in ibm pureapplication system," Tech. Rep., 2013. [Online]. Available: http://www.ibm.com/developerworks/websphere/techjournal/1309_tost/1309_tost-pdf.pdf

[54] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, "Deconstructing amazon ec2 spot instance pricing," *ACM Trans. Econ. Comput.*, vol. 1, no. 3, pp. 16:1–16:20, Sep. 2013.

[55] "Introduction to aws economics: Reducing costs and complexity," Tech. Rep., 2015. [Online]. Available: http://d0.awsstatic.com/whitepapers/introduction-to-aws-cloud-economics-final.pdf

[56] M. P. Papazoglou, P. Traverso, S. Dustdar, and F. Leymann, "Service-oriented computing: State of the art and research challenges," *Computer*, vol. 40, no. 11, pp. 38–45, 2007.

[57] S. Newman, *Building Microservices. Designing Fine-Grained Systems.* O'Reilly Media, Inc., 2015.

[58] J. Thones, "Microservices," *Software, IEEE*, vol. 32, no. 1, pp. 116–116, Jan 2015.

[59] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003.

[60] S. Tai, P. Leitner, and S. Dustdar, "Design by units: Abstractions for human and compute resources for elastic systems," *IEEE Internet Computing*, vol. 16, no. 4, pp. 84–88, 2012.

[61] M. Zhang, R. Ranjan, S. Nepal, M. Menzel, and A. Haller, "A declarative recommender system for cloud infrastructure services selection," in *International Conference on Economics of Grids, Clouds, Systems, and Services (GECON)*. Springer Berlin Heidelberg, 2012, pp. 102–113.

[62] C.-W. Chang, P. Liu, and J.-J. Wu, "Probability-based cloud storage providers selection algorithms with maximum availability," in *2012 41st International Conference on Parallel Processing (ICPP)*, 2012, pp. 199–208.

[63] E. Wittern, J. Kuhlenkamp, and M. Menzel, "Cloud service selection based on variability modeling," in *International Conference on Service-Oriented Computing (ICSOC)*. Springer Berlin Heidelberg, 2012, pp. 127–141.

[64] I. Patiniotakis, S. Rizou, Y. Verginadis, and G. Mentzas, "Managing imprecise criteria in cloud service ranking with a fuzzy multi-criteria decision making method," in *European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer Berlin Heidelberg, 2013, vol. 8135, pp. 34–48.

[65] E. Kamateri, N. Loutas, D. Zeginis, J. Ahtes, F. D'Andria, S. Bocconi, P. Gouvas, G. Ledakis, F. Ravagli, O. Lobunets, and K. Tarabanis, "Cloud4soa: A semantic-interoperability paas solution for multi-cloud platform management and portability," in *European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer Berlin Heidelberg, 2013, pp. 64–78.

[66] C. Sofokleous, N. Loulloudes, D. Trihinas, G. Pallis, and M. Dikaiakos, "c-eclipse: An open-source management framework for cloud applications," in *International Conference on Parallel Processing (Euro-Par)*. Springer International Publishing, 2014, pp. 38–49.

[67] G. Goncalves, P. Endo, M. Santos, D. Sadok, J. Kelner, B. Melander, and J.-E. Mangs, "Cloudml: An integrated language for resource, service and request description for d-clouds," in *International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2011, pp. 399–406.

[68] D. Trihinas, G. Pallis, and M. Dikaiakos, "Jcatascopia: Monitoring elastically adaptive applications in the cloud," in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE/ACM, May 2014, pp. 226–235.

[69] M. Mao and M. Humphrey, "Scaling and scheduling to maximize application performance within budget constraints in cloud workflows," in *International Symposium on Parallel Distributed Processing (IPDPS)*, May 2013, pp. 67–78.

[70] D.-H. Le, H.-L. Truong, G. Copil, S. Nastic, and S. Dustdar, "Salsa: A framework for dynamic configuration of cloud services," in *International Conference on Cloud Computing Technology and Science (CloudCom)*, Dec 2014, pp. 146–153.

