Pós-Graduação em Ciência da Computação

Eliomar Gomes Campos

PERFORMANCE EVALUATION OF AUTO SCALING MECHANISMS IN PRIVATE CLOUDS FOR SUPPORTING A WEB SERVICE APPLICATION

M.Sc. Dissertation

Federal University of Pernambuco
posgraduacao@cin.ufpe.br
www.cin.ufpe.br/~posgraduacao

RECIFE
2015

Eliomar Gomes Campos

PERFORMANCE EVALUATION OF AUTO SCALING MECHANISMS IN PRIVATE CLOUDS FOR SUPPORTING A WEB SERVICE APPLICATION

A M.Sc. Dissertation presented to the Center for Informatics of Federal University of Pernambuco in partial fulfillment of the requirements for the degree of Master of Science in Computer Science.

Advisor: *Paulo Romero Martins Maciel*
Co-Advisor: *Rubens de Souza Matos Júnior*

RECIFE
2015

Catalogação na fonte
Bibliotecário Jefferson Luiz Alves Nazareno CRB4-1758

C198p Campos, Eliomar Gomes.
Performance evaluation of auto scaling mechanisms in private clouds for supporting a web service application / Eliomar Gomes Campos. – Recife: O Autor, 2015.
97 f.: fig., tab.

Orientador: Paulo Romero Martins Maciel.
Dissertação (Mestrado) – Universidade Federal de Pernambuco. CIN, Ciência da Computação, 2015.
Inclui referências e apêndices.

1. Avaliação de desempenho (Computadores). 2. Computação em nuvem. 3. Web services. I. Maciel, Paulo Romero Martins. (Orientador). II. Titulo.

004 CDD (22. ed.) UFPE-MEI 2015-126

Dissertação de mestrado apresentada por **Eliomar Gomes Campos** ao programa de Pós-Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco, sob o título **Performance evaluation of auto scaling mechanisms in private clouds for supporting a web service application**, orientada pelo **Prof. Paulo Romero Martins Maciel** e aprovada pela banca examinadora formada pelos professores:

Visto e permitida a impressão.
Recife, 3 de agosto de 2015.

Profa. Edna Natividade da Silva Barros
Coordenadora da Pós-Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal de Pernambuco

RECIFE
2015

I dedicate this dissertation to the great Creator of the universe.

Acknowledgements

Is at this moment I stop in time, and I remember the struggles I faced in my professional journey. So I try to search in the memory the countless times with people who helped me to come here.

I understand how much I ought to be grateful to the **Creator of the universe**, to the Creator of the love. The only to whom I owe all honor, glory and praise. He always comforted my heart, and maintained me steady when I thought about giving up, he made me understand that I was not alone.

I thank of ineffable way to my family, that in the first periods of my life expressed the following phrase: "This boy will be the future of the family". This phrase believe to be the largest contributor for my victories. My father **Eleomar Campos**, my mother **Maria das Graças**, my grandparents **Floriano Lopes** (in memoriam) and **Elza Campos**, my sister **Mayara Campos**. They deserve much more than I the honors of this work. To this crazy, united, and beautiful family, I give my eternal and inexorable love.

I direct my gratitude to whom absorbed my moments of defeats and victories, sorrows and joys. **Rafaela Wanderley**, his company and love, it is essential to my happiness.

I am very grateful to all the teachers who have gone through all of my educational background, and at this, especially to my advisor **Paulo Maciel**, if it was not the constant and patient support in my doubts, maybe I had not achieved this victory.

I could not fail to thank those who are part the **MoDCS Research Group**. Especially for **Rubens Matos**, by the spirit of unity, knowledge sharing, and friendship. Certainly, the merits of this work are also of yours.

I would like to thank the Foundation for Support to Science and Technology of Pernambuco (FACEPE) for their financial support.

I have fought the good fight, I have finished the course, I have kept the faith: henceforth there is laid up for me the crown of righteousness, which the Lord, the righteous judge, shall give to me at that day; and not to me only, but also to all them that have loved his appearing.

—2 TIMOTHY 4:7-8

Resumo

Serviços web compostos, também conhecidos como mashups, são úteis para construir produtos de valor agregado na web. Ambientes de computação em nuvem têm sido amplamente utilizados para hospedar serviços web, devido à possibilidade de aumentar ou diminuir os recursos disponíveis através de mecanismos automáticos (i.e.: escala automática). Tal comportamento elástico facilita a tarefa de alcançar um desempenho satisfatório nos picos de demanda sem desperdiçar recursos. É difícil determinar os componentes certos para ajustar o desempenho desses sistemas eventualmente, quando necessário. Este estudo avalia o desempenho dos mecanismos de escala automática e elasticidade para nuvens privadas hospedando um serviço web de recomendação de eventos. Uma abordagem de modelagem hierárquica é utilizada afim de lidar com a complexidade de tal sistema, e representar detalhes específicos desses mecanismos. Nosso estudo aplicou análise de sensibilidade paramétrica a partir de várias métricas de desempenho dos modelos, tais como o tempo médio de execução do monitoramento de escala automática, tempo médio da instanciação de VMs e o tempo médio da resposta percebida pelo usuário do serviço web. Realizamos também um Experimento Geral Fatorial Completo, com o objetivo de calcular os efeitos e relevâncias de cada fator envolvido nos processos escala automática e instanciação de máquinas virtuais (virtual machines - VMs). Para o monitoramento de escala automática, analisamos os fatores: período de coleta de uma métrica, número de máquinas virtuais monitoradas, e o tempo de monitoração de uma métrica. Quanto ao processo de instanciação, os seguintes fatores foram escolhidos: tipo de VM, tamanho da imagem da VM, e cache da VM. Estas análises permitem verificar o impacto dos parâmetros sobre o tempo de resposta do sistema e apontar formas eficazes de melhoria do desempenho.

Palavras-chave: Computação em Nuvem. Nuvem Privada. Escala Automática. Avaliação de Desempenho. Modelagem Analítica.

Abstract

Composite web services, also known as mashups, are useful to build added-value products in the web. Cloud computing environments have been widely used for hosting web services due to the possibility of increasing or decreasing available resources through automatic mechanisms (i.e.: auto scaling). Such elastic behavior ease the task of reaching satisfactory performance on peaks of demand without wasting resources. It is hard to determine the right components to tune such systems performance when eventually needed. This study evaluates the performance of auto scaling mechanisms for private clouds hosting an event recommendation web service. A hierarchical modeling approach is used to cope with the complexity of such a system, and represent specific details of these mechanisms. Our study applies parametric sensitivity analysis from several performance metrics of the models, such as mean execution time of the auto scaling monitoring, mean time of VMs instantiation, and the mean response time perceived by the web service user. We also have carried a General Full Factorial Experiment, in order to calculate the relevance and effects of each factor involved in the processes of auto scaling and virtual machines (VMs) instantiation. For the auto scaling monitoring, we analyze the factors: collection period of a metric, number of monitored virtual machines, and the time of monitoring of a metric. Regarding the instantiation process, the following factors have been chosen: VM type, VM image size, and VM caching. This analysis allows checking the impact of parameters on the system response time and pointing out effective ways for improvement of performance.

Keywords: Cloud Computing. Private Cloud. Auto Scaling. Performance Evaluation. Analytical Modeling.

List of Figures

List of Tables

List of Acronyms

SLAs	Service Level Agreements	15
SPN	Stochastic Petri Net	16
CTMC	Continuous Time Markov Chain	16
QoS	Quality of Service	16
DTMC	Discrete Time Markov Chain	18
DoE	Design of Experiments	21
IT	Information Technology	23
NIST	National Institute of Standards and Technology	23
IaaS	Infrastructure as a Service	26
PaaS	Platform as a Service	26
SaaS	Software as a Service	26
CRM	Customer Relationship Management	26
ERP	Enterprise Resource Planning	26
API	Application Programming Interface	27
PN	Petri Nets	29
AWS	Amazon Web Services	33
CLC	Cloud Controller	33
EMI	Eucalyptus Machine Image	34
EKI	Eucalyptus Kernel Image	34
ERI	Eucalyptus Ramdisk Image	34
S3	Amazon Simple Storage Service	34
CC	Cluster Controller	34
NC	Node Controller	34
SC	Storage Controller	34
EBS	Eucalyptus Block Store	34
ELB	Elastic Load Balancing	34
GPS	Global Positioning System	35
DSPN	Deterministic and Stochastic Petri Net	45

STA	Stochastic Timed Automaton	47
ANOVA	Analysis of Variance	59
MTTA	Mean Time to Absorption	78

Contents

1 Introduction

Cloud computing provides on-demand access to shared computing resources and services, such as: network infrastructure, storage, operating systems, and applications. Such resources and mechanisms can be easily acquired and released with minimal management effort (NIST, 2013). These features enable administrators to focus only on the business model, without worrying about infrastructure details (NIST, 2013; SOUSA et al., 2012). The experience in acquiring cloud services is often compared to the consumption of public utilities, such as electricity, so the user just pay for that service without worrying or knowing about the infrastructure that provides it (BAUER; ADAMS, 2012).

Despite the benefits of acquiring services from third-party providers, some companies prefer investing on privately managed infrastructures, known as private clouds, to get the advantages of flexible and efficient usage of physical resources while keeping the control over the data. Such a solution also allows a smoother transition to the cloud paradigm before a complete migration to a third-party service (i.e., public cloud), as well as to setup a hybrid cloud, where both infrastructures – private and public – are simultaneously used (EUCALYPTUS, 2013a).

Auto scaling and elastic load balancing are important mechanisms that enable flexible allocation of resources in cloud computing and enforce the fulfillment of Service Level Agreements (SLAs) in environments with highly varying workloads (CARON et al., 2012; GUSEV et al., 2013). Applications running on cloud environments, and using auto scaling and load balancing features, are designed to instantiate or terminate virtual machine (VM) instances according to the current workload level. Such a behavior avoids the waste of idle resources (e.g.: memory, CPU, disk space, power) in periods of low load, whereas enables the fast increase of computational power when facing a burst of high load (CARON et al., 2012; EUCALYPTUS, 2013b).

This study evaluates the performance of auto scaling mechanisms for private clouds hosting a web service application that must deal with changes on workload intensity. The evaluation uses an experimental approach combined with analytical modeling. The main focus is on measures such as the response time of the web service and the time for completion of auto scaling activities. A hierarchical modeling approach is used to cope with the complexity of such a system and represent details of specific processes, such as the auto scaling monitoring, instantiation of VMs, and the calls for the providers of specific web services that compose the mashup application. The hierarchical model comprises a Stochastic Petri Net (SPN) (MOLLOY, 1982; MARSAN; CONTE; BALBO, 1984) as main model and Continuous Time Markov Chain (CTMC) (KLEINROCK, 1975; BOLCH et al., 2001) as submodels.

This study applied a *General Full Factorial Design of Experiments* (GUIMARÃES; MACIEL; MATIAS JR, 2013; MONTGOMERY, 2012; JAIN, 2008) for analyzing two specific processes of the private cloud system, namely the auto scaling monitoring and instantiation of VMs. The experiments evaluated the impact of three factors on the total time for triggering auto scaling actions: the period taken to collect metrics, the number of monitored VMs, and window time for monitoring the metric compliance to a given threshold. The efficient instantiation of VMs is one requirement for the elastic behavior of cloud-based applications. Therefore, this study also characterizes how factors such as VM type, VM image size, and VM caching impact the total time spent for creating one VM instance in the private cloud.

1.1 Motivation and Justification

Applications that combine data from distinct web services, known as mashup services, are a current trend for building added-value products in the web (MA et al., 2013; LEITNER; HUMMER; DUSTDAR, 2011; LI; FANG; XIONG, 2008). These applications might have complex application logic, i.e.: dependency among components and various points of parallelism, so it is non-trivial to check whether or not the composed service will meet the Quality of Service (QoS) characteristics that users may expect. In this context, analytical approaches to determine the overall performance and reliability of composite web services have been proposed in the literature (MATOS; MACIEL; SILVA, 2013; REN et al., 2009; ZHONG; QI; XU, 2008; SATO; TRIVEDI, 2007).

Furthermore, many web service administrators face the challenge of dynamically adapting to workload fluctuations which sometimes require more or less infrastructure resources such as storage, processing, memory, and bandwidth. Banking, social networking, e-commerce, and access to online games, are examples of applications that might receive sudden surges in traffic, and thus, compromise performance of service. The inability to deal with workload peaks might also cause service interruption in some cases, resulting in customer dissatisfaction, and financial damage. On the other hand, very low traffic may underuse resources. It is complex to predict the exact moment that these sudden events happen and reallocate resources efficiently (EUCALYPTUS, 2014a; SULEIMAN; VENUGOPAL, 2013; YANG et al., 2013).

In order to overcome these challenges, cloud computing environments allow increasing or decreasing available resources manually or through automatic mechanisms (i.e.: auto scaling), by adopting the elasticity concept. This elasticity, or scalability, is mainly due to the virtualization technologies, that allow optimizing the usage of physical resources and reducing energy consumption, by allocating more services in less computers than seen in non-virtualized environments (EUCALYPTUS, 2014b; SULEIMAN; VENUGOPAL, 2013).

Private cloud platforms depend on the auto scaling mechanism to make more efficient usage of available resources, and allow adaptation of web services execution to fluctuating workload. In other words, the promptness to dynamically adapt to sudden workload peaks depends on the total time of the auto scaling process, comprising the interval between detecting the necessity for better performance, and subsequently implementing a policy that resizes the number of VM instances.

In order to take the most benefits from these mechanisms, some relevant factors must be properly configured in the cloud platform. Otherwise, the performance may be strongly affected (EUCALYPTUS, 2014a). A plethora of related research studies overlook some cloud configuration factors that have direct impact on auto scaling performance.

Therefore, it is important to evaluate the performance of a web service application running on a private cloud with elasticity mechanisms, and through the combination of statistical techniques, identify the impact of some configuration and workload factors. The evaluation approach and results from this study can help system administrators to properly configure the auto scaling mechanism in private clouds frameworks. Such results can also aid in the development of techniques or algorithms to improve performance of auto scaling functions in private clouds frameworks.

1.2 Objectives

This section describes the general and specific objectives of this research. The major objective of this work is to evaluate the performance of auto scaling mechanisms for private clouds hosting an event recommendation web service. There are some specific objectives, described as follows:

- To propose a methodology for evaluation of the auto scaling mechanisms of a private cloud;
- To investigate the effects of private cloud configuration factors on the performance of the auto scaling mechanism;
- To characterize the times spent in each specific activity of the auto scaling triggering and VM instantiation processes;
- To develop an analytical model for predicting the response time of a web service hosted in a scalable private cloud.

1.3 Related works

Other authors have proposed mechanisms for QoS prediction of composite web services through analytical models without considering the employment of auto scaling or cloud computing. In (SILVA et al., 2006) and (ZHONG; QI; XU, 2008), the prediction of metrics for composite services using SPNs is studied, considering performance and reliability aspects, respectively. In (REN et al., 2009), Discrete Time Markov Chain (DTMC) are used for reliability computation, while CTMCs are used in (SATO; TRIVEDI, 2007) to find reliability and performance bottlenecks of a travel agent composite web service through sensitivity analysis. Closed-form expressions are also used in (SATO; TRIVEDI, 2007) for computing the response time and reliability of the composite web service. The work presented in (MATOS; MACIEL; SILVA, 2013) integrates an optimization framework to analytical models for the prediction of QoS metrics of fixed web services compositions. Our dissertation also uses stochastic models for performance prediction of composite web services. But, our approach differs from the mentioned papers (SILVA et al., 2006; ZHONG; QI; XU, 2008; REN et al., 2009; SATO; TRIVEDI, 2007; MATOS; MACIEL; SILVA, 2013) by assessing the auto scaling mechanism in private clouds.

Performance evaluation of cloud computing systems has been a topic of many recent works. There have been some efforts to propose auto scaling mechanisms or evaluate them in distinct systems, usually with single (non-composite) web services or similar applications. Several studies evaluated the performance of services in an environment using auto scaling (SULEIMAN; VENUGOPAL, 2013; BOTRAN et al., 2014; YANG et al., 2013; LIN et al., 2012; FERRARIS et al., 2012; AL-HAIDARI; SQALLI; SALAH, 2013).

Suleiman *et al.* (SULEIMAN; VENUGOPAL, 2013) determined when and how to add or remove instances of servers in cloud environments. They built an analytical model to study the effects of some metrics and their respective thresholds, such as the CPU utilization, the application response time, monitoring time windows, and the number of requests to a Web service. Using the proposed elasticity models and algorithms, they conducted simulation experiments with a number of elasticity rules with different CPU utilization thresholds. They have validated the resulting metrics against the results from the experiments have conducted using the same rules and workload on Amazon EC2. The simulation results demonstrated reasonable accuracy of their elasticity models and algorithms in approximating CPU utilization, application's response time, and number of servers. The models have been able to capture the trends and relationship between changing CPU utilization thresholds and these metrics with acceptable variations.

Suleiman *et al.* (SULEIMAN; VENUGOPAL, 2013) did not consider the subsystems or specific mechanisms of the scalable web service, i.e., modeled only the architecture of the main system at a high level, using a queue-based model. The application used was a simple web service (non-composite). Suleiman *et al.* (SULEIMAN; VENUGOPAL, 2013) did not try to identify the most impacting factors on the system. In our work, we conducted analyses on models and experimental designs, considering the whole system and also some subsystems and details of specific mechanisms.

Al-Haidari *et al.* (AL-HAIDARI; SQALLI; SALAH, 2013) analyzed the impact of configuring some factors (CPU utilization threshold and scaling size) on the performance of Amazon EC2 (AMAZON, 2014a) services. They aimed at improving performance indices by configuring system parameters. A queue model similar to that proposed by Suleiman was used in the system behavior simulation. From the simulation results, they found that such configuration parameters of the provisioning mechanism have a considerable impact on the performance and cost of cloud services. In addition, they formulated and solved optimization problems for tuning the upper CPU utilization threshold and scaling size based on input loads, considering both the cost and the response time.

In a similar manner to the study of Al-Haidari *et al.* (AL-HAIDARI; SQALLI; SALAH, 2013) our dissertation first focuses on presenting a model that includes the auto scaling mechanism. Second, our model mainly considered tuning some parameters related to a cloud provider and not to the cloud customers. Finally, in our study, we present the impact of the auto scaling configuration parameters on the mean response time of the web service hosted in the cloud. Whereas Al-Haidari *et al.* only consider the factors CPU utilization threshold and scaling size, we consider several other factors. As well as Suleiman *et al.* (SULEIMAN; VENUGOPAL, 2013), Al-Haidari *et al.* (AL-HAIDARI; SQALLI; SALAH, 2013) also did not consider the subsystems or specific mechanisms of scalable web services.

In order to achieve the scalability, is important to know when and how to scale virtual resources assigned to different services. Yang *et al.* (YANG et al., 2013) used a linear regression model to predict the workload for services in a virtualized environment, and proposed an auto scaling mechanism to preallocate or free virtual resources according to the predicted workload. The automatic scaling mechanism combines the realtime scaling and the pre-scaling. Vertical scaling is implemented by changing the partition of resources (e.g. CPU, memory, storage, etc.) inside a VM, modern hypervisors support on-line VM resizing without shutting it down. The vertical scaling can scale virtual resources in a few milliseconds, according to them, for this reason, this type of technique can be in realtime scaling. The horizontal scaling adjusts the amount of Virtual Machine (VM) instances, it incurs a considerable waste of resources sometimes. Beyond that, the horizontal scaling takes several minutes to boot a VM, and the new VM cannot be used at once. According to them, for this reason, this type of technique can be in pre-scaling. Experimental results are provided to demonstrate that this approach can satisfy the user SLA while keeping scaling costs low.

However, the pre-scaling algorithm proposed by Yang *et al.* could be improved, if it took into account adjustments on various parameters that involve VM instantiation, and not only the types of VMs. They also overlook the time that the auto scaling monitoring mechanism takes for detecting that the threshold violation, and thereby increase or decrease a resource at realtime. Depending on this monitoring time, the vertical scaling can suffer long delays. The work proposed by Yang *et al.* (YANG et al., 2013) contributes to a lower risk of SLA violation, however, they should evaluate the parameter settings of scaling mechanisms, in order to improve even more their proposal.

Lin *et al.* (LIN et al., 2012) developed an auto scaling system and proposed an algorithm capable of analyzing the changes of workload in a web application. According Lin *et al.* (LIN et al., 2012), most of the existing solutions are subject to some of the following constraints: replying on user-provided scaling metrics and threshold values; employing the simple Majority Vote scaling algorithm, which is ineffective for scaling web applications; and lack of capability for predicting workload changes. Their auto scaling system is not subject to the aforementioned constraints. The ability of inferring the workload in advance is used to reduce the impact of some factors on system performance. The experiment results demonstrate that the proposed auto scaling system can keep the response time of Web applications low even when facing sudden load changing.

Ferraris *et al.* (FERRARIS et al., 2012) investigated the performance of some auto scaling characteristics offered by the cloud providers Amazon and Flexiscale. They aimed at analyzing patterns of a useful configuration for executing web applications in the cloud, and highlighted the critical factors which affected the performance of both providers. They performed a large set of experiments that demonstrated the importance of tuning correctly the auto scaling parameters. The experiments showed that the tuning of auto scaling parameters is challenging and more work shall be developed in order to implement a pro-active behaviour, since simple actions triggered after threshold violations cannot provide performance guarantees under critical workload conditions.

Notice that the work of Ferraris *et al.* (FERRARIS et al., 2012) is very close to our proposal. They consider the auto scaling monitoring period and VM instantiation in different scenarios. Their work proposes parameter settings in order to check the impact on performance, however they perform only few adjustments in each scenario. Our dissertation not only adjusts the scenarios, but also varies each parameter in a reasonable range of possible values, and verify the impact of these changes on a performance metric, i.e., we performed parametric sensitivity analysis through the proposed analytical model. Despite considering in the experimental design many of the factors that we also have studied, Ferraris *et al.* did not perform statistical analysis of the most relevant factors that deserve priority in the auto scaling settings. Furthermore, the stochastic models that we propose may help to analyze and predict various metrics of the systems and its subsystems. Beyond those observed in experiments, going a step further from the work of Ferraris *et al.*.

In (SOUSA et al., 2012), Sousa *et al.* evaluated the performance of distinct VM types in the Eucalyptus platform, using well-known benchmarks. Their analysis enables to assess the quality of service and prevent performance degradation related to fluctuations in workload in private clouds. The relationship of that work with ours is just in the proposal to properly configure one or more parameters, while also pointing out factors and scenarios that need more attention for adjustments.

Table 1.1 shows the relationship between the proposals of this dissertation and main related works. Notice that majority of mentioned authors studied environments of public cloud computing, whereas we performed experiments in a private cloud environment. All authors that evaluated composite web services did not use any cloud environment. The remaining evaluated a simple web applications only. One work specifically evaluated the auto scaling monitoring, and two the VM instantiation stage. The majority investigated only the process of auto scaling as a whole. Everyone that used stochastic models do not consider any cloud environment. In our models we consider cloud environment, and in addition the elasticity mechanisms. Our main distinction is the employment of hierarchical models in a context of cloud-based web services. The hierarchical heterogeneous approach enables representing details of specific processes (subsystems) of the main system, such as auto scaling monitoring, VM instantiation, and the calls for composite web service, thus, most of the related works do not consider such subsystems. The hierarchical model comprises an SPN as main model and CTMC as submodels of the main model. In addition, all studies using analytical modeling did not employ the design of experiments. Only three papers consider Design of Experiments (DoE), however without proposing models. Our study used models and DoE for parametric sensitivity analysis, just one paper applied such a type of study, however, only on the model. Table 1.1 highlights that our work merges the various contributions that are lacking in related works.

Table 1.1: Relationship between the proposals of this dissertation and related works

			Main contributions of the dissertation								
		Cloud Type	Evaluates composite web service	Evaluates single web service	Evaluates all auto scaling process	Evaluates auto scaling monitoring	Evaluates VM instantiation	Uses models	Heterogeneous hierarchical modeling	DoE	Parametric sensitivity analysis
Related Works	This dissertation	Private	✓		✓	✓	✓	SPN, CTMC	✓	✓	✓
	(SILVA et al., 2006)		✓					SPN			
	(ZHONG; QI; XU, 2008)		✓					SPN			
	(REN et al., 2009)		✓					DTMC			
	(SATO; TRIVEDI, 2007)		✓					CTMC			✓
	(MATOS; MACIEL; SILVA, 2013)		✓					CTMC			
	(SULEIMAN; VENUGOPAL, 2013)	Public		✓	✓					✓	
	(YANG et al., 2013)	Public		✓	✓		✓				
	(LIN et al., 2012)	Private		✓	✓						
	(FERRARIS et al., 2012)	Public		✓	✓	✓	✓			✓	
	(SOUSA et al., 2012)	Private								✓	

1.4 Structure of the dissertation

The remaining parts of the dissertation are organized as follows. Chapter 2 presents all the theoretical basis for the work, introducing the main concepts regarding cloud computing, and then we have approached the foundations performance evaluation of systems by focusing on measurement and analytical modelling. Chapter 3 describes the composite web service system evaluated here and its mechanisms (or subsystems), providing a glance at the interaction between components and their behavior. Chapter 4, presents details on the methodology of performance evaluation adopted in this dissertation. Shortly thereafter we present the models proposed for the prediction of some performance metrics. Chapter 5, presents the results of the models and design of experiments. These analyses are arranged in four case studies. Each case study includes sensitivity analysis on certain parameters of the evaluated mechanism. Chapter 6 draws the final conclusions for this study by presenting its main contributions and suggesting possible future studies.

2 Background

This chapter presents the main concepts about cloud computing (EUCALYPTUS, 2014a), and performance evaluation of systems, through analytical techniques using hierarchical modeling, and measurement with the DoE.

2.1 Cloud Computing

Cloud computing, by definition is usually defined as the on-demand delivery of Information Technology (IT) resources and applications via the Internet with pay-as-you-go pricing (AMAZON, 2015). Figure 2.1 illustrates in a general way the six stages in the history of computing paradigms: from mainframes and thin terminals to PCs, from networking computing to grid and cloud computing (VOAS; ZHANG, 2009; FURHT, 2010).

In stage 1, mainframes are shared with many users and accessed through terminals (keyboards and monitors). In stage 2, personal computers become powerful enough to suit all the daily needs of users, thus there is no need to share a mainframe with other users. Stage 3, introduces computer networks that allow multiple computers to connect to each other. PCs, laptops, and servers are connected together through local networks to share resources and increase performance. In stage 4, local networks are connected to other local networks forming a global network such as the Internet to use remote applications and resources. In stage 5, grid computing provided shared computing power and storage through a distributed computing system, i.e., the people use PCs to access a grid of computers in a transparent manner. In stage 6, cloud computing lets you access and explore hardware and software resources available on the Internet in a scalable and simple way (VOAS; ZHANG, 2009; FURHT, 2010).

The U.S. National Institute of Standards and Technology (NIST) defines cloud computing as follows:

> Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction (MELL; GRANCE, 2011).

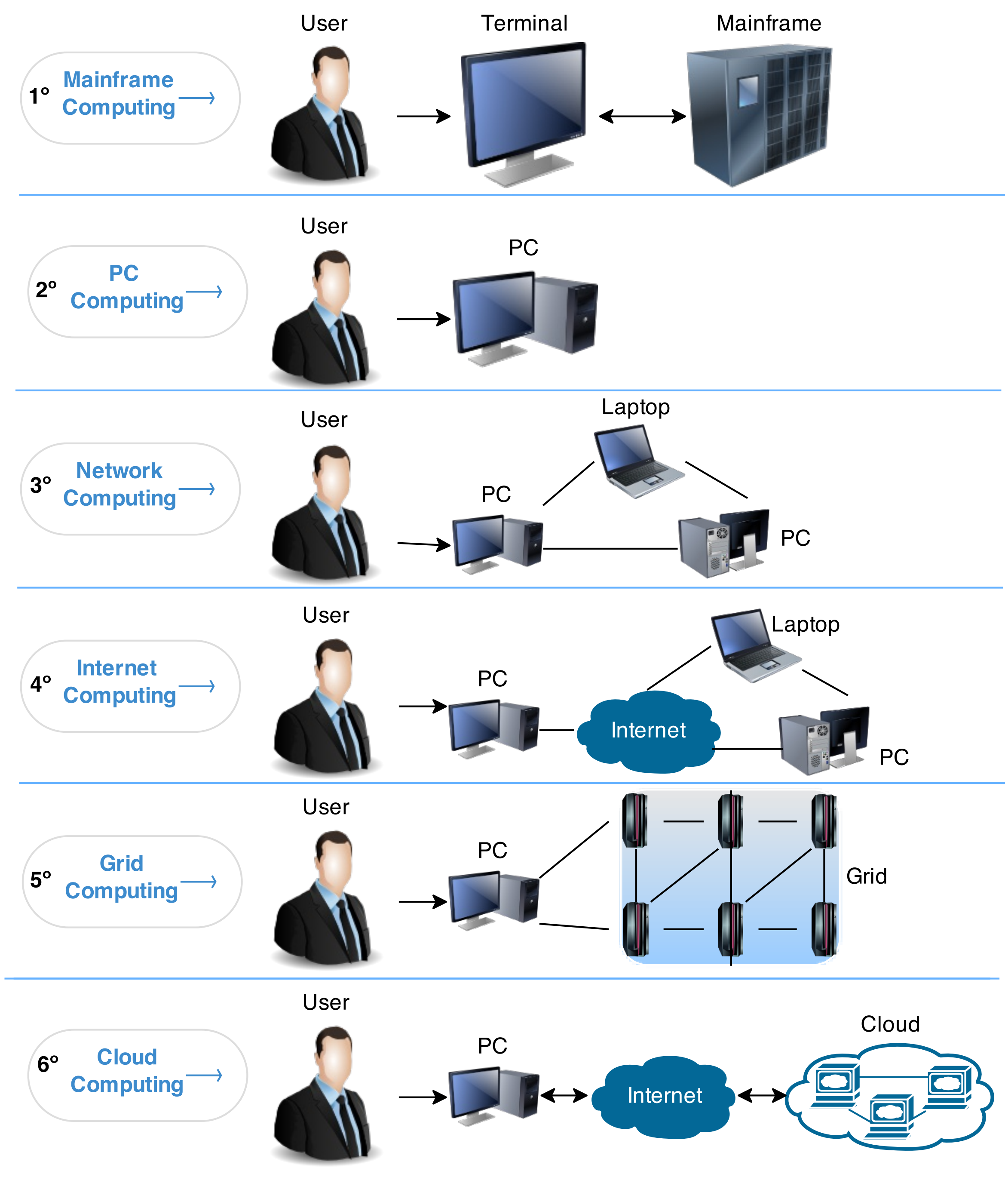


Figure 2.1: Computing paradigm shift over six distinct stages.

In Figure 2.1, the cloud computing paradigm is shown in conceptual form, because all resources (hardware or software) are not visible to the user, i.e., they are offered in a transparent manner as services. The services are accessed of the Internet through standard interfaces. Cloud computing thus refers to the techniques that enable and facilitate this scenario. Therefore, for the cloud computing paradigm, there is no need for powerful PCs, because all services are processed on servers on the Internet. From this perspective, cloud computing seems like a "return" to the original mainframe paradigm. Although, the cloud computing paradigm is not that simple.

Unlike a mainframe, which is a physical machine that offers finite computing power, a cloud represents a highly scalable set of resources on the Internet, suggesting virtually infinite power and capacity. Meanwhile, unlike a simple terminal acting as a user interface to a mainframe, a PC in the cloud computing paradigm possesses enough power to provide a certain degree of local computing and caching support (VOAS; ZHANG, 2009; FURHT, 2010). The cloud model comprises five essential characteristics, three service models, and four deployment models, which are described as follows.

2.1.1 Essential Characteristics

The NIST specifies that cloud computing has five essential traits (MELL; GRANCE, 2011; EUCALYPTUS, 2014c):

- **On-demand self-service**: Users are able to provision infrastructure, development tools, software, and other resources on their own, often via a Web browser. As such, they can get what they need when they need it, without having to go through a service provider or conduct a lengthy procurement process (EUCALYPTUS, 2014c).
- **Broad network access**: The resources (e.g., storage, processing, memory, and network bandwidth) are available through the network, and can be accessed by different platforms (e.g., mobile phones, tablets, laptops, and workstations) in a transparent manner through standardized mechanisms (MELL; GRANCE, 2011).
- **Resource pooling**: The resources of several physical servers are combined and offered to different types of customers (multi-tenant model), and dynamically distributed according to the current demand of the customer. There is a sense of location independence of these resources, i.e., customers do not have control or knowledge over the exact location of the resources. But at a higher level of abstraction, through a customer's control panel can be configured (e.g., country, state, or datacenter) (MELL; GRANCE, 2011).
- **Elasticity**: Cloud resources are flexible, capable of being dynamically reassigned and/or released in response to sudden changes in user demand (EUCALYPTUS, 2014c).
- **Measured service**: Users and administrators can monitor and control how resources are utilized. As a result, they can track cloud computing spend, which often follows a variable, pay-as-you-go business model (EUCALYPTUS, 2014c).

2.1.2 Service Models

The services offered by cloud computing can be presented as a layered architecture (FURHT, 2010), as show in Figure 2.2. These services are also known as business models. NIST formally defines three service models for cloud computing: infrastructure as a service, platform as a service, and software as a service.

Infrastructure as a Service (IaaS): Cloud providers offer storage, processing, memory, and network where customers can install operating systems and applications. The cloud infrastructure control is the responsibility of the providers, customers only have control over operating systems, applications, storage configurations, and possibly limited control of network components (e.g., host firewalls). Thus, customers may use infrastructure of third-party servers paying only for what you use. Without worrying about security, maintenance, and space, while decreasing spending on professionals and cooling (MELL; GRANCE, 2011; BAUER; ADAMS, 2012).

Platform as a Service (PaaS): Provides to developers of software a development environment, allowing the implementation of the applications developed using such an environment. Developers have no control over the infrastructure, but only access to the development environment configurations and applications. PaaS services include: operating system, virtual desktop, web services delivery and development platforms, and database services (MELL; GRANCE, 2011; BAUER; ADAMS, 2012).

Software as a Service (SaaS): This service model provides applications running on a cloud. SaaS applications include: e-mail and office productivity suites, Customer Relationship Management (CRM), Enterprise Resource Planning (ERP), social networking, collaboration, document, and content management. The customer does not manage or configure the cloud infrastructure, but it is possible limited settings for specific applications. (MELL; GRANCE, 2011; BAUER; ADAMS, 2012).

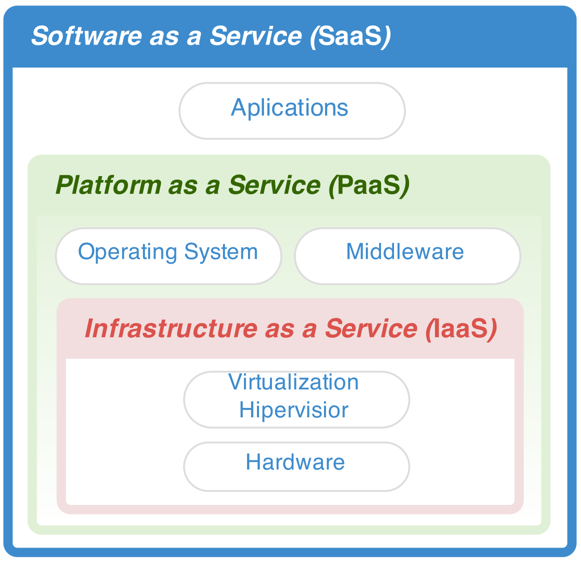

Software as a Service (SaaS)
Aplications
Platform as a Service (PaaS)
Operating System
Middleware
Infrastructure as a Service (IaaS)
Virtualization Hipervisior
Hardware

Figure 2.2: Relationship between the service models

2.1.3 Deployment Models

There are many issues to consider when moving an enterprise application to the cloud environment. For example, some service providers are mostly interested in lowering operation cost, while others may prefer high reliability and security. Accordingly, there are different types of cloud computing deployments, each with its own benefits and drawbacks (ZHANG; CHENG; BOUTABA, 2010). The NIST recognizes four cloud deployment models:

- **Private Cloud**: The cloud infrastructure is provided to a single organization, i.e., the resources are not shared with more than one organization. Private clouds are managed by the organization itself, thus is possible to get even more security on the information. (MELL; GRANCE, 2011; EUCALYPTUS, 2014a).
- **Community Cloud**: The cloud infrastructure is provided for exclusive use by a specific community of consumers from organizations, i.e., the resources are shared with more than one organization that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). Community clouds are managed by one or more organization in the community (MELL; GRANCE, 2011).
- **Public Cloud**: The cloud infrastructure resources are shared to the general public (e.g., companies, universities, governments, and common customers). The infrastructure is provided and managed by the cloud providers. For this model, security requirements is the subject of several studies and debates (MELL; GRANCE, 2011; EUCALYPTUS, 2014a).
- **Hybrid Cloud**: This type of infrastructure combines two or more distinct cloud infrastructures (private, community, or public). Through mechanisms that have standardized technology (e.g., Application Programming Interface (API)s) it is possible to balance the workload between the two or more infrastructure when a predefined threshold is reached (MELL; GRANCE, 2011; EUCALYPTUS, 2014d).

2.2 Performance Evaluation of Systems

System administrators need to provide the highest performance at the lowest cost. A performance evaluation is necessary when a system administrator wants to compare a number of alternative configuration scenarios to find the best one. It is also used to compare two similar systems and decide which one is the best for a given task. Performance evaluation can also help to determine how well a system is performing certain tasks, and if some improvements are necessary. Generally, evaluating the performance of a system means to verify its behavior according to a defined metric. The researcher must select appropriate evaluation techniques (e.g.:

analytical modeling, *simulation* or *measurement*), perform a statistical analysis to identify possible bottlenecks and propose improvement solutions. This work applied a parametric sensitivity analysis from the analytical modeling with SPN and CTMC models, and measurements based on the DoE technique.

2.2.1 Measurement

DoE technique allows to obtaining a maximum of information about a system, regarding many factors, with a reasonable number of experiments and effort (JAIN, 2008; MONTGOMERY, 2012). A set of experiment executions planned through DoE can be analyzed to determine if the factors have significant effects, or if the differences in the observed effects are due to variations caused by measurement errors and not controlled parameters (GUIMARÃES; MACIEL; MATIAS JR, 2013; JAIN, 2008; MONTGOMERY, 2012).

This study adopts the *General Full Factorial Design*, which uses all possible combinations of levels for all factors, i.e., there are no limits to the number of factors and the number of levels. This type of DoE allows every configuration to be examined, so we can find the effects of all factors and their interactions, which is an advantage; the disadvantage is that the cost of analysis can be very high if the number of factors and levels is too high, and also considering that each of these experiments may have to be repeated several times. It is possible to reduce the number experiments by reducing the number of factors, and/or the number of levels for each factor, or using *Fractional Factorial Design* instead (JAIN, 2008).

2.2.2 Continuous Time Markov Chains

As it is shown in Figure 2.3, Markov chains can be represented as a directed graph with labeled transitions, indicating the probability or rate at which such transitions occur. In Markov chains, the states represent different conditions that the system may follow. The transitions between the states indicate the occurrence of events (SILVA et al., 2013) (e.g.: the arrival of tasks, or completion of service). In Figure 2.3, a new task arrives with rate λ, and a server completes the task with rate μ. For example, Figure 2.3 depicts a model for a system with two servers that process incoming jobs. If we observe the number of busy servers as a time function, we can consider it as a random variable or function $X(t)$. Each modification of X over (t) is called *state* $X_n(t)$. The set of all possible states is the *state space* of the model. Thus, it is possible to find the transition probabilities from a state to its successor $X_{n+1}(t)$. For this, you need to specify the probability distribution function of $X_n(t)$. Such sequences or random functions of time are called stochastic processes. Stochastic processes are processes in which the random variable changes its state over time (JAIN, 2008; MACIEL et al., 2011; KLEINROCK, 1975). They are usually adopted to characterize systems whose behavior is inherently probabilistic (SILVA et al., 2013).

Analytical modeling may consider a random variable or several sequences or families of random variables. With only one random variable is simple to know what is the probability of its states over time (stationary) probability or at a specific time (transient) probability. Those probabilities are obtained by computing the *distribution function*. However, when we represent a number of phenomena in a system, i.e., several random variables, the calculation may be complex, because it requires computing the *joint distribution function*. On the other hand, the calculation of probabilities for a random variable can be simplified when applied to an *exponential distribution function* or *geometric distribution function*. Markov chains is a state space model widely adopted to work with such distribution functions, and therefore simplify the analysis of systems modeled through many random variables.

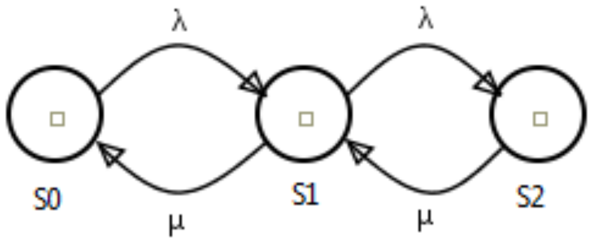


Figure 2.3: Example of a CTMC model

Markov chains are associated to a Markov process (HAVERKORT, 2002), and are outstanding stochastic models, used to analyze a variety of systems (SILVA et al., 2013). We have a Markov process if the past history is not important to know the probability of reaching a given future state. Only the current state is enough to know such a probability (property known as lack of memory). When the Markov process has a discrete state space, then it is known as a Markov chain. A Markov chain with discrete time parameter is called a DTMC. On the other hand, if the time parameter assumes real values, the model is called a CTMC (JAIN, 2008; MACIEL et al., 2011; STEWART, 1994). In a homogeneous DTMC, the time spent in a state follows a geometric distribution, while in the homogeneous CTMC follows an exponential distribution. Markov chains are said to be homogeneous, when the transition probability between states does not depend on time but only on the current state (MACIEL et al., 2011). Markov chains have been used extensively in dependability, performance, and performability modeling (MACIEL et al., 2011; TRIVEDI, 2001). CTMC was a useful modeling formalism for evaluating the performance of the private cloud system studied in this work.