[71] R. Singh, U. Sharma, E. Cecchet, and P. Shenoy, "Autonomic mix-aware provisioning for non-stationary data center workloads," in *International Conference on Autonomic Computing (ICAC)*, 2010, pp. 21–30.

[72] C. Wang, K. Schwan, V. Talwar, G. Eisenhauer, L. Hu, and M. Wolf, "A flexible architecture integrating monitoring and analytics for managing large-scale data centers," in *International Conference on Autonomic Computing (ICAC)*, 2011, pp. 141–150.

[73] S. Meng, A. K. Iyengar, I. Rouvellou, L. Liu, K. Lee, B. Palanisamy, and Y. Tang, "Reliable state monitoring in cloud datacenters," in *International Conference on Cloud Computing Technology and Science (CLOUD)*. IEEE, 2012, pp. 951–958.

[74] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in *International Conference on Autonomic Computing (ICAC)*. ACM, 2012, pp. 191–200.

[75] S. He, M. Ghanem, L. Guo, and Y. Guo, "Cloud resource monitoring for intrusion detection," in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 2, Dec 2013, pp. 281–284.

[76] P. Xiong, C. Pu, X. Zhu, and R. Griffith, "vperfguard: An automated model-driven framework for application performance diagnosis in consolidated cloud environments," in *International Conference on Performance Engineering (ICPE)*. New York, NY, USA: ACM, 2013, pp. 271–282.

[77] R. Singh, P. Shenoy, M. Natu, V. Sadaphal, and H. Vin, "Analytical modeling for what-if analysis in complex cloud computing applications," *SIGMETRICS Performance Evaluation Review*, vol. 40, no. 4, pp. 53–62, Apr. 2013.

[78] L. Ramakrishnan, K. R. Jackson, S. Canon, S. Cholia, and J. Shalf, "Defining future platform requirements for e-science clouds," in *Symposium on Cloud Computing (SOCC)*. ACM, 2010, pp. 101–106.

[79] H. Fernandez, G. Pierre, and T. Kielmann, "Autoscaling web applications in heterogeneous cloud infrastructures," in *International Conference on Cloud Engineering (IC2E)*. IEEE, March 2014, pp. 195–204.

[80] U. Sharma, P. Shenoy, S. Sahu, and A. Shaikh, “A cost-aware elasticity provisioning system for the cloud,” in *International Conference on Distributed Computing Systems (ICDCS)*. IEEE Computer Society, 2011, pp. 559–570.

[81] P. Jamshidi, A. Ahmad, and C. Pahl, “Autonomic resource provisioning for cloud-based software,” in *International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS)*. ACM, 2014, pp. 95–104.

[82] O. T. Committee, “Oasis topology and orchestration specification for cloud applications,” OASIS, Tech. Rep., 2010. [Online]. Available: http://docs.oasis-open.org/tosca/TOSCA/v1.0/cs01/TOSCA-v1.0-cs01.pdf

[83] V. Andrikopoulos, A. Reuter, S. Gómez Sáez, and F. Leymann, “A gentl approach for cloud application topologies,” in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2014, vol. 8745, pp. 148–159.

[84] A. Dastjerdi and R. Buyya, “An autonomous reliability-aware negotiation strategy for cloud computing environments,” in *International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE/ACM, 2012, pp. 284–291.

[85] Y. Demchenko, C. Ngo, P. Martínez-Julia, E. Torroglosa, M. Grammatikou, J. Jofre, S. Gheorghiu, J. Garcia-Espin, A. Perez-Morales, and C. Laat, “Gembus based services composition platform for cloud paas,” in *European Conference on Service-Oriented and Cloud Computing (ESOCC)*. Springer Berlin Heidelberg, 2012, pp. 32–47.

[86] L. Lu, H. Zhang, G. Jiang, H. Chen, K. Yoshihira, and E. Smirni, “Untangling mixed information to calibrate resource utilization in virtual machines,” in *International Conference on Autonomic Computing (ICAC)*, 2011, pp. 151–160.