2.2.3 Stochastic Petri Nets

Petri Nets (PN) (MURATA, 1989) are a family of formalisms very well suited for modeling several system types, since concurrency, synchronization, communication mechanisms as well as deterministic and probabilistic delays are naturally represented. The original PN does not have the notion of time for analysis of performance and dependability. The introduction of duration of events associated with PN transitions results in a timed Petri Net. A special case of timed Petri Nets is the Stochastic Petri Net (SPN) (MOLLOY, 1982), where the trigger times of the transitions are considered random variables with exponential distribution. A SPN can be translated to a CTMC, which may then be solved to get the desired performance or dependability results. This work adopts a particular extension, namely, SPN (MARSAN et al., 1994), which allows the association of stochastic delays to timed transitions using the exponential distribution, and the respective state space can be converted into Continuous Time Markov Chains (CTMC) (TRIVEDI, 2001).

Figure 2.4 depicts an example of an SPN model. Building a Markov model manually may be tedious and error prone, especially when the number of states becomes very large. SPN family of formalisms is a possible solution to deal with such an issue.

Places are represented by circles, whereas transitions are depicted as filled rectangles (immediate transitions) or hollow rectangles (timed transitions).

Arcs (directed edges) connect places to transitions and vice versa. Tokens (small filled circles) may reside in places, which denote the state (i.e., marking) of an SPN. An inhibitor arc is a special arc type that depicts a small white circle at one edge, instead of an arrow, and they usually are used to disable transitions if there are tokens present in a place. The behaviour of a SPN is defined in terms of a token flow, in the sense that tokens are created and destroyed according to the transition firings (GERMAN, 2000).

Immediate transitions represent instantaneous activities, and they have higher firing priority than timed transitions. Besides, such transitions may contain a guard condition, and a user may specificity a different firing priority among other immediate transitions. SPNs also allow the adoption of simulation techniques for obtaining dependability and performance metrics, as an alternative to the generation of a CTMC. Regarding SPN formal definitions and semantic, the reader is referred to (MARSAN et al., 1994).

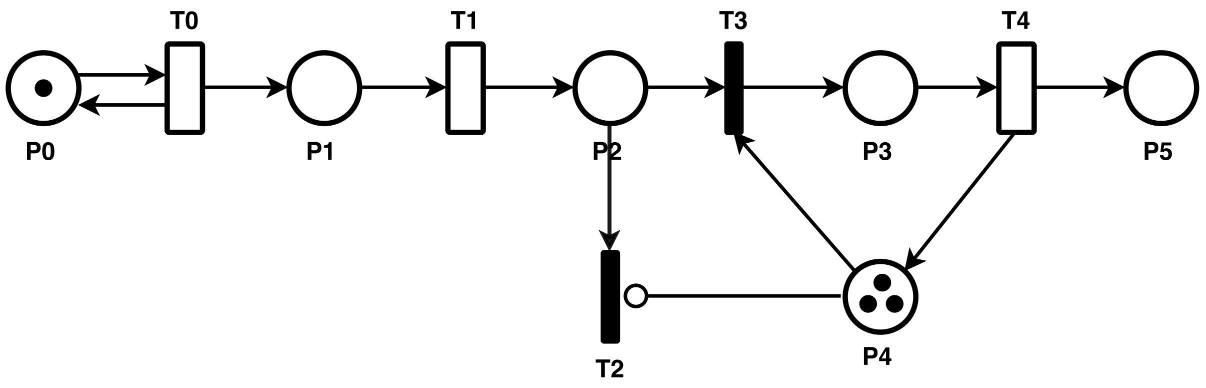


Figure 2.4: Example of a SPN model

2.2.4 Parametric Sensitivity Analysis

Parametric sensitivity analysis aims at identifying the factors for which the smallest variation implies the highest impact in model's output measure (FRANK, 1978; HAMBY, 1994).

The main aim of parametric sensitivity analysis is to predict the effect on outputs (measures) with respect to variations in inputs (parameters), helping to find performance, availability, or reliability bottlenecks (BLAKE; REIBMAN; TRIVEDI, 1988). There are many ways of performing parametric sensitivity analysis. Factorial experimental design (JAIN, 2008), correlation analysis and regression analysis (ROSS, 2010) are some well known techniques. The simplest method is to repeatedly vary one parameter at a time while keeping the others constant. When applying this method, a sensitivity ranking is obtained by computing the changes to the model output.

Another useful method, known as differential analysis, computes partial derivatives of measures of interest with respect to each input parameter. This method is considered as the backbone of many parametric sensitivity analysis techniques (HAMBY, 1994). The sensitivity of a given measure Y, which depends on a specific parameter θ, is computed as shown in Equation 2.1, whereas Equation 2.2 provides scaled sensitivity.

$$S_\theta(Y) = \frac{\partial Y}{\partial \theta}, \tag{2.1}$$

$$SS_\theta(Y) = \frac{\lambda}{Y}\frac{\partial Y}{\partial \theta}. \tag{2.2}$$

Specific methods for performing the differential sensitivity analysis in analytic models are needed when there is no direct closed-form equations for computing measures of interest, and finding their derivative expressions. Many papers have already described how to apply differential sensitivity analysis in a variety of analytic models, including CTMC (BLAKE; REIBMAN; TRIVEDI, 1988) (OU; DUGAN, 2003), Markov Reward Models (ABDALLAH; HAMZA, 2002), GSPN (MUPPALA; TRIVEDI, 1990), and Queuing Networks (YIN et al., 2007).

Partial derivatives are an important means of performing sensitivity analysis, but they may not be the proper technique in some cases. When the model under analysis has integer-valued parameters, the partial derivatives approach cannot be employed because it is designed for input values in a continuous domain. Other prohibitive condition is the usage of simulation instead of analytical solution for the models, because partial derivatives might not be computable due to the absence of closed-form equations or systems of equations to solve.

Sensitivity analysis may also be performed by calculating the percentage difference when varying one input parameter from its minimum value to its maximum value. Such an approach can be used an alternative for partial derivatives. Hoffman and Gardner (HOFFMAN; GARDNER, 1983) advocate the employment of each parameter's entire range of possible values to compute parameter sensitivities by means of percentage difference. Equation 2.3 shows the expression for this approach, where $max\{Y(\theta)\}$ and $min\{Y(\theta)\}$ are the maximum and minimum output values, respectively, computed when varying the parameter θ over the range of its n possible values of interest.

$$S_\theta(Y) = \frac{max\{Y(\theta)\} - min\{Y(\theta)\}}{max\{Y(\theta)\}}, \tag{2.3}$$

where

$$max(Y(\theta)) = max\{Y(\theta_1), Y(\theta_2), ..., Y(\theta_n)\}, \tag{2.4}$$

and

$$min(Y(\theta)) = min\{Y(\theta_1), Y(\theta_2), ..., Y(\theta_n)\}. \tag{2.5}$$

Another important method to assess the importance of each parameter is the analysis of a factorial experimental design. Design of Experiments (DoE) techniques can be used to determine simultaneously the individual and interactive effects of many factors that may affect the output measures (JAIN, 2008).

When dealing with hierarchical or composite models, the analysis needs to consider parameters from every model and determine their impact to the global measure of interest. Sensitivity indices for each submodel shall be computed and integrated in the computation of sensitivity of the complete model so we can obtain a unified sensitivity ranking. In (MATOS et al., 2014), a hierarchical heterogeneous model for mobile cloud computing is evaluated through a unified sensitivity ranking. Distinct sensitivity indices obtained through partial derivatives, percentage difference, and DoE are used in such analysis to provide a deeper comprehension of system characteristics and overcome specific limitations of each approach.

3
Auto-Scalable Private Cloud Environment

Many companies have been investing in private cloud platforms to make use of an existing infrastructure with full control, allowing greater flexibility in managing their IT services while ensuring the highest levels of privacy, confidentiality, and security they would possibly have with a public cloud (EUCALYPTUS, 2014a; SOTOMAYOR et al., 2009). We have evaluated a composite web application hosted in a private cloud with elastic mechanisms. The following sections of this chapter describe these systems and their mechanisms (or subsystems), their interactions and behaviour, i.e., their architectures.

3.1 Eucalyptus Private Cloud

We have chosen the private cloud platform Eucalyptus (Elastic Utility Computing Architecture for Linking Your Programs To Useful Systems) as environment for this work, due to the widely utilization by large companies that offer critical web services. Eucalyptus is an open source platform that provides IaaS leveraging existing hardware, and running behind the company's own firewall (EUCALYPTUS, 2014a). Eucalyptus API is strongly compatible with the Amazon Web Services (AWS) public cloud engine, enabling users to move workloads between AWS and Eucalyptus environments. The compatibility allows using tools, scripts, and images, in both AWS and Eucalyptus, thus taking advantage of ability to build a hybrid cloud. Eucalyptus is compatible with various Linux distributions including Ubuntu, Red Hat Enterprise, OpenSuse, Debian, Fedora, and CentOS. It is allows working with Xen, KVM and VMware ESX/ESXi hypervisors (EUCALYPTUS, 2014a,d).

Eucalyptus has a modular, distributed, and highly scalable architecture. It consists of five distinct components that are arranged in three layers, represented in Figure 3.1. These components are software and services that can be installed on the same physical server, or distributed among distinct machines, i.e., physical servers can be deployed in various architectures (EUCALYPTUS, 2014e; DANTAS et al., 2012).

In Figure 3.1, the top layer contains only two components. Cloud Controller (CLC) with Walrus controls all other components of the cloud. The CLC is responsible for presenting and managing a unified view of virtualized resources (servers, storage and network); it also exposes an administrative interface for managing the entire cloud infrastructure. Walrus provides storage for the Eucalyptus Machine Image (EMI), Eucalyptus Kernel Image (EKI), and the Eucalyptus Ramdisk Image (ERI) that are used to create a VM. Walrus provides persistent storage at file level, similar to Amazon Simple Storage Service (S3). Walrus provides persistent storage for both instances running within the cloud, and for programs running out of the cloud. Therefore, some customers have used the Walrus as a solution of SaaS separately, similar to the S3 (AMAZON, 2013; EUCALYPTUS, 2014e).

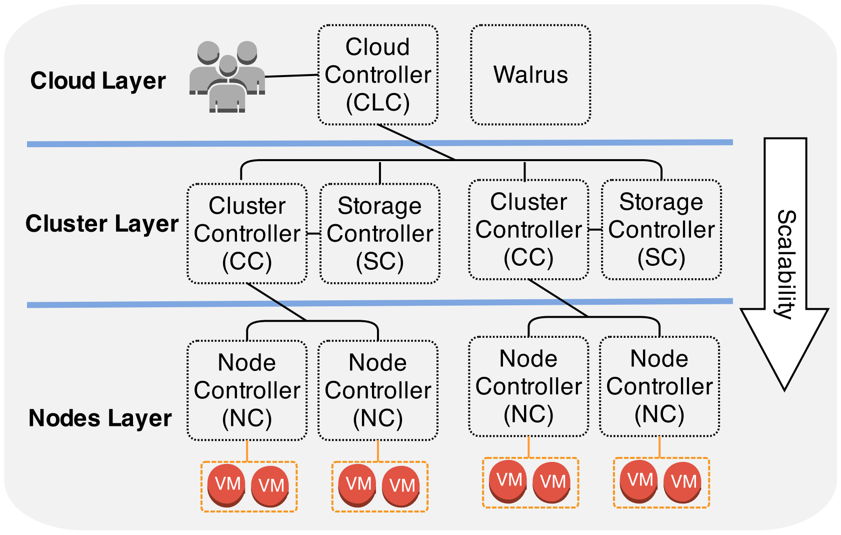


Figure 3.1: Conceptual representation of Eucalyptus architecture

The Cluster Controller (CC) manages services of a single cluster, and data resources of each physical node collection. Node Controller (NC) is a component installed in each physical machine that is meant to run instances of VMs. The NC communicates with a hypervisor (e.g.: KVM, Xen, VMware) to control VMs. A set of NCs is called a cluster and there may be several clusters (up to eight sets). Storage Controller (SC) builds and shares Eucalyptus Block Store (EBS) volumes, which are virtual block devices that provide persistent storage for VM instances (EUCALYPTUS, 2014e).

Eucalyptus Auto Scaling is a mechanism designed to handle applications that require adding and removing VM instances based on predefined thresholds of selected metrics (e.g.: CPU usage, number of user requests). Auto Scaling is particularly useful for applications that exhibit variability in use by hour, day or week. During demand peaks, the auto scaling mechanism increases the number of VM instances automatically to maintain the performance of the application hosted in the cloud. In a similar manner, when the demand decreases, the number of VM instances might be reduced to minimize costs and save physical resources (EUCALYPTUS, 2014d; AMAZON, 2014b). Eucalyptus Auto Scaling works in conjunction with CloudWatch and Elastic Load Balancing (ELB) mechanisms. The next section describe in detail the functioning of these mechanisms.

Eucalyptus Auto Scaling requires the configuration of three main components: Auto Scaling group, launch configuration, and scaling plan. An Auto Scaling group should contain all information of current VM instances, and specification of the minimum and maximum number of instances. A launch configuration has information that Auto Scaling needs to instantiate a VM. Scaling plan is a set of policies defining how and when Auto Scaling automatically acts in response to an alert from CloudWatch. In addition to operations defined in the scaling plan, Eucalyptus Auto Scaling monitors health of the instances periodically. If by chance an instance fails, a new instance is started (EUCALYPTUS, 2014d).

3.2 Scalable Composite Web Service Architectures

Based on the mentioned descriptions in before section, we defined the evaluation environment, i.e., private cloud platform, its components, its interfaces, interactions and settings. Therefore, we used an architectural configuration based on Eucalyptus platform, as an example to characterize the performance of elasticity mechanisms on a private cloud environment. However, the study of this work may be applied to other cloud infrastructures. We decided to use the Eucalyptus as the study environment for this paper, as large companies that offer critical web services use it widely. Another reason for us to have chosen Eucalyptus is that the elasticity mechanisms (e.g., auto scaling) were only incorporated in its latest 3.4 version, that is, few performance studies were applied until then.

The application evaluated here is an event recommendation mashup (MATOS; MACIEL; SILVA, 2013), i.e.: a composite web service, which is hosted on a VM of the Eucalyptus NC. This mashup receives the location (city or neighborhood) from the user and combines data from publicly available web services in order to recommend a musical event that will occur nearby.

Figure 3.2 depicts a UML activity diagram for such a service. The first activity is the search for musical events around the current location of the user. The location data might be acquired by communicating with a Global Positioning System (GPS) application, or manually provided by the user. After obtaining a list of nearby musical event, the application issues concurrent calls to two distinct services: search on venue statistics, such as average users rating of previous events in that concert place; and search for similarities between the lineup of artists in the event and the user's preferences.

When data from both services are acquired, the mashup selects the best event based on the venue and artist criteria. Once the event is selected, the application searches for map directions from the user current place to the event venue, and gets a link for one sample song from the main artist in that event. The last activity is the presentation of all gathered information to the user. We elaborated a CTMC submodel to represent an event recommendation mashup application, presented in the next chapter.

The mashup application must take advantage of elasticity mechanisms to avoid performance degradation even in sudden bursts of users requests. The elasticity also avoids wasting system resources in low workload periods. The Eucalyptus Auto Scaling mechanism is responsible for adapting the number of VMs that run the web service.

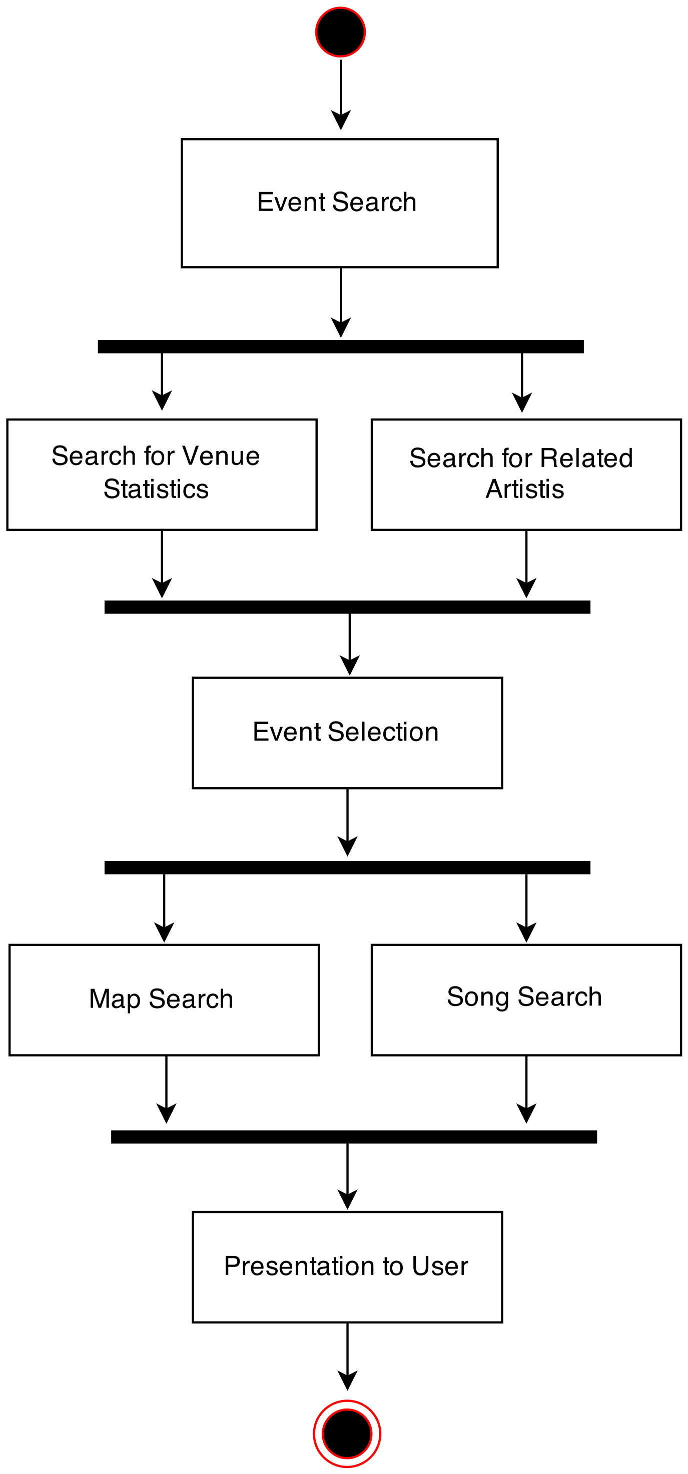
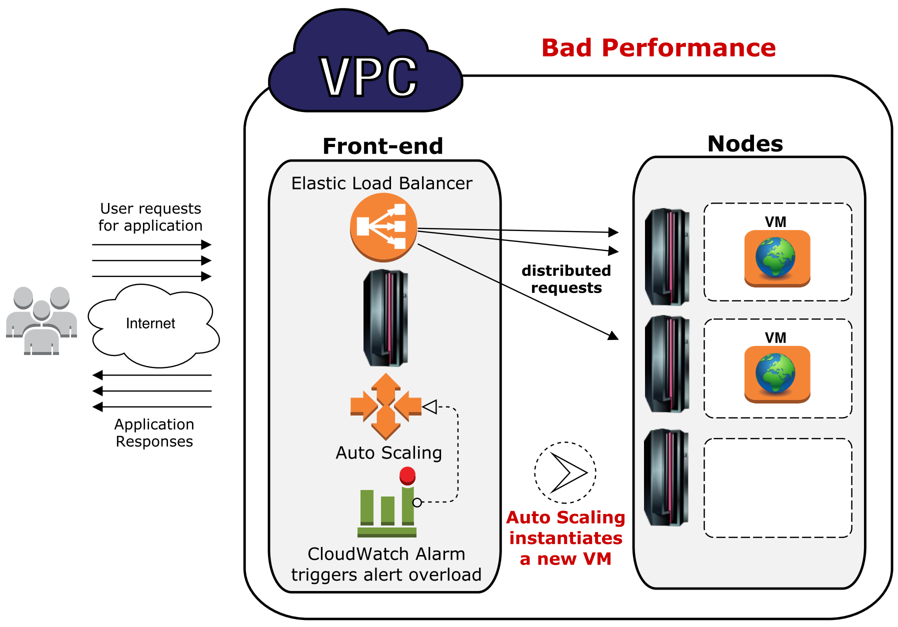


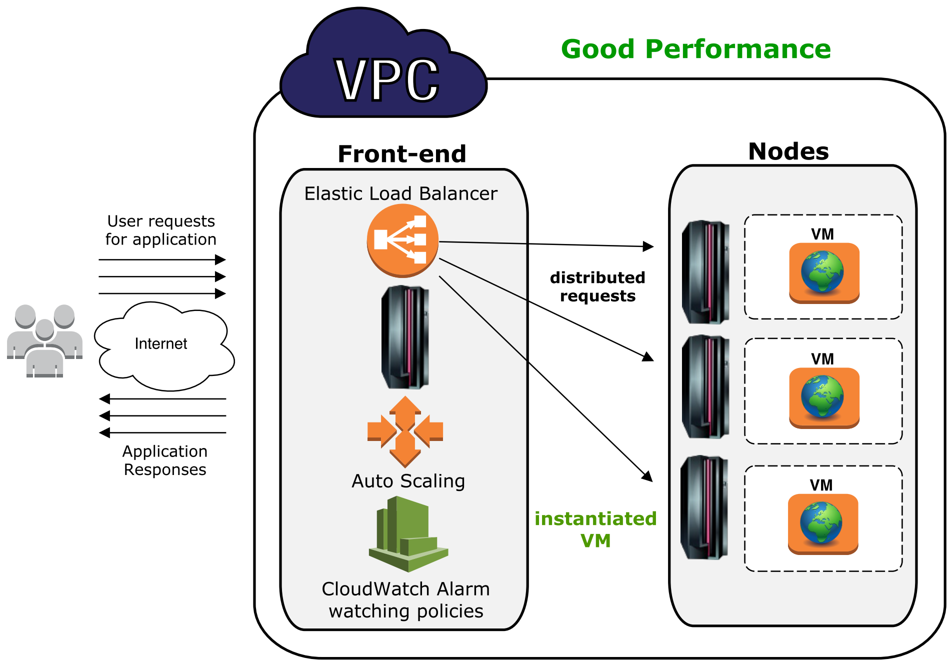
Figure 3.2: Event recommendation mashup architecture

The Auto Scaling interacts with the CloudWatch and ELB components to avoid performance degradations through the creation of new VM instances when a given metric reaches a threshold predefined by the system administrator. This architecture is depicted in Figure 3.3, the

ELB distributes client requests to the existing VMs of the target web application (mashup). The CloudWatch monitors established metrics (e.g.: average number of web requests per second, CPU or memory utilization, idle VMs, etc.) periodically (EUCALYPTUS, 2014d).



(a) CloudWatch triggers an alarm for Auto Scaling



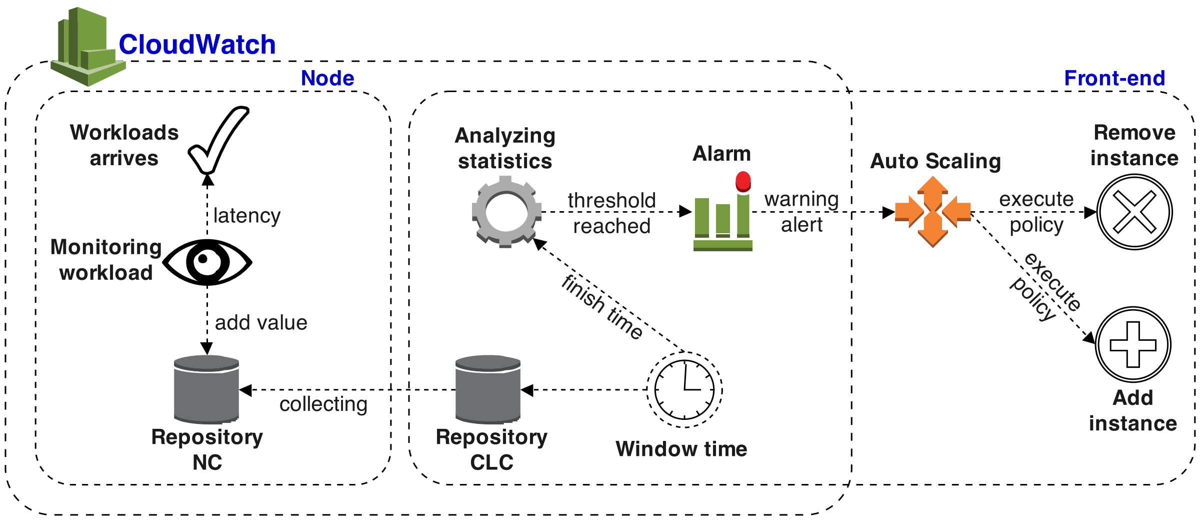
(b) ELB automatically distributes the requests considering the new VM

Figure 3.3: Scalable Web service architecture

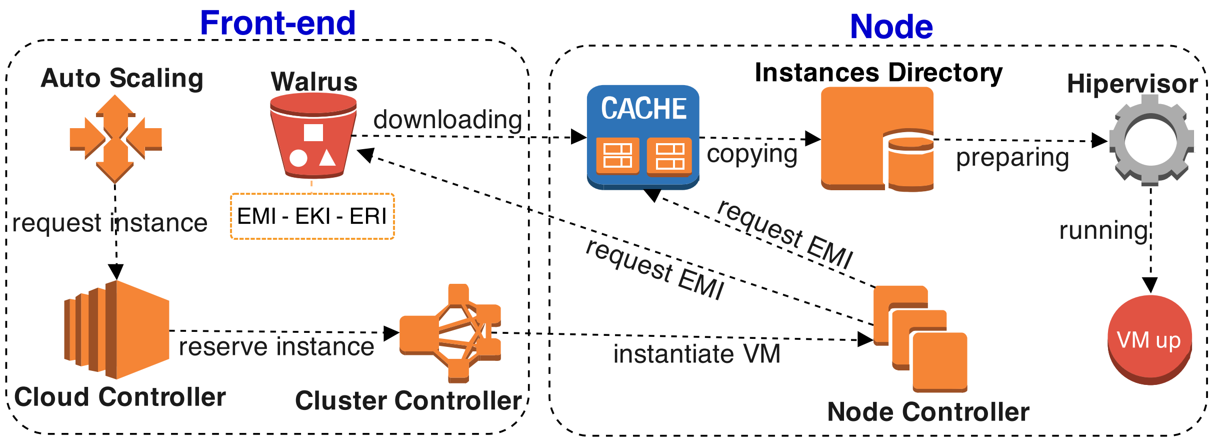
The CloudWatch service inserts data from monitored metrics at arbitrary intervals and extract statistics of the collected data for a particular time interval (time window), with a user-defined granularity (EUCALYPTUS, 2014f). Statistical information allows making operating and business decisions.

When a certain condition is met (e.g.: a poor performance metric), as it is illustrated in Figure 3.3 (a), the CloudWatch triggers an alarm for Auto Scaling, which instantiates one or more VMs. Shortly thereafter, as it is illustrated in Figure 3.3 (b), ELB automatically distributes the requests considering the new VM. The collaboration of these services results in a performance gain and allows an efficient usage of cloud resources, while it might also be used for fault tolerance purposes (EUCALYPTUS, 2014d). We elaborated a SPN main model to represent this auto scalable Web service, presented in the next chapter.

Figure 3.4 (a) details of the auto scaling monitoring (CloudWatch process). CloudWatch monitors information that is used by Auto Scaling to add or remove instances. Note that while workload arrives in a VM, the CloudWatch, after a certain latency, monitors a metric (e.g.: average CPU utilization of all VMs) and adds the value in the NC repository, along with the time stamp of data collection. This information is collected from repositories of all NCs each 5 minutes (by default), and sent to a unified repository on CLC that gathers data from all clusters.



(a) Auto scaling monitoring process architecture



(b) VM instantiation process architecture

Figure 3.4: Detailed representation of the auto scaling process architectures

At the end of a specified period of time (time window), CloudWatch aggregates the metric values from the CLC repository, which were added within the range of the time window. Statistics (e.g.: minimum, maximum, average) are computed from the aggregate data, and if the result reaches the specified threshold, CloudWatch Alarm would modify its state from *ok* to *alarm*. If an alarm state is maintained over a predefined number of time windows, the CloudWatch Alarm triggers the warning threshold for the Auto Scaling, which performs actions (i.e.: add or remove instances) based on policies previously determined (EUCALYPTUS, 2014f).

We consider the total time of auto scaling monitoring as metric of interest for the proposed design of experiment. There are some factors which may influence the performance this phase. These factors may be easily tuned, such as window time, collected period, and dimension (amount of VMs monitored). Therefore, in our experiments, the measurement of the CloudWatch (auto scaling monitoring) starts with the arrival of a workload that violates the predefined threshold, and finishes when the action was triggered (VM instantiation or termination), as detailed in Figure 3.4 (a). The auto scaling monitoring process was introduced in a part of the main model SPN, wherein each monitoring process represented a transition of the SPN model.

The VM instantiation is another important part of the auto scaling mechanism. Every request for the creation of a new VM instance takes some time to be fully serviced. That time depends on factors of the Eucalyptus instantiation process, which are explained as follows. As shown in Figure 3.4 (b), when the auto scaling mechanism —or a user — calls for a new VM instance, the CLC checks the existence of available resources for creating such a VM. This is accomplished through queries to the CC, which stores information about its nodes. If there are enough resources, the CLC reserves a unique identification number for the instance, the CC assigns the node where the VM should be instantiated, and the NC starts copying three images: EMI; EKI; and ERI. These images can be downloaded from Walrus or copied from a local cache, maintained by NC (EUCALYPTUS, 2014d).

The cache will not be used if it is not enabled in system configuration or if the requested EMI had never been instantiated on that node before. In the latter case, the CLC transfers EMI from Walrus to the cache and to the NC instance directory. Note that EKI and ERI are also downloaded if they are not in the NC cache. When the cache is not enabled, the EMI is transferred directly to the NC instance directory and is not cached for later use. When the cache is working and the node already has a copy of the EMI, the CLC does not download from Walrus, but only copies the EMI from the cache to the instance directory (EUCALYPTUS, 2014d).

After obtaining the EMI, EKI, and ERI, the NC interacts with the hypervisor (KVM, Xen, or VMware) to prepare the disk space required by the VM instance, according to the chosen VM type (e.g.: m1.small, c1.xlarge, etc.). Such a procedure usually requires creating, partitioning and formatting virtual block devices. The hypervisor, then, starts the current VM, completing the instantiation process (EUCALYPTUS, 2014d).

We identified three main phases that occur for instantiating a VM in a Eucalyptus cloud (CAMPOS ELIOMAR; MATOS; SILVA, 2015a): (i) resource and instance reservation; (ii) copy (or download) of the VM image files (EMI, EKI, and ERI); and (iii) VM preparation and deployment. We consider the time taken by each of these phases as metrics of interest for the proposed design of experiment, beyond the total instantiation time. There are some factors which may influence the performance of each phase, and subsequently the overall instantiation process. Some factors may be easily tuned or influenced by choices of cloud administrators and users, such as usage of cache, VM type, size of EMI, ERI, and EKI. These three phases are considered in a CTMC submodel for VM instantiation that is presented in next chapter.

Therefore, in this chapter, four architectures were defined to performance evaluation: (1) the architecture of the web service running with elastic mechanisms in a private cloud Figure 3.3, (2) the architecture of the mashup application service Figure 3.2, (3) the architecture of the VM instantiation mechanism Figure 3.4 (a), (4) and the architecture of the auto scaling monitoring mechanism Figure 3.4 (b). The following chapters describe the methodology, proposed models, and the case studies of the systems and mechanisms detailed here.

4
Methodology and Models

This chapter presents the hierarchical heterogeneous modeling proposed to evaluate the system described in Chapter 3. The main system modeled here is a composite web service running with elastic mechanisms in a private cloud. The hierarchical heterogeneous approach enables representing details of specific processes (subsystems) of the main system, such as VM instantiation, and the calls for the providers of specific web services that compose the mashup application. The hierarchical model comprises a SPN (MOLLOY, 1982; MARSAN; CONTE; BALBO, 1984) as main model and CTMC (KLEINROCK, 1975; BOLCH et al., 2001) as submodels of the main model. Before presenting the models, our approach methodology is described in next section, which contains a fluxogram with the sequence of activities that were carried out on the development this work.

4.1 Methodology

The methodology used to evaluate the performance of a composite web service running with elastic mechanisms in a private cloud, comprises the performing of five general steps, Figure 4.1: (1) understanding the systems and define the evaluation architectures, (2) design of experiments of the auto scaling mechanisms, (3) hierarchical heterogeneous modeling of the evaluation architectures, (4) models validation, and (5) performance evaluation with sensitivity analysis of the models and experiments. Every step of the methodology was performed related activities to be presented following:

- **Understanding the systems and define the evaluation architectures**: The first activity approaches the system to being solved, and understanding the mechanisms, their interactions and behaviour. In the previous chapter was presented the system and its mechanisms, in which four architectures were defined: (1) the architecture of the web service running with elastic mechanisms in a private cloud, (2) the architecture of the mashup application service, (3) the architecture of the VM instantiation mechanism, (4) and the architecture of the auto scaling monitoring mechanism.

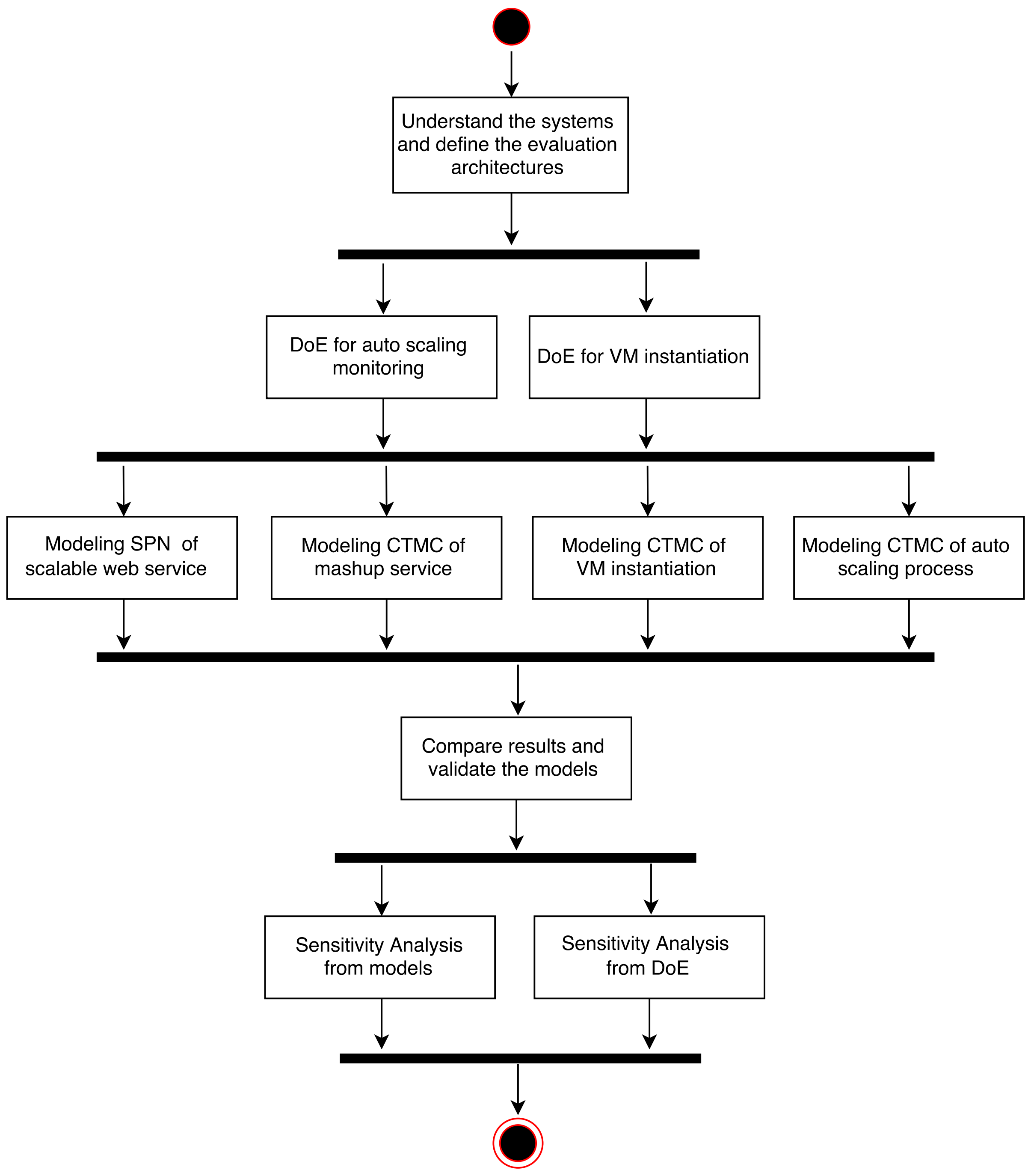

Understand the systems and define the evaluation architectures
DoE for auto scaling monitoring
DoE for VM instantiation
Modeling SPN of scalable web service
Modeling CTMC of mashup service
Modeling CTMC of VM instantiation
Modeling CTMC of auto scaling process
Compare results and validate the models
Sensitivity Analysis from models
Sensitivity Analysis from DoE

Figure 4.1: Methodology for performance evaluation of auto scaling in a private cloud

- **Design of experiments of the auto scaling mechanisms**: The design of experiments was also defined based on the first step. We performed various measurements (testbeds) of the time to complete the auto scaling process, i.e., measure the auto scaling monitoring time and VM instantiation time. These measurements provide values for the parameters and validation of the models proposed. The testbed step can be performed in parallel with the modeling step. Therefore, note that in Figure 4.1 the two steps are in sequence only for a better understanding of this work methodology.

- **Hierarchical heterogeneous modeling of the evaluation architecture**: From the evaluation architectures defined in the previous step, the models were elaborated. In this work, the modeling process adopts a heterogeneous hierarchical strategy that uses combinatorial models and state-based models to represent the performance features of the system. Therefore, for the main system, i.e., for the architecture the web service running with elastic mechanisms in a private cloud, we elaborate a SPN main model. For the subsystems of this main model (mashup application service, VM instantiation, and auto scaling monitoring) we elaborate CTMC models. These models will be presented in the next section of this chapter.

- **Models validation**: Through the metrics of the models and experiments, we compare the results and validate the models with a certain confidence interval. Therefore, our submodel provides consistent results to those observed in the experimental testbed, i.e., we verify the equivalence between the models and experiments.

- **Performance evaluation with sensitivity analysis of the models and experiments**: After validating the models, we can then analyze them. The models and design of experiments was used to perform a sensitivity analysis that compare a plethora of configuration scenarios in these systems. In other worlds, these models and measurements are used to analyze the impact of some factors on some metrics on different conditions.

4.2 SPN Main Model for Scalable Composite Web Service

The SPN, depicted in Figure 4.2, is a performance model of the web service deployed in a private cloud with the auto scaling mechanism. This model captures the main activities of the system, from client requests to service completion. It also represents the creation and termination of VM instances through the auto scaling mechanism. The architecture of this model was presented in previous chapter.

A token in place **PReq** represents a user request for the mashup. The firing delay of transition **TReq** corresponds to the mean time between arrivals of requests. When **TReq** fires, it stores one token in the place **PSend**, which denotes the transmission of requests through the network. The network latency between client and server is assigned to transition **TSend**. A token in place **JobAdmission** represents user request arrival in the cloud. Such a request may be admitted by the Load Balancer if its buffer is not full (immediate transition **TAccept**), or discarded otherwise (immediate transition **TReject**). If the request is admitted, it waits in the place **Queue** for being assigned to one of the VMs hosting the mashup application. The time spent by the Load Balancer to forward the request is represented by transition **TLB**. Notice that **TLB** requires one token from place **IdleVMs**, which initially has two tokens, denoting the number of available VM instances we defined for initial cloud setup.

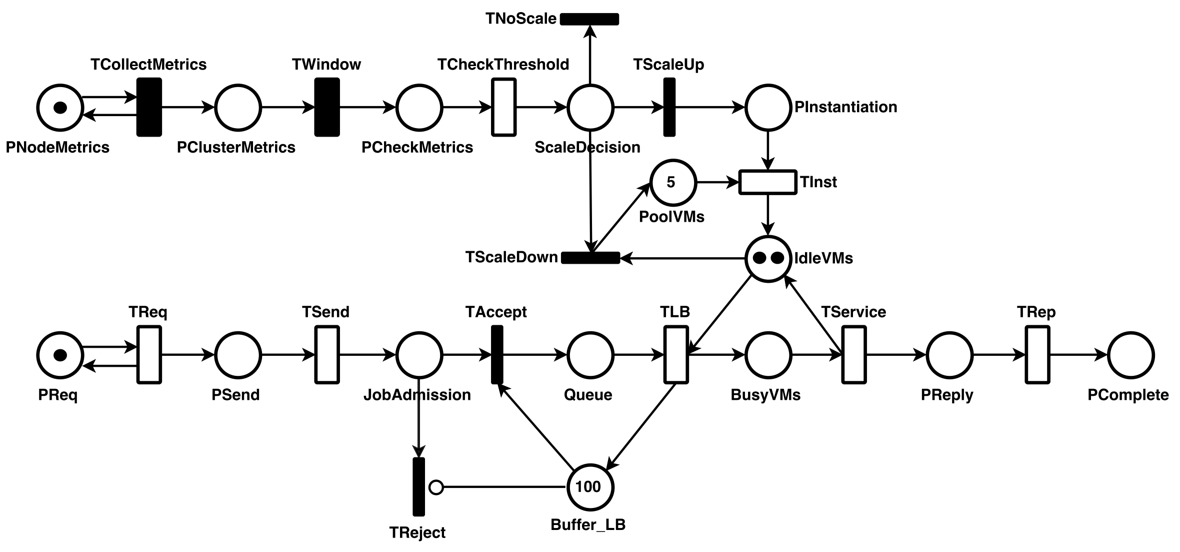


Figure 4.2: SPN model for the scalable web service on private cloud

The VMs that are busy processing a user request are represented by tokens in place **BusyVMs**. The time that one VM takes to serve a request is assigned to transition **TService**, which is refined by a CTMC submodel presented further. Notice that transition **TService** has an infinite server firing policy in order to properly represent the parallel execution of all requested VMs. After processing a request, a response is sent back to the client, this is represented by one token stored in the place **PReply**. This activity is denoted by transition **TRep**. Place **PComplete** represents the client response arrival.