[87] J. Shao, H. Wei, Q. Wang, and H. Mei, “A runtime model based monitoring approach for cloud,” in *International Conference on Cloud Computing (CLOUD)*, July 2010, pp. 313–320.

[88] M. Dhingra, J. Lakshmi, and S. K. Nandy, “Resource usage monitoring in clouds,” in *International Conference on Grid Computing (GRID)*. ACM/IEEE, 2012, pp. 184–191.

[89] G. Katsaros, G. Kousiouris, S. V. Gogouvitis, D. Kyriazis, A. Menychtas, and T. Varvarigou, “A self-adaptive hierarchical monitoring mechanism for clouds,” *Journal of Systems and Software*, vol. 85, no. 5, pp. 1029 – 1041, 2012.

[90] S. Clayman, A. Galis, C. Chapman, G. Toffetti, L. Rodero-Merino, L. M. Vaquero, K. Nagin, and B. Rochwerger, “Monitoring service clouds in the future internet,” in *Future Internet Assembly*, 2010, pp. 115–126.

[91] B. Konig, J. Alcaraz Calero, and J. Kirschnick, “Elastic monitoring framework for cloud infrastructures,” *IET Communications*, vol. 6, no. 10, pp. 1306–1315, 2012.

[92] P. Leitner, C. Inzinger, W. Hummer, B. Satzger, and S. Dustdar, “Application-level performance monitoring of cloud services based on the complex event processing paradigm,” in *International Conference on Service-Oriented Computing and Applications (SOCA)*, Dec 2012, pp. 1–8.

[93] M. L. Massie, B. N. Chun, and D. E. Culler, “The ganglia distributed monitoring system: design, implementation, and experience,” *Parallel Computing*, vol. 30, no. 7, pp. 817 – 840, 2004.

[94] J. S. Ward and A. Barker, “Self managing monitoring for highly elastic large scale cloud deployments,” in *International Workshop on Data Intensive Distributed Computing (DIDC)*. New York, NY, USA: ACM, 2014, pp. 3–10.

[95] D. L. Quoc, L. Yazdanov, and C. Fetzer, “Dolen: User-side multi-cloud application monitoring,” in *International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE, Aug 2014, pp. 76–81.

[96] K. Alhamazani, R. Ranjan, P. Jayaraman, K. Mitra, F. Rabhi, D. Georgakopoulos, and L. Wang, “Cross-layer multi-cloud real-time application qos monitoring and benchmarking as-a-service framework,” no. 99, pp. 1–1, 2015.

[97] L. Cianciaruso, F. Di Forenza, E. Di Nitto, M. Miglierina, N. Ferry, and A. Solberg, “Using models at runtime to support adaptable monitoring of multi-clouds applications,” in *International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, 2014, pp. 401–408.

[98] T. A. B. Nguyen, M. Siebenhaar, R. Hans, and R. Steinmetz, “Role-based templates for cloud monitoring,” in *International Conference on Utility and Cloud Computing (UCC)*. IEEE/ACM, Dec 2014, pp. 242–250.

[99] A. Agelastos, B. Allan, J. Brandt, P. Cassella, J. Enos, J. Fullop, A. Gentile, S. Monk, N. Naksinehaboon, J. Ogden, M. Rajan, M. Showerman, J. Stevenson, N. Taerat, and T. Tucker, “The lightweight distributed metric service: A scalable infrastructure for continuous monitoring of large scale computing systems and applications,” in *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2014, pp. 154–165.

[100] F. Doelitzscher, M. Knahl, C. Reich, and N. Clarke, “Anomaly detection in iaas clouds,” in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 1, Dec 2013, pp. 387–394.

[101] D. Venzano and P. Michiardi, “A measurement study of data-intensive network traffic patterns in a private cloud,” in *Utility and Cloud Computing (UCC), 2013 IEEE/ACM 6th International Conference on*, Dec 2013, pp. 476–481.

[102] R. Yang, I. Moreno, J. Xu, and T. Wo, “An analysis of performance interference effects on energy-efficiency of virtualized cloud environments,” in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 1, Dec 2013, pp. 112–119.

[103] A. Gullhav, B. Nygreen, and P. Heegaard, “Approximating the response time distribution of fault-tolerant multi-tier cloud services,” in *IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, Dec 2013, pp. 287–291.

[104] W. Lloyd, S. Pallickara, O. David, J. Lyon, M. Arabi, and K. Rojas, “Performance modeling to support multi-tier application deployment to infrastructure-as-a-service clouds,” in *International Conference on Utility and Cloud Computing (UCC)*. IEEE, Nov 2012, pp. 73–80.

[105] Y. Mei, L. Liu, X. Pu, and S. Sivathanu, “Performance measurements and analysis of network i/o applications in virtualized cloud,” in *International Conference on Cloud Computing (CLOUD)*. IEEE, July 2010, pp. 59–66.

[106] F. Chen, J. Grundy, J.-G. Schneider, Y. Yang, and Q. He, “Stresscloud: A tool for analysing performance and energy consumption of cloud applications,” in *International Conference on Software Engineering (ICSE)*, vol. 2. IEEE/ACM, May 2015, pp. 721–724.

[107] I. K. Kim, J. Steele, Y. Qi, and M. Humphrey, “Comprehensive elastic resource management to ensure predictable performance for scientific applications on public iaas clouds,” in *International Conference on Utility and Cloud Computing (UCC)*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 355–362.

[108] J. Panneerselvam, L. Liu, N. Antonopoulos, and Y. Bo, “Workload analysis for the scope of user demand prediction model evaluations in cloud environments,” in *International Conference on Utility and Cloud Computing (UCC)*. Washington, DC, USA: IEEE Computer Society, 2014, pp. 883–889.

[109] A. Mdhaffar, R. Ben Halima, M. Jmaiel, and B. Freisleben, “A dynamic complex event processing architecture for cloud monitoring and analysis,” in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, vol. 2, Dec 2013, pp. 270–275.

[110] M. Ding, V. Singh, Y. Zhang, and G. Jiang, “Application dependency discovery using matrix factorization,” in *International Workshop on Quality of Service (IWQoS)*, June 2012, pp. 1–4.

[111] A. Gambi, G. Toffetti, C. Pautasso, and M. Pezze, “Kriging controllers for cloud applications,” in *IEEE Internet Computing*, vol. 17, no. 4. Los Alamitos, CA, USA: IEEE Computer Society, 2013, pp. 40–47.

[112] N. Kaviani, E. Wohlstadter, and R. Lea, “Cross-tier application and data partitioning of web applications for hybrid cloud deployment,” in *International Middleware Conference (Middleware).* Springer Berlin Heidelberg, 2013, pp. 226–246.

[113] V. Andrikopoulos, S. Gómez Sáez, F. Leymann, and J. Wettinger, “Optimal distribution of applications in the cloud,” in *International Conference on Advanced Information Systems Engineering (CAiSE).* Springer International Publishing, 2014.

[114] O. Agmon Ben-Yehuda, M. Ben-Yehuda, A. Schuster, and D. Tsafrir, “Deconstructing amazon ec2 spot instance pricing,” in *International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2011, pp. 304–311.

[115] S. Yi, D. Kondo, and A. Andrzejak, “Reducing costs of spot instances via checkpointing in the amazon elastic compute cloud,” in *International Conference on Cloud Computing (CLOUD).* IEEE, July 2010, pp. 236–243.

[116] I. Jangjaimon and N.-F. Tzeng, “Effective cost reduction for elastic clouds under spot instance pricing through adaptive checkpointing,” *IEEE Transactions on Computers*, vol. 64, no. 2, pp. 396–409, Feb 2015.

[117] S. Imai, S. Patterson, and C. Varela, “Cost-efficient high-performance internet-scale data analytics over multi-cloud environments,” in *International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, May 2015, pp. 793–796.