The auto scaling mechanism is modeled by places and transitions in the upper part of the SPN. The place **PNodeMetrics** and transition **TCollectMetrics** denote the period that the CloudWatch collects the metrics results that are stored in the all NCs repositories. **TCollectMetrics** is a deterministic transition, so it properly represents the fixed time interval (collected period) at which the metrics are collected in the NCs and stored in the cluster repository, represented by tokens in place **PClusterMetrics**.

The place **PClusterMetrics** and transition **TWindow** denote the periodic trigger of CloudWatch monitor. **TWindow** is a deterministic transition, so it properly represents the fixed time interval (window) at which the CloudWatch waiting to start a summary of the cluster metrics data, and storing one token in place **PCheckMetrics**. The transition **TCheckThreshold** delay represents the time to summarize and analyze the metrics data of the cluster. But, only the data that were collected within the current time window, according to the predefined metric. **TCheckThreshold** stores a token in place **ScaleDecision**, which has three outgoing transitions: **TNoScale**, **TScaleDown**, and **TScaleUp** , that denote the options of holding, remove, or increasing the current number of VM instances, respectively. Table 4.1 presents the enabling functions for those three transitions.

If the transition **TScaleUp** fires, it consumes a token from **ScaleDecision** and stores a token in place **PInstantiation**. The transition **TInst** delay represents the time required for instantiating one VM in the private cloud. A CTMC submodel was developed to represent the VM instantiation process presented further in detail, hence the delay of **TInst** is computed from that submodel.

Table 4.1: Immediate transitions of the SPN model for scalable web service on private cloud

Transition	**Description**	**Enabling function**	Priority
TNoScale	Decision of keeping the current number of VMs	#IdleVMs≥1	1
TScaleUp	Decision of increasing the current number of VMs	#IdleVMs<1	1
TScaleDown	Decision of decreasing the current number of VMs	(#IdleVMs>4) AND (#Queue<1)	2
TAccept	Decision of accepting the user request	–	1
TReject	Decision of rejecting the user request	–	1

Notice that **TInst** requires one token available in the place **PoolVMs**. Such a place denotes the maximum capacity (in number of VMs) that might be added to the mashup application. For the current analysis, there are five tokens in **PoolVMs**, indicating that the auto scaling might create up to five new VMs. When transition **TInst** fires, it stores one token in the place **IdleVMs**. The immediate transition **TScaleDown** represents the activity of terminating VMs in periods of low workload, in order to avoid under utilization of resources. Table 4.1 shows the function that determines whether **TScaleDown** is enabled. Otherwise, if the transition **TNoScale** fires, consumes all tokens in **ScaleDecision**, keeping the number of VMs, i.e., without removing or adding VMs. Table 4.1 also shows the function that determines whether **TNoScale** is enabled.

Table 4.1 shows the enabling functions and priorities for each immediate transition of the SPN model. **TNoScale** is enabled whenever there is at least one token in the place **IdleVMs**. **TScaleUp** is enabled if there is less than one token in **IdleVMs**, i.e.: if the place is empty. **TScaleDown** requires more than four tokens in **IdleVMs**. Considering only the enabling functions, **TNoScale** and **TScaleDown** could be simultaneously enabled, but **TScaleDown** was assigned the highest priority of both, so it always fires first. The transitions **TAccept** and **TReject** do not have any enabling functions because they depend only on the existence of tokens in place **Buffer_LB**. There is an inhibitor arc in **TReject** coming from **Buffer_LB**, so **TReject** can only fire if **Buffer_LB** is empty. The arc from **Buffer_LB** to **TAccept** only enables transition **TAccept** if there is at least one token in **Buffer_LB**.

It is important to highlight that this SPN shall not be solved through numerical analysis, but only through simulation, due to the existence of non-exponential timed transitions (**TCollectMetrics** and **TWindow** are deterministic), in this case, we can characterize this model as a Deterministic and Stochastic Petri Net (DSPN). Next sections deal with the two CTMC submodels that also comprise our hierarchical heterogeneous modeling approach.

4.3 CTMC Submodel for Mashup Application

A CTMC submodel was created to evaluate the performance of the mashup application presented in Chapter 3. The CTMC input data are the response times of each individual service depicted in the UML diagram of Figure 3.2. Each state in the CTMC, depicted in Figure 4.3 denotes a service request, the only exception is the final state. The transition rates are estimated as the reciprocal of mean response time for each web service ($1/mrt_X$). All response times are assumed to be exponentially distributed. The state "**Event Analysis**" represents the execution of concurrent calls to the "Search for Venue Statistics" and "Search for Related Artists" services.

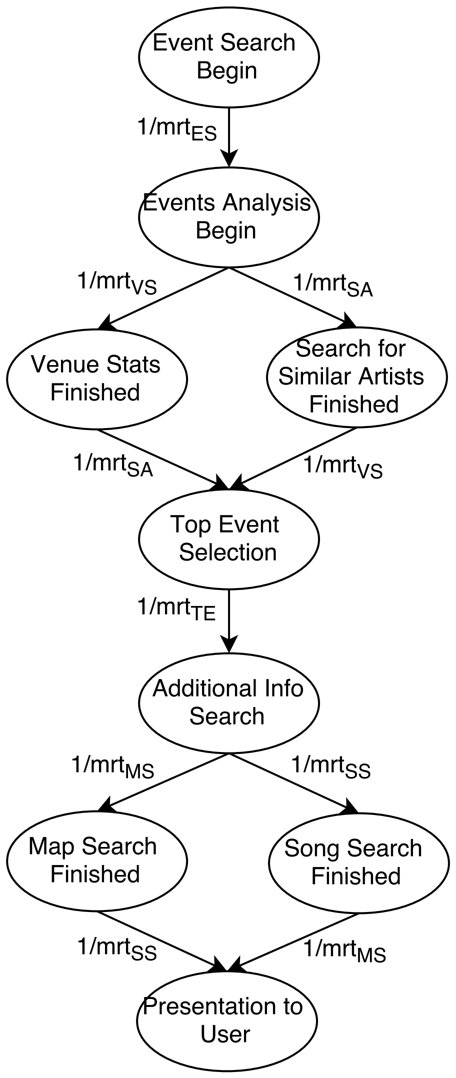


Figure 4.3: CTMC submodel for the event recommendation mashup

The submodel might go to state "**Venue Stats Finished**", with a rate of $1/mrt_{VS}$, or to "**Similar Artists Finished**", with a rate of $1/mrt_{SA}$, indicating which web service replied first. The state "**Top Event Selection**" indicates that the responses of both services, i.e.: "Search for Venue Statistics" and "Search for Related Artists" were received. "**Top Event Selection**" also denotes the analysis of all events using the previously collected data, which is completed with rate $1/mrt_{TS}$. Then the submodel goes to state **"Additional info search"**, representing the concurrent execution of queries to "**Map Search**" and "**Song Search**" services. The submodel reaches "**Map Search Finished**" with rate $1/mrt_{MS}$, and "**Song Search Finished**" with rate

$1/mrt_{SS}$, denoting which service replied first. After finishing both services, the CTMC finally reaches state "**Complete**", an absorbing state that indicates the end of mashup execution.

4.4 CTMC Submodel for VM Instantiation

Figure 4.4 depicts a CTMC proposed to represent the instantiation process of a VM in a Eucalyptus private cloud. As mentioned in the previous, this CTMC submodel was developed to represent in detail the transition **TInst** of the SPN main model, hence the delay of **TInst** is computed from this CTMC submodel. Therefore, **TInst** corresponds to VM instantiation process.

In Figure 4.4, the abstract submodel is composed of states **RI**, **CI**, **DI**, **PV**, and **VR**. They mean: **RI** = reserving instance on Cluster Controller; **CI** = copying EMI from cache to the directory of instances in Node Controller; **DI** = downloading EMI from Walrus to the Node Controller cache and directory of instances; **PV** = formatting of virtual block device and VM configuration by hypervisor, and finally, **VR** = VM running.

The CTMC submodel begins on **RI** state. From **RI** state, the VM instantiation process begins. If the EMI is already in the node's cache, the model goes to **CI** state with rate *pCache* $\times$ *(1/tRI)*. If the node needs to download the EMI from Walrus, the model goes to **DI** state with rate *(1-pCache)* $\times$ *(1/tRI)*. In **CI** state, the transition to **PV** state occurs with rate *1/tCI*. In **DI** state, the transition to **PV** state occurs with rate *1/tDI*. The last step of instantiation process occurs when the submodel goes from **PV** state to **VR** state with rate *1/tPV*, indicating that the VM is running.

Statistical analysis on experimental data does not reject exponential distributions as good fit for the time of every activity in VM instantiation process, except by *tDI*. The importance of the exponential distribution is based on the fact that it is the only continuous distribution that possesses the memoryless property (TRIVEDI, 2001). Therefore, the generic submodel can not be considered an actual CTMC, since the times of those two stages are not exponentially distributed. The submodel is actually a Stochastic Timed Automaton (STA). By means of poly-exponential refinements (i.e., phase-type distributions), we finally can transform this STA in a CTMC, using only exponentially distributed transition rates. For such a reason, we refined our submodel to represent *tDI* by means of poly-exponential distributions, by means of a moment matching method described in (WATSON J.F.; DESROCHERS, 1991).

In Figure 4.5, we have split the **DI** state in a hypoexponential distribution with 4 phases. The phases are denoted by **DI(1)**, **DI(2)**, **DI(3)**, and **DI(4)**, with rate $1/(tDI/4)$ for the output transition of each state. In Figure 4.5, we finally have a CTMC submodel with all exponentially distributed times. Therefore, this refined submodel will be used to perform additional analyses, presented in the next chapter.

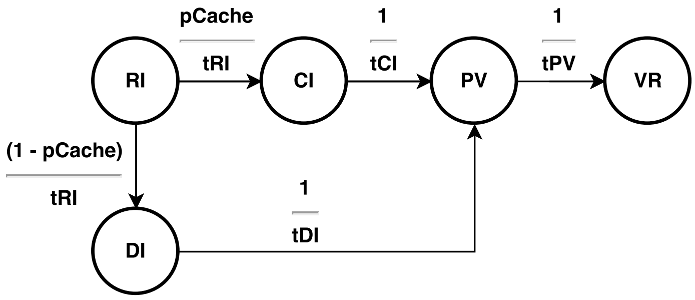


Figure 4.4: Generic submodel for performance evaluation of VM instantiation

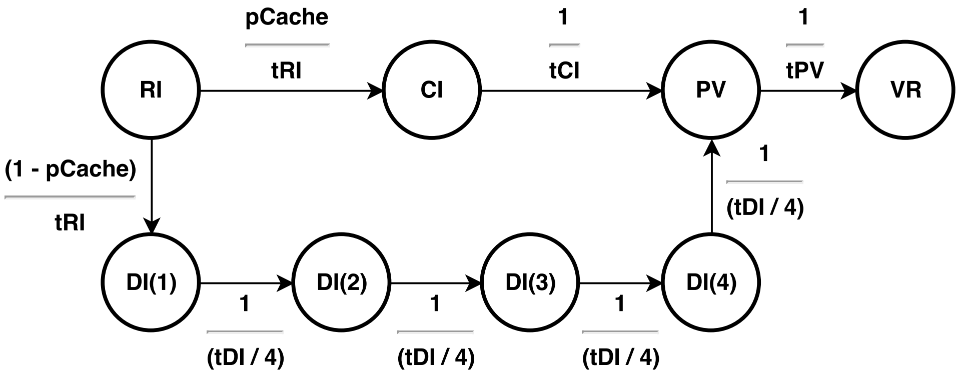


Figure 4.5: CTMC refined submodel with a hypoexponential distribution of 4 phases in **DI** state

4.5 CTMC Model for Auto Scaling Process

The auto scaling process starts with the arrival monitoring of a workload, and finishes with VM instantiation (when the arrival of a workload that violates the predefined threshold). VM instantiation is the final step of the auto scaling process, the auto scaling success also depends on the performance of the instantiation process. Figure 4.6 depicts a CTMC proposed to represent the auto scaling monitoring and the VM instantiation process in a Eucalyptus private cloud. This CTMC model was developed to represent some transitions the SPN main model, that corresponding to the entire scaling process, i.e.: **TCollectMetrics**, **TWindow**, and **TCheckThreshold** (corresponding to auto scaling monitoring), and **TInst** (corresponding to instantiation process).

It should be noted that this CTMC model can not be considered an actual submodel, since it does not represent only one transition the SPN main model. The intention is to represent all automatic provisioning process resource. From auto scaling monitoring until the completion of the VM instantiation. Through this CTMC model we can perform additional analyses, we carried out a performance evaluation and sensitivity analysis in all parameters of the auto scaling process.

In Figure 4.6, the abstract model is composed of states **AS**, **RI**, **CI**, **DI**, **PV**, and **VR**. They mean: **AS** = auto scaling monitoring (i.e., monitoring, detection, and action); **RI** = reserving instance on Cluster Controller; **CI** = copying EMI from cache to the directory of instances in Node Controller; **DI** = downloading EMI from Walrus to the Node Controller cache and directory of instances; **PV** = formatting of virtual block device and VM configuration by hypervisor, and finally, **VR** = VM running.

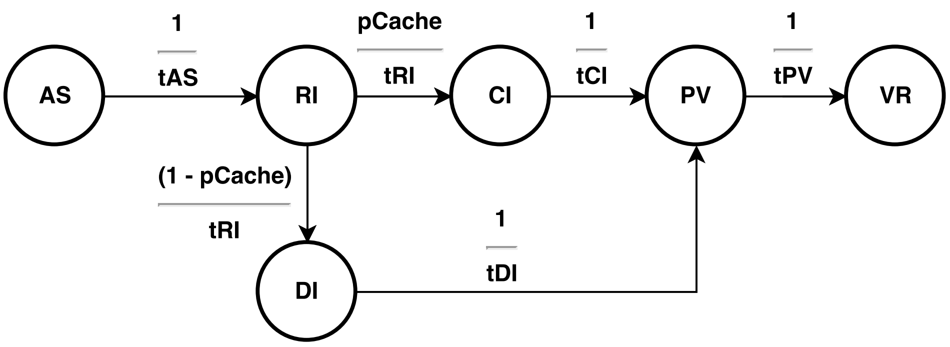


Figure 4.6: Generic model for the auto scaling performance

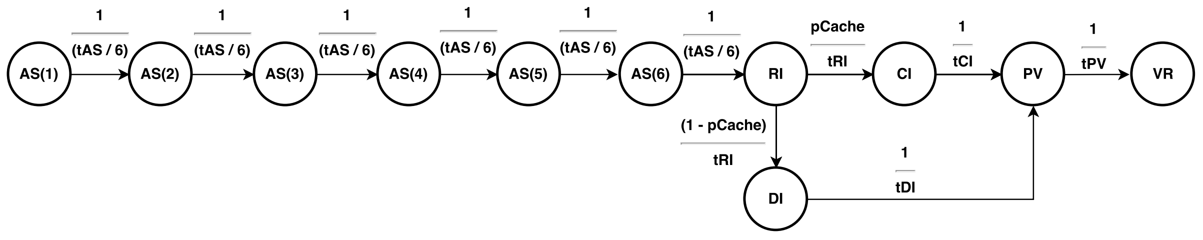
The model begins on **AS** state, where the auto scaling mechanism detects the threshold and triggers an action to instantiate a VM, then the model goes to **RI** state with *1/tAS* rate. From **RI** state, VM instantiation process begins. If the EMI is already in the node's cache, the model goes to **CI** state with rate *pCache* × *(1/tRI)*. If the node needs to download the EMI from Walrus, the model goes to **DI** state with rate *(1-pCache)* × *(1/tRI)*. In **CI** state, the transition to **PV** state occurs with rate *1/tCI*. In **DI** state, the transition to **PV** state occurs with rate *1/tDI*. The last step of instantiation process occurs when the model goes from **PV** state to **VR** state with rate *1/tPV*, indicating that the VM is running. Note the states that are part of the instantiation process, are the same states and rates shown in the instantiation submodel presented in the previous section.

Statistical analysis on experimental data does not reject exponential distributions as good fit for the time of every activity in auto scaling process, except by *tAS* and *tDI*. Therefore, the abstract model is actually a STA. By means of poly-exponential refinements, we finally can transform this STA in a CTMC, using only exponentially distributed transition rates. For such a reason, we refined our model to represent *tAS* and *tDI* by means of poly-exponential distributions, by means of a moment matching method described in (WATSON J.F.; DESROCHERS, 1991).

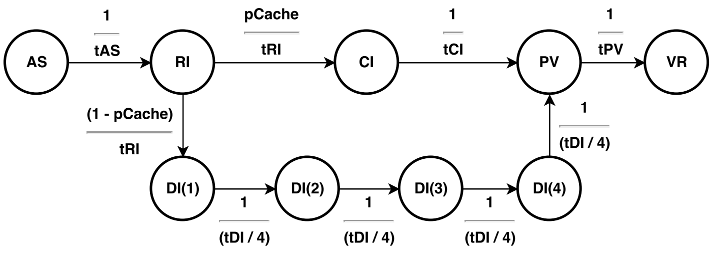
In Figure 4.7 (a), we have split the **AS** state outgoing rate in an Erlang distribution with 6 phases. The phases are denoted by **AS(1)**, **AS(2)**, **AS(3)**, **AS(4)**, **AS(5)**, and **AS(6)**, with rate $1/(tAS/6)$ for the output transition of each state.

In Figure 4.7 (b), we have split the **DI** state in a hypoexponential distribution with 4 phases. The phases are denoted by **DI(1)**, **DI(2)**, **DI(3)**, and **DI(4)**, with rate $1/(tDI/4)$ for the output transition of each state.

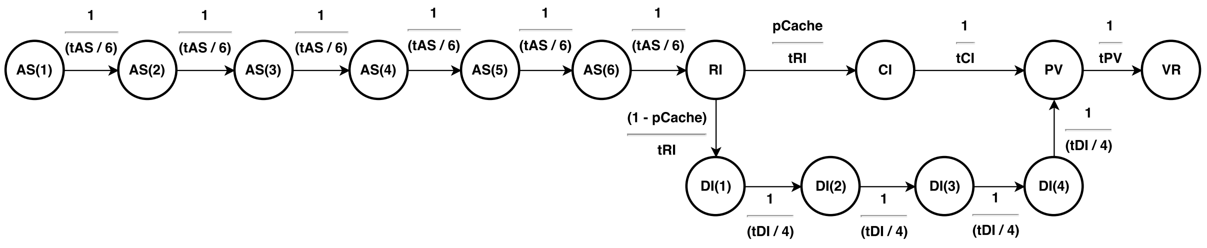
In Figure 4.7 (c), we finally have a CTMC submodel with all exponentially distributed times. This refined submodel will be used to perform additional analyses, presented in the next chapter.



(a) Model with a Erlang distribution of 6 phases in **AS** state



(b) Model with a hypoexponential distribution of 4 phases in **DI** state



(c) Refined CTMC model with all exponential distributions

Figure 4.7: Refined CTMC models by polyexponential distributions

5 Case Studies

This chapter presents the results of performance evaluation for the scalable web service system. In the first case study (Section 5.1) it was performed a parametric sensitivity analysis through design of experiments in a real private cloud testbed, for auto scaling monitoring and VM instantiation. We carried out most analyses through the stochastic models in order to predict metrics and compare a plethora of configuration scenarios in that system. We used the models to perform parametric sensitivity analyses, by changing the model rates, a factor at a time, and checking how the total response or execution time was affected. The two case study (Section 5.2) evaluate the performance of a composite web service, the second (Section 5.3) evaluate the performance of the instantiation process, the third case study (Section 5.4) evaluate the performance of the auto scaling process .