[118] A. Raghavan, A. Chandra, and J. B. Weissman, “Tiera: Towards flexible multi-tiered cloud storage instances,” in *International Middleware Conference (Middleware).* ACM, 2014, pp. 1–12.

[119] D. Franceschelli, D. Ardagna, M. Ciavotta, and E. Di Nitto, “Space4cloud: A tool for system performance and cost evaluation of cloud systems,” in *International Workshop on Multi-cloud Applications and Federated Clouds (MultiCloud).* New York, NY, USA: ACM, 2013, pp. 27–34.

[120] H. L. Truong and S. Dustdar, “Composable cost estimation and monitoring for computational applications in cloud computing environments,” in *International Conference on Computational Science (ICCS).* Elsevier, 2010, pp. 2175–2184.

[121] G. Douglas, B. Drawert, C. Krintz, and R. Wolski, “Cloudtracker: Using execution provenance to optimize the cost of cloud use,” in *Economics of Grids, Clouds, Systems, and Services (GECON).* Springer International Publishing, 2014, pp. 99–113.

[122] W.-T. Tsai, G. Qi, and Y. Chen, “Choosing cost-effective configuration in cloud storage,” in *International Symposium on Autonomous Decentralized Systems.* IEEE, March 2013, pp. 1–8.

[123] P. Teregowda and C. Giles, “Scaling seersuite in the cloud,” in *International Conference on Cloud Engineering (IC2E)*. IEEE, March 2013, pp. 146–155.

[124] Z. Zhang, L. Cherkasova, and B. T. Loo, “Exploiting cloud heterogeneity to optimize performance and cost of mapreduce processing,” *SIGMETRICS Perform. Eval. Rev.*, vol. 42, no. 4, pp. 38–50, Jun. 2015.

[125] M. Silva, M. Hines, D. Gallo, Q. Liu, K. D. Ryu, and D. da Silva, “Cloudbench: Experiment automation for cloud environments,” in *International Conference on Cloud Engineering (IC2E)*. IEEE, March 2013, pp. 302–311.

[126] R.-H. Hwang, C.-N. Lee, Y.-R. Chen, and D.-J. Zhang-Jian, “Cost optimization of elasticity cloud resource subscription policy,” *Transactions on Services Computing*, vol. 7, no. 4, pp. 561–574, Oct 2014.

[127] T. Guo, U. Sharma, P. Shenoy, T. Wood, and S. Sahu, “Cost-aware cloud bursting for enterprise applications,” *ACM Transactions on Internet Technology (TOIT)*, vol. 13, no. 3, pp. 10:1–10:24, May 2014.

APPENDIX A

Complete list of author's publications

Conference proceedings:

- **Daniel Moldovan**, Hong-Linh Truong, Schahram Dustdar, *Cost-aware scalability of applications in public clouds*, International Conference on Cloud Engineering, IC2E, IEEE, Berlin, Germany, 4-8 April, 2016, *accepted*
- Tien-Dung Nguyen, Hong-Linh Truong, Georgiana Copil, Duc-Hung Le, **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *On Developing and Operating of Data Elasticity Management Process*, Springer-Verlag, International Conference on Service Oriented Computing, ICSOC, Nov 16-19, 2015. Goa, India, `http://dx.doi.org/10.1007/978-3-662-48616-0_7`
- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *On Analyzing Elasticity Relationships of Cloud Services*, International Conference on Cloud Computing, CloudCom, IEEE, Singapore, 15-18 December, 2014, `http://dx.doi.org/10.1109/CloudCom.2014.93`
- Georgiana Copil, **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *On Controlling Cloud Services Elasticity in Heterogeneous Clouds*, Cloud Control Workshop, International Conference on Utility and Cloud Computing, UCC, IEEE/ACM, Singapore, London, 8-11 December, 2014, `http://dx.doi.org/10.1109/UCC.2014.88`
- Georgiana Copil, Demetris Trihinas, Hong-Linh Truong, **Daniel Moldovan**, George Pallis, Schahram Dustdar, Marios Dikaiakos, *ADVISE - a Framework for Evaluating Cloud Service Elasticity Behavior*, International Conference on Service Oriented Computing, Springer Berlin Heidelberg, ICSOC, Paris, France, 3-6 November, 2014. `http://dx.doi.org/10.1007/978-3-662-45391-9_19`

- Hong-Linh Truong, Schahram Dustdar, Georgiana Copil, Alessio Gambi, Waldemar Hummer, Duc-Hung Le, **Daniel Moldovan**, *CoMoT - A Platform-as-a-Service for Elasticity in the Cloud*, International Workshop on the Future of PaaS, International Conference on Cloud Engineering, IC2D, IEEE, Boston, Massachusetts, USA, 10-14 March 2014, `http://dx.doi.org/10.1109/IC2E.2014.44`

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *QUELLE - a Framework for Accelerating the Development of Elastic Systems*, European Conference on Service-Oriented and Cloud Computing, ESOCC, Springer Berlin Heidelberg, Manchester, UK, 2-4 September, 2014, `http://link.springer.com/section/10.1007/978-3-662-44879-3_7`

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *MELA: Monitoring and Analyzing Elasticity of Cloud Services*, International Conference on Cloud Computing, CloudCom, IEEE, Bristol, UK, 2-5 December, 2013, `http://dx.doi.org/10.1109/CloudCom.2013.18`

- Georgiana Copil, **Daniel Moldovan**, Hong-Linh Truong, Schahram Dustdar, (Short Paper), *Multi-Level Elasticity Control of Cloud Services*, International Conference on Service Oriented Computing, ACM, Berlin, Germany, 2-5 December, 2013, `http://dx.doi.org/10.1007/978-3-642-45005-1_31`

- Georgiana Copil, **Daniel Moldovan**, Hong-Linh Truong, Schahram Dustdar, (Demo Paper), *Specifying, Monitoring, and Controlling Elasticity of Cloud Services*, International Conference on Service Oriented Computing, ACM, Berlin, Germany, 2-5 December, 2013, `http://dx.doi.org/10.1007/978-3-642-45005-1_31`

- Alessio Gambi, **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *On Estimating Actuation Delays in Elastic Computing Systems*, International Symposium on Software Engineering for Adaptive and Self-Managing Systems (SEAMS), ACM, May 20-21, 2013, San Francisco, USA, `http://dx.doi.org/10.1109/SEAMS.2013.6595490`

- Georgiana Copil, **Daniel Moldovan**, Hong-Linh Truong, Schahram Dustdar, *SYBL: an Extensible Language for Controlling Elasticity in Cloud Applications*, International Symposium on Cluster, Cloud and Grid Computing (CCGrid), ACM/IEEE May 14-16, 2013, Delft, the Netherlands, `http://dx.doi.org/10.1109/CCGrid.2013.42`

- Ioan Salomie, Tudor Cioara, Ionut Anghel, **Daniel Moldovan**, Georgiana Copil and Pierluigi Plebani, *An Energy Aware Context Model for Green IT Service Centers*, Springer Berlin Heidelberg, International Workshop on Services, Energy,a nd Ecosystem, International Conference on Service Oriented Computing, ICSOC, San Francisco, CA, USA, December 7-10, 2010, `http://dx.doi.org/10.1007/978-3-642-19394-1_18`

- **Daniel Moldovan**, Georgiana Copil, Ioan Salomie, Ionut Anghel, Tudor Cioara, *A membrane computing inspired packing solution and its application to service center workload distribution*, International Conference on Intelligent Computer Communication and Processing, IEEE, ICCP, 30 Aug.-1 Sept. 2012, `http://dx.doi.org/10.1109/ICCP.2012.6356200`