5.1 Case Study One

This case study shows the various measurements of the time to complete the auto scaling process, which includes the auto scaling monitoring time and VM instantiation time. These measurements are used to analyze the impact of some factors on the average time under distinct conditions. In addition, it provide values for the parameters of the proposed models and help on the validation of those models.

The experimental performance evaluation carried out in this work comprises 5 steps as shown in Figure 5.1. These steps correspond the activities that contemplate the design of experiment (DoE) phase present in the methodology of the previous chapter. The first step corresponds to the identification of main phases of the mechanisms evaluated. Next, it is important to define some details for planning the experimental design (metrics, factors and levels). The measurement tools and the workload used during experiments are further determined. The remaining steps are execution of experiments or data collection, and analysis of results.

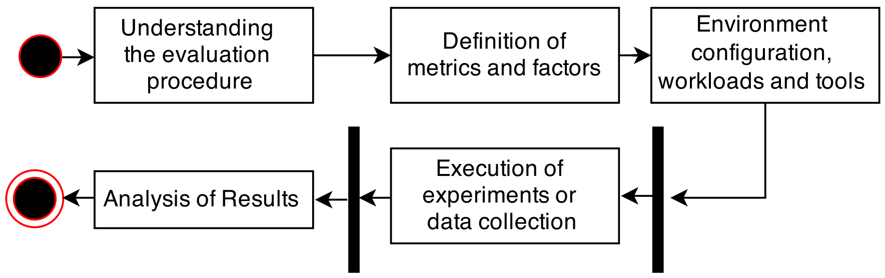

Understanding the evaluation procedure
Definition of metrics and factors
Environment configuration, workloads and tools
Execution of experiments or data collection
Analysis of Results

Figure 5.1: Activity diagram of the experimental evaluation

5.1.1 DoE for Auto Scaling Monitoring

In order to evaluate the auto scaling mechanism we measured specifically the time to complete the auto scaling monitoring throughout many experiment runs. In our experiments, each measurement started when the metric threshold was reached and finished when the action was triggered (VM instantiation or termination), as detailed in Chapter 3. This section presents the steps related to the experiment performed, and depicts the analysis of results obtained using general full factorial DoE technique.

In the second step of this case study (see Figure 5.1) we defined the metrics, factors and performance evaluation technique for the study of the auto scaling monitoring. As mentioned previous, before performing an experiment, it is necessary to identify the metrics, factors and their variables (levels) that might have a certain influence on behavior. Soon after, choose a appropriate experimental design technique. Therefore, first the metric of interest was identified as it corresponded to the total time elapsed between reaching the threshold and triggering actions. After the metric definition, we carefully chose the factors and levels to ensure that likely relevant aspects of the system were evaluated (GUIMARÃES; MACIEL; MATIAS JR, 2013; JAIN, 2008). Table 5.1 shows the adopted factors and their levels.

Table 5.1: Relevant factors and parameters

Factors	Levels
Collection Period (min.)	1, 5
Dimension (VMs)	1, 9
Window Time (min.)	1, 2, 4, 8, 16, 32

As previously explained, we have analyzed three factors: *dimension*, *window time*, and *collection period*. The *dimension* corresponds to the amount of VMs monitored. It is expected that the more VMs are monitored, the longer the time to detect desired metrics (workload). We have adopted 1 and 9 VMs as levels of the *dimension* factor.

Another considered factor was *window time* which is related to the period taken to observe a metric and store aggregate results according to established policy. If the aggregate result reaches a defined threshold, an action is performed by auto scaling based on established policy. Therefore, the monitoring *window time* may have some impact on auto scaling monitoring total time. The levels of such factor are divided into six periods: 1, 2, 4, 8, 16, and 32 minutes.

The factor *collection period* was one of the chosen factors because it is associated with the time interval that CloudWatch collects information from all NCs repositories. CloudWatch sends these pieces of information to the CLC repository in order to organize the information in all clusters using a single repository. With very short periods of monitoring window, it was observed that the factor *collection period* might influence the time to detect violated thresholds. Two levels were considered for *collection period*: 1 minute (default of the Eucalyptus) and 5 minutes.

After setting the metric, and levels of each factor. Finally we choose the experimental design technique. This study has several factors and more than two levels in one factor Table 5.1. Therefore, adopting the special case *General Full Factorial Design* (MONTGOMERY, 2012; JAIN, 2008).

General Full Factorial Design of Experiments was adopted to study the impact of each factor on the results. The reasons for adopting this technique in this study type have already been explained. But in an overview, this method helps to find viable and efficient settings that affect system performance, considering various measurement scenarios (GUIMARÃES; MACIEL; MATIAS JR, 2013; JAIN, 2008). Evaluation of systems that involve many factors and levels usually results in a significant number of experiments that might make the evaluation of results infeasible (JAIN, 2008). In our case, the total time and the effort required to execute a *General Full Factorial Design of Experiments* was not too excessive.

The whole this procedure of the DoE, is shown a general way in Figure 5.2. We performed this procedure by Minitab tool (MINITAB, 2013). First provide the input parameters (factors and levels). Then we chose the experimental design technique and analysis appropriate, in we case *General Full Factorial Design*. Finally Minitab generated every 24 sheet scenarios or combinations of all levels of the factors. In order to get results in an acceptable confidence level, each scenario was replicated 50 times, producing a total of 1200 experiments or measurements. Considering three factors and corresponding levels, there where 24 execution scenarios, which are described in Table 5.2.

Table 5.2: Scenarios of the auto scaling monitoring experiment

Factors	Scenarios																							
Collection period	1	1	1	1	1	1	1	1	1	1	1	1	5	5	5	5	5	5	5	5	5	5	5	5
Dimension	1	1	1	1	1	1	9	9	9	9	9	9	1	1	1	1	1	1	9	9	9	9	9	9
Window time	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32	1	2	4	8	16	32

After choosing the metrics, all factors with corresponding levels, and the experimental design technique. The next step was selecting workloads and tools to measure what it was planned.

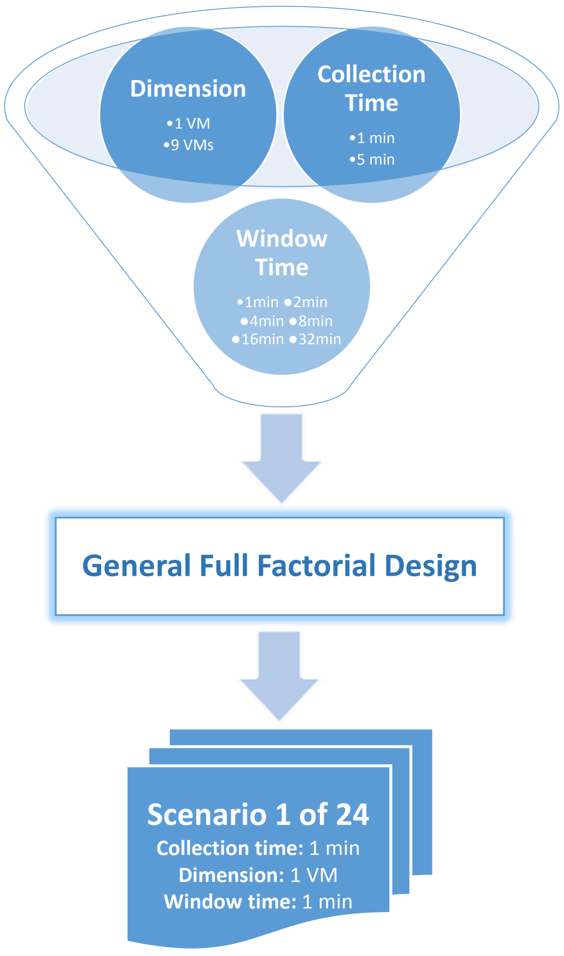


Figure 5.2: Overview of the experimental design for auto scaling monitoring

Figure 5.3 shows the components of our experimental environment. The cloud environment under test is fully based on the Eucalyptus framework and the KVM hypervisor. The environment setup includes installation and prior verification of the testbed to be used in experiments. In this phase, it was eliminated any type of internal or external interference which may influence the measurements. Some examples of these actions are: finishing unnecessary processes, disabling automatic operating system updates, and ensuring that the network is isolated from not involved computers.

We use four machines with the same hardware configuration: Intel(R) Core(TM) i7-3770 3.4 GHz CPU, 4 GB RAM DDR3, 500 GB SATA HD. These one machine was configured as front-end for execution of CLC, CC, SC, and Walrus. The other three run the NCs. The four machines run CentOS Linux 6 operating system with ext4 filesystem, and the Eucalyptus 3.4 platform. A 10/100 Mbps Ethernet network was adopted to connect the PCs through a single switch.

Preliminary tests with Eucalyptus auto scaling were performed to verify if the environment was working as expected, i.e., results of monitored metrics were checked and properly accounted for one or more instances. We also checked if the tasks were triggered according to

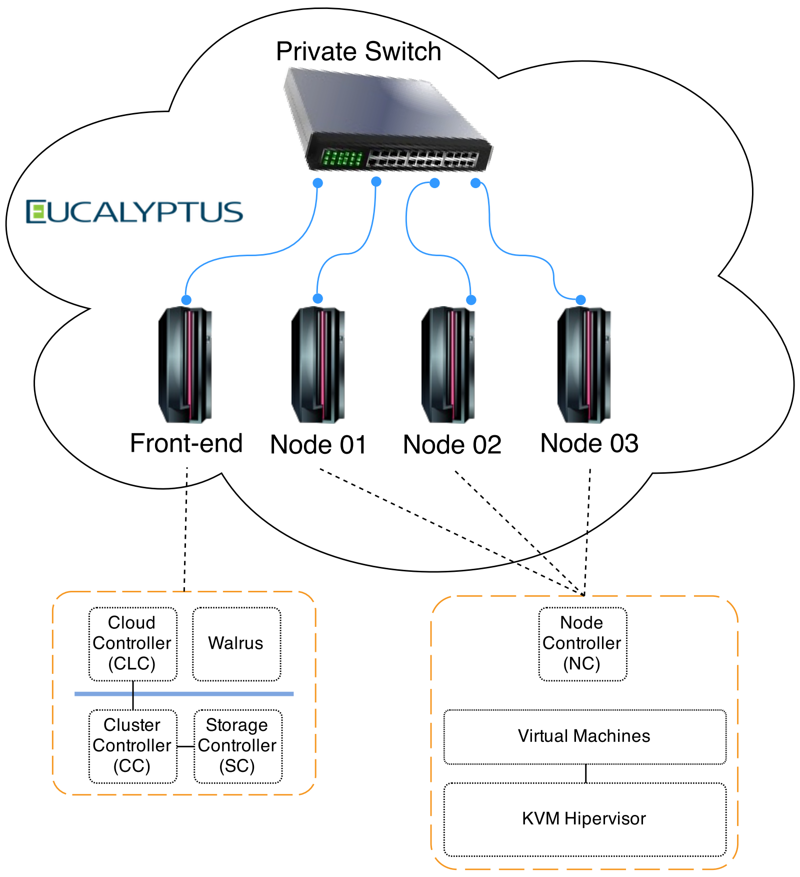


Figure 5.3: Components of the auto scaling monitoring environment

defined policies. At the end of this phase, we considered the environment controlled and ready to perform the experiments.

We created a software script to inject synthetic workloads that made the VMs reaching the defined CPU utilization threshold, forcing auto scaling triggering actions, whereas registering the time spent by auto scaling monitoring (detection and action). The workload cycle was implemented by means of Shell script language functions (Bash – Bourne-again Shell) that perform the operations we have just mentioned. The Figure 5.4 represents the workload cycle performed by this script. Each function may be seen below:

- **Verify Status Function**: To generate the workload and start measuring, we must ensure that are running only the desired number of VMs. This is possible with the *euscale-set-desired-capacity* command, by parameter *scale -c* we specify the number of VMs that auto scaling mechanism should work. Furthermore, we need to check if all the VMs are not with pending status with *euca-describe-instances* command. Then, we need to check if the CloudWatch Alarm it's with *OK* status, i.e., not its ALARM status with *euwatch-describe-alarms* command. Finally, after we certify that the environment was controlled, we can invoke the function that will start the workload. This function is represented by a Shell script that is shown below:

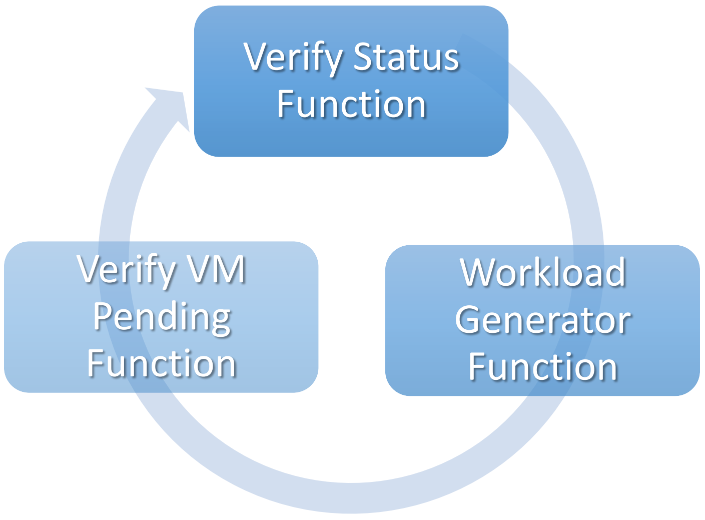


Figure 5.4: Workload cycle illustration of the auto scaling monitoring.

```
verifyStatus(){
   euscale-set-desired-capacity scale -c 1
   v=FALSE
   while [ $v = FALSE ] do
     statusVm=`euca-describe-instances | grep pending | wc -l`
     statusAlarm=`euwatch-describe-alarms | grep ALARM | wc -l`
    if [ $statusVm -eq 0 ] && [ $statusAlarm -eq 0 ] then
       workloadGenerator
       v=TRUE
    fi
   done
}
```

- **Workload Generator Function**: We define the scaling policies that the auto scaling mechanism must instantiate a VM when the level of CPU utilization achieve 40 % (threshold). The metric that we measure corresponds to the interval between the breach of the threshold and the beginning of VM instantiation. Therefore, we use the Lookbusy application (LOOKBUSY, 2013) to generate a synthetic load suddenly. With the *lookbusy -c 50* command, suddenly we generate a CPU utilization level of 50 %. For this reason, this function marks the start time of the metric in the same moment as the workload is generated. After generating the workload, is invoked the *verify VM pending function* that checks when the scaling policy is applied, i.e., when a VM starts the instantiation process. By invoking this function, the start time of the metric is passed as parameter, in order to be used in the calculation of metric duration. This function is represented by a Shell script that is shown below:

```
workloadGenerator(){
    beginTimeThreshold=$(($(date +%s%N)/1000000))
    ssh -i /home/frontend/Downloads/access.pem 192.168.0.171 "
       lookbusy -c 50" &
    verifyVmPending $beginTimeWorkload
}
```

- **Verify VM Pending Function**: The previous *workload generator function* marks the initial time that the threshold is reached. Then by *euca-describe-instances* command, this *verify VM pending function* continuously checks whether the scaling policy is triggered, i.e., identifies the time that arises some VM with pending status. By identifying the pending status, this time is marked. Known initial time of the threshold, and the action initial time of the scaling policy, is calculated the metric duration. Finally, the time of the metric is stored in a log file. In order to control the environment to a new measurement, the synthetic load generator is finalized through the *killall -9 lookbusy* command. This function is represented by a Shell script that is shown below:

```
verifyVmPending(){
   booting=FALSE
   while [ $booting = FALSE ] do
     statusVm=`euca-describe-instances | grep pending | wc -l`
     if [ $statusVm -gt 0 ] then
       beginTimeVmPending=$(($(date +%s%N)/1000000))
       durationMetric=$(($beginTimeVmPending-$beginTimeThreshold))
       echo $durationMetric >> timess.txt
       ssh -i /home/frontend/Downloads/access.pem 192.168.0.171 "
          killall -9 lookbusy"
       booting=TRUE
     fi
  done
}
```

After that, the process of the synthetic load generator was finalized. The script kills the VM instantiated, and waited a random time to request a new measurement, i.e., start the cycle of functions again. For each scenario this cycle was repeated 50 times, according to the experimental design (see previously section). The time interval required between a cycle and another, depends on the random time, which is described in the complete script in Appendix A.

The experiment was performed in May 2014. As seen in Table 5.1, we had two levels for the factor *collection period*, two levels for the factor *dimension*, and 6 levels for the factor *window time*, resulting in 24 scenarios. In order to get results in an acceptable level of confidence, each scenario was measured 50 times, producing a total of 1200 experiments.

Table 5.3 shows the total average time in minutes, standard deviation, and coefficient of variation for each scenario of the auto scaling monitoring. For the sake of conciseness, in the first column header the factor *collection period* was shortened to **C**, the factor *dimension* to **D**, and the factor *window time* to **W**.

Table 5.3: Results of Each Scenario the auto scaling monitoring

C (min) D (VMs) W (min)	Mean (min)	Std. Deviation	Coef. of Variation
1 1 1	188.6	0.148	0.0008
1 1 2	211.9	28.94	0.1366
1 1 4	222.2	29.98	0.1349
1 1 8	327.4	178.1	0.544
1 1 16	457.5	398.4	0.8707
1 1 32	790.8	872	1.1026
1 9 1	188.6	0.336	0.0018
1 9 2	215	30.12	0.1401
1 9 4	274.2	91.8	0.3348
1 9 8	341	195.1	0.5721
1 9 16	492.9	334.8	0.6792
1 9 32	911.8	900	0.9869
5 1 1	788.6	193.6	0.2455
5 1 2	777.5	328.1	0.4221
5 1 4	747.9	157.2	0.2102
5 1 8	596	334.1	0.5605
5 1 16	678.2	284.4	0.4194
5 1 32	795.8	189.8	0.2385
5 9 1	862.2	361.9	0.4197
5 9 2	814.9	100.1	0.1229
5 9 4	755.9	132.5	0.1727
5 9 8	648	305	0.4708
5 9 16	688.2	292.5	0.4251
5 9 32	860.6	217.2	0.2524

Observing Table 5.3, it should be noted that the total average time for auto scaling monitoring is at least 3 times lower when the *collecting period* (**C**) is changed from 5 (Eucalyptus default time) to 1 minute, considering *window time* (**W**) of 1, 2, and 4 minutes. For **W** values between 8 and 32 minutes, this difference is not so significant. Therefore, the **C** factor was very important for system performance, but its effect is influenced by the **W** factor.

The auto scaling monitoring time increased linearly with the changes in **W** when **C** is 1 minute. On the other hand, for a **C** of 5 minutes, the auto scaling monitoring time has not a monotonic behavior regarding the changes in **W**. Thus it is noticeable the existence of some interaction between **C** and **W** factors, and their importance for the auto scaling monitoring time. Focusing on scenarios where the factor *dimension* (**D**) was 9 VMs, we did not perceive a significant difference in time compared with 1 VM.

These results and analyses are not intended to point out the best scenario, since such a conclusion would be questionable, because it depends on other system conditions. Nevertheless, the results are useful to guide cloud administrators to configure parameters of their systems. Additional statistical analyses of the effect and relevance of each factor are presented, and aid at drawing more accurate and detailed conclusions.

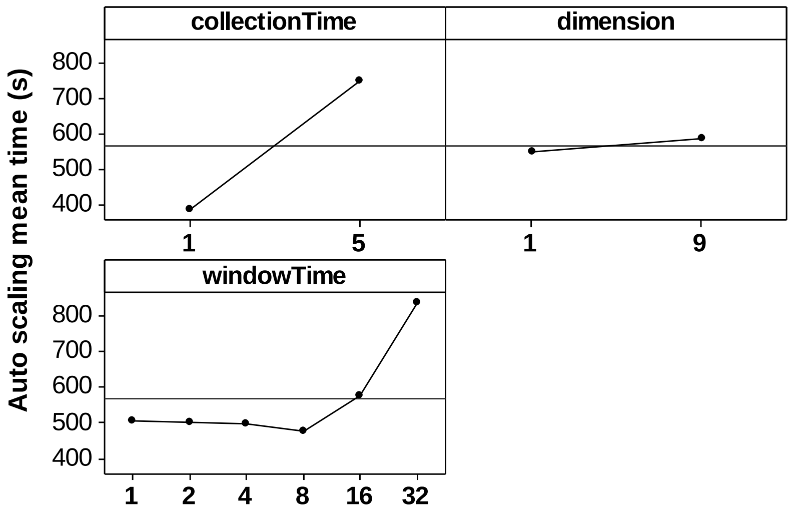
The Table 5.4 presents the effects and relevancies computed for each factor from the DoE analysis. For the sake of conciseness, the factors are represented by letters: *collection period* = **A**, *dimension* = **B**, and *window time* = **C**. We adopted in columns the initials DF for degrees of freedom. The MS (mean square) is calculated dividing the sum-of-squares by degrees of freedom. The factors which most affect the performance are **A**, generating a performance effect with relevance 84.224 %, followed by **C** with 7.913 %, and **AC** interaction with 6.743 %. The factor **B** with 0.969 % and their interactions (**AB**, **BC**, **ABC**) did not have significant relevance, since **AB** got 0.002 %, **BC** had 0.075 %, and **ABC** had 0.070 %.