- Tudor Cioara, Ionut Anghel, Ioan Salomie, Georgiana Copil, **Daniel Moldovan**, Alexander Kipp, *Energy Aware Dynamic Resource Consolidation Algorithm for Virtualized Service Centers Based on Reinforcement Learning*, International Symposium on Parallel and Distributed Computing, ISPDC, IEEE, 6-8 July 2011, `http://dx.doi.org/10.1109/ISPDC.2011.32`

- Tudor Cioara, Ionut Anghel, Ioan Salomie, **Daniel Moldovan**, Georgiana Copil, Pierluigi Plebani Dynamic consolidation methodology for optimizing the energy consumption in large virtualized service centers, Federated Conference on Computer Science and Information Systems, FedCSIS, IEEE, 18-21 Sept. 2011, `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=6078295`

- Ionut Anghel, Tudor Cioara, Ioan Salomie, Georgiana Copil, **Daniel Moldovan**, *An autonomic algorithm for energy efficiency in service centers*, International Conference on Intelligent Computer Communication and Processing, ICCP, IEEE, 26-28 Aug. 2010, `http://dx.doi.org/10.1109/ICCP.2010.5606425`

  Ionut Anghel, Tudor Cioara, Ioan Salomie, Mihaela Dinsoreanu, Georgiana Copil, **Daniel Moldovan**, *A self-adapting algorithm for context aware systems*, Roedunet International Conference, RoEduNet, IEEE, Sibiu, Romania, 24-26 June 2010 `http://ieeexplore.ieee.org/xpl/articleDetails.jsp?arnumber=5541540`

- Tudor Cioara, Ionut Anghel, Ioan Salomie, Mihaela Dinsoreanu, Georgiana Copil, **Daniel Moldovan**, *A reinforcement learning based self-healing algorithm for managing context adaptation*, International Conference on Information Integration and Web-based Applications & Services, iiWAS, ACM, Paris, France, 2010, `http://doi.acm.org/10.1145/1967486.1967634`

Journals:

- **Daniel Moldovan**, Georgiana Copil, Hong-Linh Truong, Schahram Dustdar, *MELA: Elasticity Analytics for Cloud Services*, International Journal of Big Data Intelligence, no. 1, vol. 2, 2015, `http://dx.doi.org/10.1504/IJBDI.2015.067569`

- Georgiana Copil, Hong-Linh Truong, **Daniel Moldovan**, Schahram Dustdar, Demetris Trihinas, George Pallis, and Marios D. Dikaiakos, *Evaluating Cloud Service Elasticity Behavior* International Journal of Cooperative Information Systems, 2015, `http://dx.doi.org/10.1142/S0218843015410026`

- Tudor Cioara, Ionut Anghel, Ioan Salomie, Georgiana Copil, **Daniel Moldovan**, Barbara Pernici, *A Context Aware Self-Adapting Algorithm for Managing the Energy Efficiency of IT Service Centres*, Ubiquitous Computing and Communication Journal, 2011, `http://www.ubicc.org/journal_detail.aspx?id=44`

Book chapters:

- Georgiana Copil, **Daniel Moldovan**, Duc-Hung Le, Hong-Linh Truong, Schahram Dustdar, Chrystalla Sofokleous, Nicholas Loulloudes, Demetris Trihinas, George Pallis, Marios D. Dikaiakos, Craig Sheridan, Evangelos Floros, Christos KK Loverdos, Kam Star, Wei Xing *On Controlling Elasticity of Cloud Applications in CELAR*, Emerging Research in Cloud Distributed Computing Systems. IGI Global, 2015, p 222-252, `http://dx.doi.org/10.4018/978-1-4666-8213-9.ch007`
- Alexander Kipp, Tao Jiang, Jia Liu, Mariagrazia Fugini, Ionut Anghel, Tudor Cioara, **Daniel Moldovan** and Ioan Salomie, *Energy-Aware Provisioning of HPC Services through Virtualised Web Services*, Springer, Studies in Computational Intelligence, Volume 432/2013, 29-53, `http://dx.doi.org/10.1007/978-3-642-30659-4_2`