Table 5.4: Effects and estimated relevance to the average total time for auto scaling monitoring

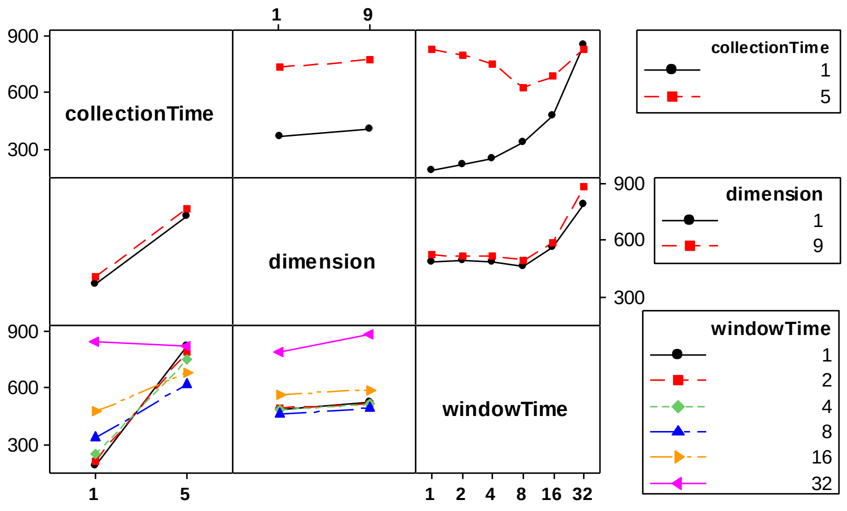
Factors	DF	MS	F-ratio	Relevance	P-value
A	1	40184930	356.31	84.224 %	0.000
B	1	462250	4.10	0.969 %	0.043
C	5	3775777	33.48	7.913 %	0.000
AB	1	899	0.01	0.002 %	0.929
AC	5	3217698	28.53	6.743 %	0.000
BC	5	36498	0.32	0.075 %	0.899
ABC	5	34277	0.30	0.070 %	0.911
Error	1176	112780			

Such a relevance in percentages was calculated based on F-ratio or F statistical values; therefore, each result has been obtained in accordance with the proportion of each F-ratio to the sum of them. The F-ratio is calculated as the ratio of mean square for each factor by the mean square error. The F-ratio can be understood as the level of impact of each factor on system performance according to the average variation among its levels. It is also worth noting that the P-values of the interactions **AB**, **BC**, and **ABC** were above 0.05 (the significance level for this study), i.e., there is not sufficient evidence that these interactions have a significant effect on performance improvement. The P-value is calculated based on degrees of freedom and F-ratio, which are in their turn calculated from the Analysis of Variance (ANOVA) table (JAIN, 2008).

The graph of main effects, Figure 5.5 (a), shows the isolated factors and the effects of their different levels. Since the difference between mean levels was significant, this highlighted the *collection period* and *window time* as high impact factors, as evidenced by the slopes in lines of both factors. The *window time* had high impact only from levels 8 to 32, because from 1 to 8 the differences between means were not significant. On other hand, the *dimension* factor alone had a much smaller effect, represented by an almost horizontal line.



(a) Main effects plot



(b) Interaction effects plot.

Figure 5.5: Main plot and interactions plot for factors effects

The analysis of factor effects one by one is prone to misinterpretation, so it is also important to analyze the interaction between factors. Figure 5.5 (b) shows the effects of such interactions. The interactions that have no parallel lines suggest a significant impact on the measure of interest, i.e., total time of auto scaling monitoring. This is the case of upper right and lower left grid cells, which represent the interaction between *collection period* and *window time*. It is interesting to observe the distinct behaviors for *collection period* 1 and 5 on the upper right cell. Note that for a *collection period* of 1, the auto scaling monitoring time increased whenever the *window time* increased, i.e., it had a monotonical behavior. For a *collection period* 5, the line decreased for window periods from 1 to 8, and enhanced between 8 and 32. The line did not have a constant pattern, making it difficult to draw accurate conclusions. Thus, a *collection period* of 1 minute is desirable instead of 5 minutes, due to both, lower times for the auto scaling monitoring, and a more predictable behavior for changes in the *window time* levels.

Also, it is important to highlight that according to Figure 5.5 (b), all interactions with factor *dimension* had parallel lines, confirming the analysis shown in Table 5.4. In other words, those were interactions without major influences on measured results, and therefore did not impact performance.

5.1.2 DoE for VM Instantiation Process

Applications running on cloud environments, and using elasticity features, are designed to create or terminate VM instances according to the current workload level. Such a behavior avoids the waste of idle resources (e.g.: memory, CPU, disk space, power) in periods of low load, whereas enables the fast increase of computational power when facing a burst of high load. On this work, we use an Eucalyptus-based setup as an example to characterize the performance of the instantiation process in a private cloud environment, but the problem and the evaluation may be extended to other cloud infrastructures (CARON et al., 2012).

In order to evaluate the process of VM instantiation specifically, we adopted a case study, measuring the time taken by each phase considered important. Next, we analyze the results obtained using the *full factorial DoE technique*. The experimental performance evaluation carried out in this work was performed in 5 steps as shown previously in Figure 5.1. These steps aim to analyze the impact of some factors in the mean time to complete the instantiation of a VM under distinct conditions. We shall obtain accurate measures to analyze the process of VM instantiation properly. This performance analysis is helpful to improve many systems which often instantiate new VMs.

The first step corresponds to the identification of main phases of the VM instantiation process, as detailed in Chapter 3. This section presents the steps related to the experiment performed, and depicts the analysis of results obtained using *full factorial DoE technique*. Based on the experimental design, we present the measurement scripts and the workload used. Next, the remaining steps: execution of experiments and analyzes of results are shown.

We can define an experiment as a test or series of tests, which are conducted by researchers in many fields of knowledge in order to discover something about a particular system or process. Before performing an experiment, it is necessary to identify the metrics, factors and their variables (levels) that might have a certain influence on behavior. Soon after, choose a appropriate experimental design technique. In the second step performance evaluation methodology we define the metrics, factors and performance evaluation technique for the study of the instantiation process.

First, we identified the metrics of interest from the study of the instantiation process. Those measures are associated with the relevant phases of the creation of a VM instance (previous section). We adopted four metrics:

1. **Instance reservation time** – The time for the CC to reserve the instance;
2. **EMI copy time** – The time to copy the EMI, EKI, and ERI (shortened here as "copy time of EMI") to the instance directory of the NC, from Walrus or cache depending on the scenario;
3. **VM preparation time** – The time that the hypervisor takes to prepare and start the VM;
4. **Total time** – The total time of instantiation, which is the sum of the previous three measures.

After the definition of metrics, we carefully chose factors and their levels to assure that the system will be evaluated in a relevant way (GUIMARÃES; MACIEL; MATIAS JR, 2013; JAIN, 2008). Table 5.5 shows the factors and their levels.

Table 5.5: Factors and levels of the VM instantiation

Factors	Levels
Cache	yes, no
VM type	m1.large, m3.xlarge, cc1.4xlarge
EMI size (GB)	2, 5, 8

We consider the *cache* as a factor to be analyzed, since the instantiation time is expected to be higher when the cache is not used, due to differences between remote copy throughput (network-bounded) and local copy throughput (hard disk-bounded). Such an issue was also observed in preliminary tests, confirming the need to carefully analyze this factor.

Another factor considered was the *type of VM instance*, which is related to the time of preparation of the VM by the hypervisor, and therefore might have some impact on this instantiation stage. The choice of three VM types (levels) intends to encompass very different requirements of CPU, RAM and disk resources. Table 5.6 describes the resources required by each type of VM analyzed. The type *m1.large* requires 2 CPU cores, 10 GB of disk space, and 512 MB of RAM. The type *m3.xlarge* requires 4 CPU cores, 15 GB of disk space, and 2048 MB of RAM. The type *cc1.4xlarge* requires 8 CPU cores, 60 GB of disk space, and 3072 MB of RAM. Those VM types were based on the Eucalyptus documentation (EUCALYPTUS, 2014e), which associates the types of VMs in 6 groups: 1) general purposes; 2) computer optimization, 3) optimization of memory; 4) storage optimization; 5) micro; and 6) graphics processing. According to our limitation of computational resources, and intended to encompass very different requirements CPU, RAM and disk resources, we opted for select two types from group 1 (m1.large and m3.xlarge), and one type from the group 2 (cc1.4xlarge).

Table 5.6: Types of VM instances chosen

Type	CPU cores	Disk (GB)	Memory (MB)
m1.large	2	10	512
m3.xlarge	4	15	2048
cc1.4xlarge	8	60	3072

The factor *EMI size* was chosen because it is associated with the download of EMI from Walrus to the NC cache, and with the copy to the directory of instances on NC, as explained previously. The sizes considered in this study are: 2 GB, 5 GB and 8 GB, based on our experience of most commonly used operating system images and size of installed applications. These values were also chosen considering that they would not exceed the disk size of the smallest instance type (m1.large), which is 10 GB.

The *factorial experiment* has several variations or special cases that depend on the number of factors and levels of each factor. As seen in Table 5.5, this study has several factors and more than two levels in one factor. Therefore, adopting the special case *Full Factorial Design* (MONTGOMERY, 2012; JAIN, 2008). Although, when evaluating systems that involve many factors and their levels, the number of experiments increases enough to hinder the evaluation of the results (JAIN, 2008). Not we chose to perform a *Fractional Factorial Design*, because in our case the total time and effort needed to run a *Full Factorial Design* is not prohibitive and the cost of analysis not was high.

We adopt a *Full Factorial Design* to obtain the desired measures and to study the impact of each factor on those measures. The *Full Factorial Design* helps in finding the configuration settings that enhance the system performance, considering various measurement scenarios (GUIMARÃES; MACIEL; MATIAS JR, 2013; JAIN, 2008). This phase aims at investigating the effects and relevance of the factors, and also to characterize the times of each instantiation phase.

The whole this procedure of the DoE, is shown a general way in Figure 5.6. We performed this procedure by Minitab tool (MINITAB, 2013). First provide the input parameters (factors and levels). Then we chose the experimental design technique and analysis appropriate, in we case *Full Factorial Design*. Finally Minitab generated every eighteen sheet scenarios or combinations of all levels of the factors. In order to get results in an acceptable confidence level, we decided to run 50 replicas for each scenario, yielding a total of 900 experiments or measurements. Considering the three factors, and the levels selected for them, there are eighteen scenarios to run, which are described in Table 5.7. For the sake of conciseness, in the scenario column, the cache factor is referenced as **Y** (working) or **N** (not working). The VM factor is denoted as **10** (m1.large), **15** (m3.xlarge), or **60** (cc1.4xlarge), where the numbers represent the size of the disk allocated for that VM type. The EMI size is simply represented by the number of gigabytes: **2**, **5** or **8**.

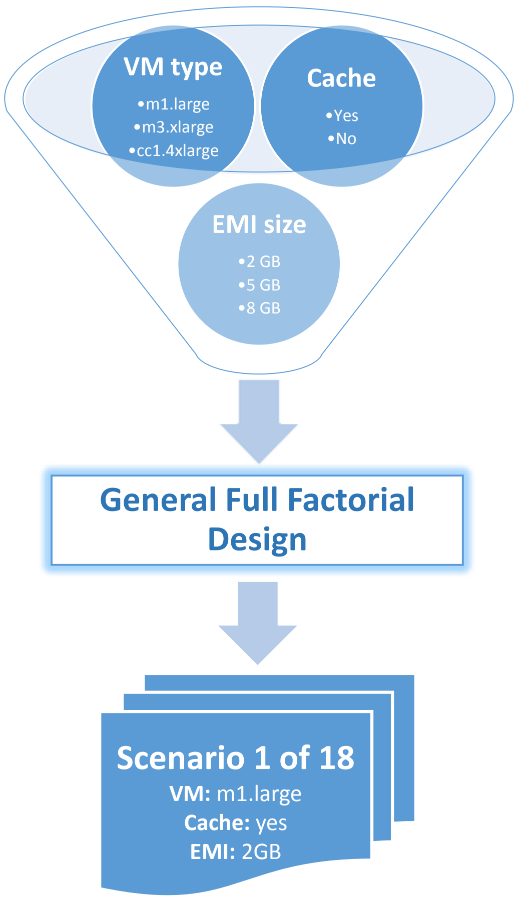


Figure 5.6: Overview of the experimental design for instantiation process.

Table 5.7: Scenarios of the instantiation experiment

Factors	Scenarios																	
Cache	Y	Y	Y	Y	Y	Y	Y	Y	Y	N	N	N	N	N	N	N	N	N
VM	10	10	10	15	15	15	60	60	60	10	10	10	15	15	15	60	60	60
EMI	2	5	8	2	5	8	2	5	8	2	5	8	2	5	8	2	5	8

Another reason to adopt the *Full Factorial Design*, is that enables see the interactions effects between factors. Which would not have with a parametric sensitivity analysis varying the parameters at a time, study also adopted from models.

After choosing all metrics, factors and experimental design technique for the study of the instantiation process. The next step was the creation of a testbed and selection of workloads and tools to enable us measuring what we planned so far.

The environment configuration comprises the setup and preliminary verification of the testbed system that will be used in experiments and measurements. At that stage we remove any sort of external or internal interference that may influence the measurements. Examples of such actions are: finishing unnecessary processes, disabling operating system automatic updates, and assuring that the network is isolated from computers not involved.

Figure 5.7 shows the components of our experimental environment. The cloud environment under test is fully based on the Eucalyptus framework and the KVM hypervisor. The environment was assembled with two machines of same hardware configuration: Intel(R) Core(TM) i7-3770 3.4 GHz CPU, 4 GB of RAM DDR3, 500 GB SATA HD. It is important to highlight that a cluster with many machines is not necessary for the purposes of this study, since the VM instantiation is a process involving only the front-end and the specific node where the VM is allocated. Therefore, the usage of only these two machines enabled accurately monitoring every stage of the instantiation process. One machine is configured as the front-end, running the CLC, CC, SC, and Walrus. The other machine runs the NC. Both execute the Linux CentOS 6 operating system with ext4 filesystem, and Eucalyptus platform 3.4.0.1. The VMs run the Linux Ubuntu Server 14.04.01 LTS operating system, with ext4 filesystem. We use a 10/100 Mbps Ethernet network to connect the PCs through a single switch.

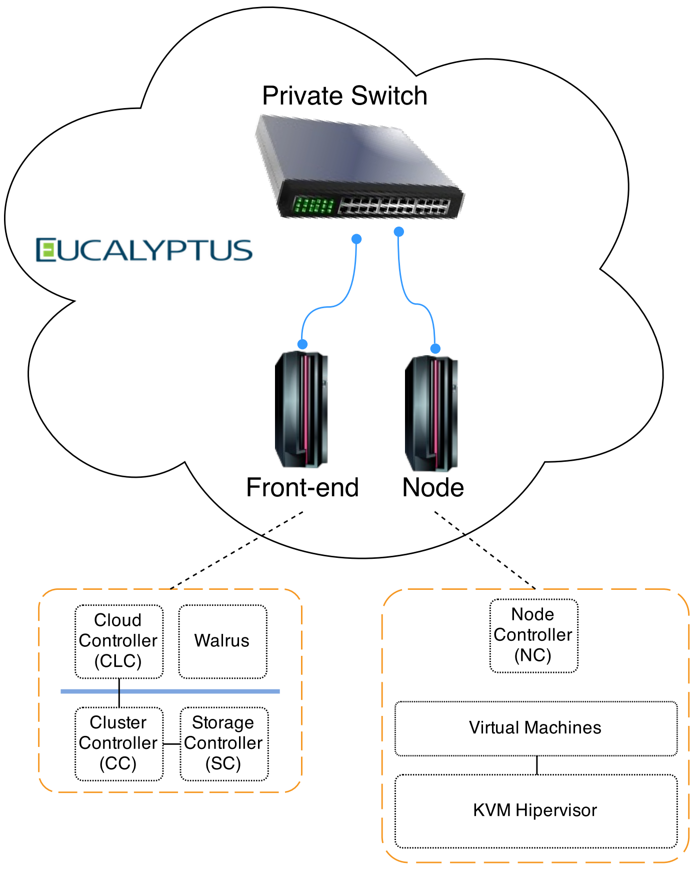


Figure 5.7: Components of the VM instantiation environment

Preliminary tests of VM instantiation were conducted to verify whether the environment was working as expected, i.e., the requests of VM instantiation were properly executed. The cache of VM images was populated by the node, etc. To disable the cache in no cache scenarios, first we had to clear the cache. To clean out the cache, we terminated all running instances on the NC (since the cache is used as part of a copy-on-write clone of the disk image of the instance) first. Then run the *rm -rf* command everything in *$INSTANCE_PATH/eucalyptus/cache* directory of the NC. After we clear the cache, we had to set the parameter *NC_CACHE_SIZE* and *NC_WORK_SIZE* in the file *eucalyptus.conf* on the NC. Any value below 10 will disable caching. The parameters *NC_CACHE_SIZE* and *NC_WORK_SIZE* are set by default, which causes the NC to pick values based on disk space available when NC is first started.

At the end of that phase, we consider the environment is controlled and ready to carry out the experiments.

We created a software script to instantiate the VMs repeatedly, and collect the times of each phase of the instantiation, i.e., each chosen metric. The workload cycle was implemented by means of Shell script language functions (Bash – Bourne-again Shell) that perform the operations we have just mentioned. The Figure 5.8 represents the workload cycle performed by this script. In each function is associated with one of the metrics (reserve instance, copy the EMI, and VM preparation) as it may be seen below:

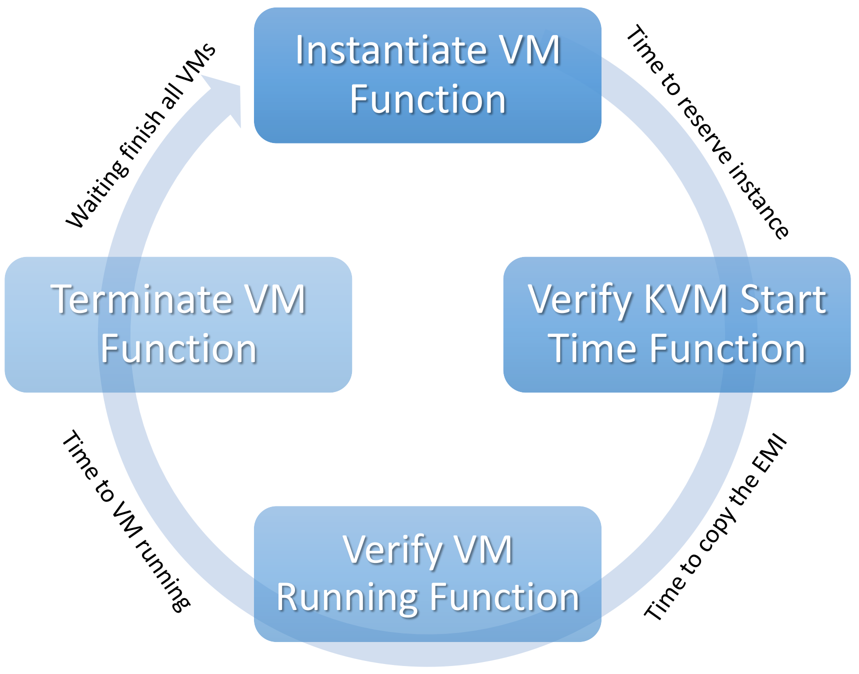


Figure 5.8: Workload cycle illustration of the VM instantiation process

- **Instantiate VM Function**: This function uses *euca-run-instances* command to instantiate only one VM at a time. The VM type, the cluster, and the image (EMI) are passed as parameters to this command. The start and end time of the *reserve instance* metric is marked before and after the *euca-run-instances* command. Soon after, the duration to reserve instance in CC is stored in a log file. Finally the *verify KVM start* *time function* is invoked to check the duration of the *copy the EMI* metric. The final time to reserve instance is marked as the start time to copy the EMI by Walrus to NC, this time is passed as argument to *verify KVM start time function*. This function is represented by a Shell script that is shown below:

```
instantiateVm(){
  beginTimeReserveInstance=$(($(date +%s%N)/1000000))
  euca-run-instances -t m3.xlarge -z CLUSTER01 -k default emi-4
      F4B3CE2
  endTimeReserveInstance=$(($(date +%s%N)/1000000))

  duration=$(($endTimeReserveInstance-$beginTimeReserveInstance))
  echo "Duration Reserve Instance by CC: "$duration >> log_times.txt

  verifyKvmStartTime $endTimeReserveInstance
}
```

- **Verify KVM Start Time Function**: This function is responsible for measuring the duration of *copy the EMI* metric. First, checks whether the *qemu-kvm* process started in NC through the *ps aux* command, which is run remotely via *ssh* command. The KVM uses *qemu-kvm* process to prepare and start a VM. When the *qemu-kvm* process starts, means that EMI has been copied or downloaded to the NC instances directory. Therefore, if identified the *qemu-kvm* process, its start time is marked. Subtracting the start time of *qemu-kvm* by the final time to reserve an instance (passed by the previous function), we have the duration of *copy the EMI* metric. The duration of *copy the EMI* metric is then stored in a log file. Finally the *verify VM running function* is invoked to check the duration of the *VM preparation* metric. The start time of *qemu-kvm* process is passed as parameter for the *verify VM running function*. This time will be used to calculate the duration of the *VM preparation* metric. This function is represented by a Shell script that is shown below:

```
verifyKvmStartTime(){
  c=FALSE
  while [ $c = FALSE ] do
    filter=`ssh 192.168.0.5 ps aux | grep  /usr/libexec/qemu-kvm | wc
        -l`
      if [ $filter -eq 1 ] then
          beginTimeKvm=$(($(date +%s%N)/1000000))
          duration=$(($beginTimeKvm-$endTimeReserveInstance))
          echo "Duration Copy the EMI to NC: "$duration >> log_times.
              txt
          c=TRUE
      fi
  done

  verifyVmRunning $beginTimeKvm
}
```

- **Verify VM Running Function**: This function checks the time it takes the KVM to prepare and start a VM, i.e., *VM preparation* metric. The algorithm performs numerous times the *euca-describe-instances* command, in order to verify a VM with the running status. If a VM is identified with the running status, this time is marked. The duration of the VM preparation metric is calculated by subtracting the start time of the running status, by the start time of the *qemu-kvm* process (passed by the previous function). finally, the *terminate VMs function* é invoked. This function is represented by a Shell script that is shown below:

```
verifyVmRunning(){
   booting=TRUE
   while [ $booting = TRUE ]
   do
     running=`euca-describe-instances | grep running | wc -l`
     if [ $running -gt 0 ]
     then
       beginTimeVmRunning=$(($(date +%s%N)/1000000))
       duration=$(($beginTimeVmRunning-$beginTimeKvm))
       echo "Duration VM Preparation by KVM: "$duration >> log_times.
           txt
       booting=FALSE
     fi
  done

  killVM
}
```

- **Terminate VMs Function**: This function uses *euca-describe-instances* command to find out which instances are running in the cloud and terminate them all with *euca-terminate-instances* command. This function is represented by a Shell script that is shown below:

```
killVM(){
   instances=`euca-describe-instances | grep INSTANCE | awk '{print
       $2}'`
   euca-terminate-instances $instances
}
```

It is important to highlight that the workload was one VM at a time, i.e., no concurrent VM instantiations were done. After the creation and execution of an instance was completed, the VM was terminated, and the script waited a fixed amount of time to request a new instantiation. For each scenario this cycle was repeated 50 times, according to the experimental design (see previously section). The time interval required between a cycle and another, depends on the time taken to terminate all the VMs with running status. The complete script can be seen in Appendix B.

The experiment was performed in November 2013. According to the design adopted, we have 2 levels of *cache* factor, 3 levels of *VM type* factor, and 3 levels of *EMI size* factor, leading to 18 scenarios or combinations. In order to get results in an acceptable confidence level, we decided to run 50 replicas for each scenario, yielding a total of 900 experiments. This amount of experiments might be considered large, but most of the actions (e.g., workload generation, and data collection) are automatically executed, so reducing the required effort and time for running the experiments.

Table 5.8 provides the average time in milliseconds measured for each phase of the instantiation process – instance reservation, copy of the EMI, and VM preparation by the hypervisor – and also the total time. The standard deviation and coefficient of variation for the total time is also presented.

Table 5.8: Results of each scenario of the experiment

Scenario (Cache VM EMI)	Instance Reserved (ms)	EMI Copied (ms)	VM Prepared (ms)	Total (ms)	Std. Deviation	Coef. of Variation
Y 10 2	280	7624	10603	18506	10002	0.5405
Y 10 5	279	7152	11416	18848	10020	0.5316
Y 10 8	271	7251	10216	17739	9844	0.5549
Y 15 2	306	7221	13266	20793	10107	0.4861
Y 15 5	306	7257	13235	20797	10085	0.4849
Y 15 8	318	7543	14115	21976	9874	0.4493
Y 60 2	329	7169	22120	29618	118	0.004
Y 60 5	314	7329	16764	24407	8918	0.3654
Y 60 8	307	7297	14813	22417	9736	0.4343
N 10 2	342	194790	14538	209670	118	0.0006
N 10 5	359	472928	16253	489540	1667	0.1026
N 10 8	357	750593	14182	765132	8326	0.0109
N 15 2	390	196716	12546	209651	109	0.0005
N 15 5	354	472524	16635	489513	85.5	0.0002
N 15 8	363	753203	13958	767525	7282	0.0095
N 60 2	405	206374	14017	220796	9996	0.0453
N 60 5	401	487273	13273	500947	9830	0.0196
N 60 8	409	767982	13949	782340	7477	0.536

Observing Table 5.8, it shall be noticed that the total time to instantiate a VM is at least 10 times higher when the cache is not used, comparing to the scenarios which use the instance cache. Such a difference indicates that this factor is of utmost importance for the system performance. Focusing in the scenarios when the cache is used (denoted by **Y**), those ones with VM type 10 are the three best configurations, but their standard deviations indicate that the difference with respect to VM types 15 and 60 might not be significant. For the scenarios without cache, the

EMI size plays an important role, since the instantiation time increases more than 100 % when the size increases from 2 GB to 5 GB, or more than 50 % when the size increases from 5 GB to 8 GB, while the other factors are kept unchanged.

These results and analyses are not intended to point out the best scenario ever, since such a conclusion might depend upon other specific conditions and requirements of the system being used. Anyway, those results are helpful to guide a cloud administrator in adjusting the parameters of his system while being aware of possible impacts on performance of applications, mainly when using mechanisms such as auto scaling. Additional analyses of the effect and relevance of each factor are presented.

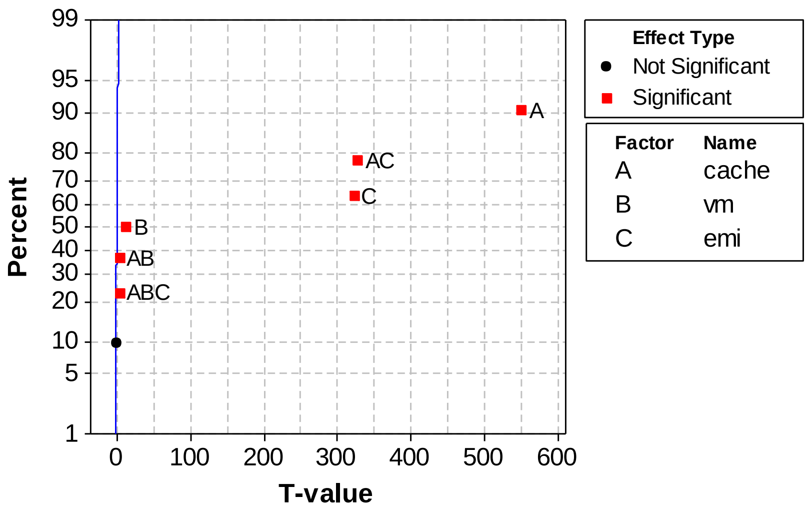
The effect and relevance of each factor and their interactions were computed based on results of total instantiation time shown in Table 5.8. Table 5.9, Figure 5.9, and Figure 5.10 show the results of these analyses just for total instantiation time. The evaluation for the times of isolated instantiation phases (i.e., instance reservation, EMI copy, and VM preparation) is not presented here for the sake of conciseness.

Table 5.9: Estimated effects and relevances for the total time of instantiation

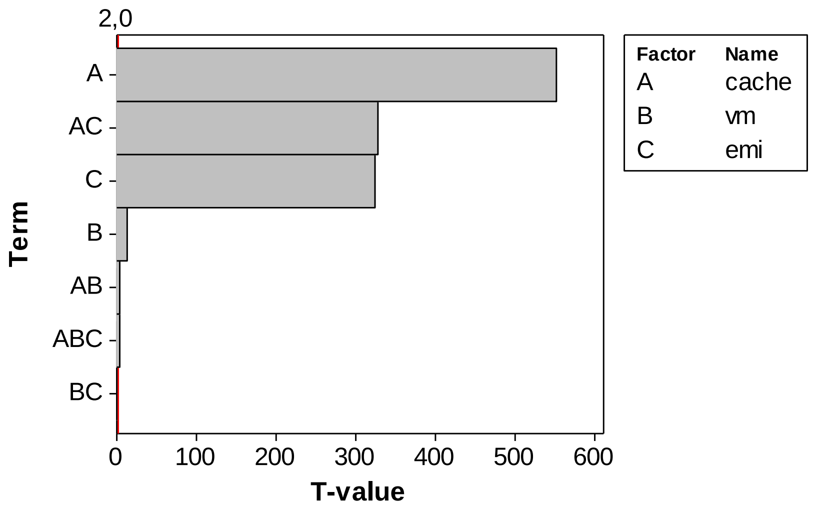
Factors	Effects	T	Relevances	P-value
Constant		603.09		0.000
A	472415	551.55	45.0720 %	0.000
B	11031	12.88	1.0525 %	0.000
C	277260	323.71	26.4532 %	0.000
AB	3136	3.66	0.2991 %	0.000
AC	281244	328.36	26.8332 %	0.000
BC	-88	-0.1	-0.0082 %	0.918
ABC	3129	3.65	0.2983 %	0.000

In Table 5.9, Figure 5.9, and Figure 5.10, each factor corresponds to a letter: Cache = **A**, VM type = **B**, and EMI size = **C**. The results showed that the factors that most impact on performance are **A** (cache), generating a performance effect with relevance of 45.072 %, followed by **C** (EMI size) with 26.453 %, and the interaction **AC** with around 26.833 %. The factor **B** (VM type) with 1.052 % and its interactions (**AB**, **BC**, and **ABC**) obtained no significant relevance, where **AB** obtained around 0.299 %, **BC** with -0.008 %, and **ABC** with 0.298 %. Note also that the *p-value* of interaction **BC** was 0.918, being above 0.05 (set as the threshold in this study), i.e., there is not enough evidence that this interaction has significant effect on performance improvement (JAIN, 2008).

Figure 5.9 depicts in two distinct views the statistical significance of main and interactions factors. The T-value or T statistic is computed from the coefficients of the regression model shown in Table 5.9, which in turn is used to compute the P-value, both serve to examine the effect or statistic relevance of factors (JAIN, 2008).



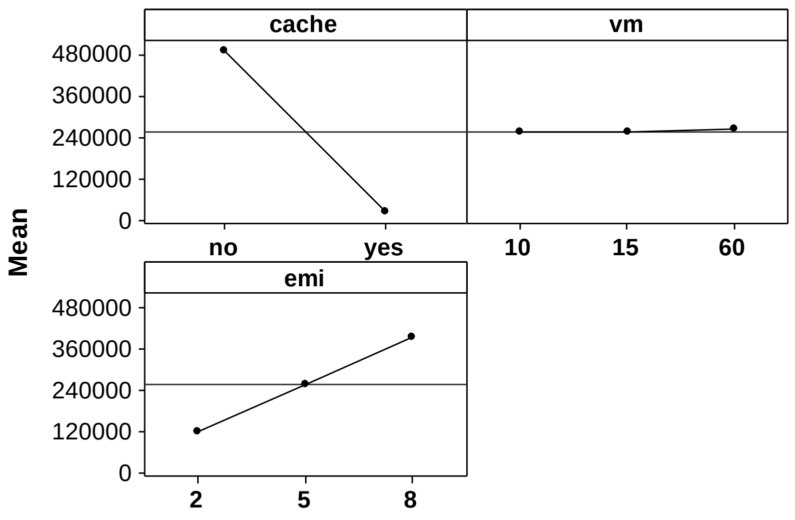
(a) Normal plot of the standardized effects



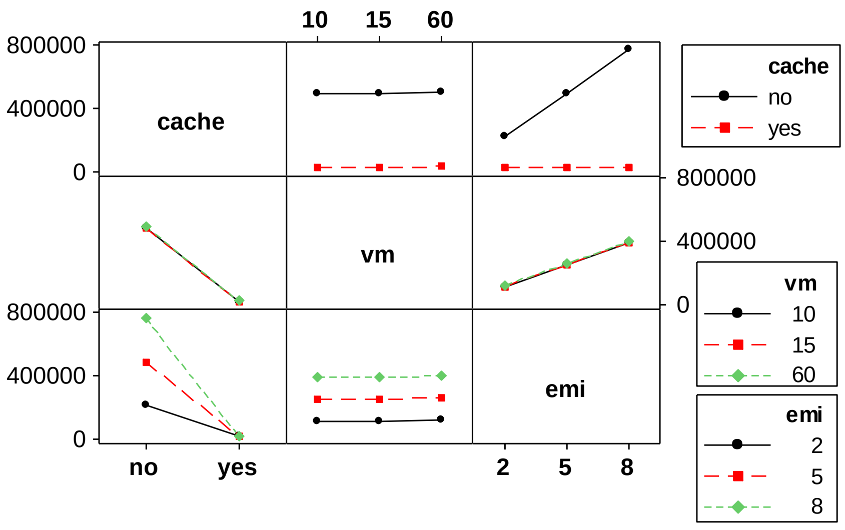
(b) Pareto chart of the standardized effects

Figure 5.9: Charts for statistical significance of effects for main factors and interactions

The Normal plot of the standardized effects (Figure 5.9 (a)), focuses on highlighting the percentage of statistical significance relative to T-value. The closer to the right the point is plotted, more significant is the corresponding factor. For instance, the factor **A** with T-value = 551.55 has 90 % of statistical significance, whereas the interaction **BC** presents a T-value = −0.1 with 10 % of significance. Factors with T-values located before the blue vertical line can be dropped from future analyzes because they do not have statistical significance, thus, these factors are not highlighted in the plot. The interaction **BC** is shown as a black dot, suggesting its lack of significance. In the Pareto chart of standardized effects (Figure 5.9 (b)), the analysis presents another aspect, all factors are shown, and statistical effect (T-value) of each one is depicted in a bar plot. Similar to Figure 5.9 (a), this plot has a red vertical line. The factors with T-value located before such a line has no significant statistical effects. In other words, it is assumed hypothetically that all the statistical results of these factors do not represent reality (JAIN, 2008).



(a) Main effects plot



(b) Interaction effects plot

Figure 5.10: Charts for main and interactions effects of factors

Figure 5.9 (b) shows the interaction **BC** with T-value less than 0 (before the line). We conclude from both views that the factors **A**, **C**, and **AC** have much more significant effects than the other factors.

The main effects plot, Figure 5.10 (a), shows the isolated factors and the effects of their distinct levels. Since the difference between levels is significant, this highlights the cache and EMI size as high impacting factors, as denoted by both slopes. On the other hand, the VM type isolated has a much smaller effect, denoted by the almost horizontal line. Figure 5.10 (b) shows a evaluation of the effects of factors interacting with each other. The interaction that has no parallel lines suggests a significant impact on the measure of interest, i.e., the total instantiation time. This is the case of the upper right and the lower left cells in the grid, which represent the interaction between cache and EMI size factors. Both plots show that when the cache is not used, the effect of varying EMI size is significant, but when the cache is working, the different levels of EMI cause negligible impact on the instantiation time. Therefore, the employment of network equipment and configuration to provide high bandwidth and throughput is especially valuable in no-cache scenarios. Moreover, efforts such as the customization of EMIs for using small disk space, or compressing the EMI for transmission through the network might be considered worthy in environments which do not use cache, but likely not when the EMI can already be in the cache of all nodes. However, a specific study is needed to evaluate the viability of EMI compression strategy, due to computational costs for compression and decompression.

It is also important to stress that all interactions with the factor VM type have parallel lines on Figure 5.10 (b), confirming the analysis shown on the other plots, i.e., are interactions without major influence on results.

5.2 Case Study Two

In this section, we used the SPN main model depicted in Figure 4.2 to evaluate the performance of a composite web service. We verified the effects of various adjustments in the main transitions' delays, obtaining measures such as mean queue size, average number of busy VMs, average utilization of VMs, and mean response time to the user. One specific study varied the rates of the mashup CTMC submodel, and verified the impact on a given metric of the SPN main model. A similar study was also carried out with the VM instantiation CTMC submodel.

The delays assigned to all timed transitions is presented in Table 5.10. The values for delays of **TReq**, **TLB**, **TSend**, and **TRep** were obtained in a Eucalyptus private cloud testbed, using default configuration parameters for the Eucalyptus CloudWatch (EUCALYPTUS, 2014d). The Apache JMeter benchmark (JMETER, 2015) was used in these tests. The Eucalyptus CloudWatch has a feature that provides the times of the ELB service (**TLB**) (EUCALYPTUS, 2014d). For **TReq**, the rate between requests arrivals was chosen according to the type of system evaluated, in this case event recommendation mashup web service. Therefore, the value of **TReq** was configured in JMeter benchmark. The network delay for sending the request and receiving the response was monitored to assign to the transitions **TSend**, and **TRep**, respectively.

The value for **TService** delay comes from CTMC submodel, which represents the event recommendation mashup. Table 5.11 presents the values assigned to parameters of the mashup application submodel. The mean response time of each specific web service provider was obtained through measurements on a real mashup application, calling specific web services provided by Foursquare (FOURSQUARE, 2015), Google Maps (GOOGLE, 2015), Last.fm (LAST.FM, 2015), and Eventful (EVENTFUL, 2015). Matos *et al.* (MATOS; MACIEL; SILVA, 2013) validated this submodel and provided some results of each specific web service provider this experimental study.

The values for **TCollectMetrics** and **TWindow** depend on the configuration performed by the system administrator (EUCALYPTUS, 2014d,f), in this study, we adopted 60 seconds for

Table 5.10: Timed transitions of the SPN model for scalable web service on private cloud

Transition	Description	Value (s)
TColletMetrics	Time period for metrics collection on the nodes	60.0
TWindow	Time window for CloudWatch	60.0
TCheckThreshold	Time for summarizing, computing and compare metrics	1.0
TInst	Time for instantiation of a new VM	37.2
TReq	Time between user requests	4.0
TSend	Network latency to send request	0.2
TLB	Time for Load Balancer forward request	1.0
TService	Response time of mashup	6.9
TRep	Network latency to send response	0.2

Table 5.11: Parameter values for the mashup CTMC model

Parameter	Description	Value (s)
mrt_{ES}	Mean resp. time of Event Search	2.333
mrt_{VS}	Mean resp. time of Venue Search	0.324
mrt_{SA}	Mean resp. time of Similar Artists	2.286
mrt_{TS}	Mean resp. time of Top Event Selection	0.226
mrt_{MS}	Mean resp. time of Map Search	0.452
mrt_{SS}	Mean resp. time of Song Search	1.909

these two deterministic transitions. Such a configuration yielded the best results in an experimental evaluation that is presented in case study one (Section 5.1). The value of **TCheckThreshold** is 1 second, because when the time window closes, Eucalyptus CloudWatch takes around 1 second to summarize, computing and compare the metrics with the predefined thresholds, to then trigger an action (add or remove an instance) (EUCALYPTUS, 2014f).

TInst delay comes from CTMC submodel which represents the VM instantiation, analyzed further. The values of this submodel were validated using results from the Eucalyptus private cloud testbed also presented in case study one.

The SPN main model was solved through stationary simulation, obtaining measures such as mean queue size, average number of busy VMs, average utilization of VMs, and mean response time to the user. Table 5.12 presents these results. The simulation was executed for a confidence level of 95%, maximum relative error of 5%, warm-up period of 50 runs, run size (i.e.: number of times each transition fires) of 1000, and maximum simulation time of 120 seconds. The CTMC submodels (VM instantiation and mashup service) were solved through stationary analysis, providing the values to be assigned as delays of **TInst** and **TService** transitions respectively. The Mercury tool was used for these analyses (SILVA et al., 2013; CALLOU et al., 2013).

Table 5.12 presents the performance measures computed considering the baseline configuration of parameters values shown in Table 5.10. The average utilization of VMs is around 38.3%, what shows that the system has enough capacity to serve user requests with the allocated

Table 5.12: Performance measures

Measure	Expression	Value
Utilization of VMs (%)	E{#BusyVMs})/(E{#IdleVMs}+E{#BusyVMs})	38.3 %
Average number of busy VMs	E{#BusyVMs}	1.716
Average number of idle VMs	E{#IdleVMs}	2.773
LB queue size (#of requests)	E{#Queue}	0.432
Mean response time - Rsp - (s)	NRequests/(P{#PReply>0}×(1/TReply))	9.029 s

resources. Such a capacity is partially provided by means of additional VMs created by the auto scaling mechanism. This is confirmed by the sum of average number of busy VMs and average number of idle VMs, that is equal to approximately 4.48, whereas the system starts with only two VMs. If the auto scaling mechanism were not working –and we had only two VMs– the average utilization could reach about 85%, incurring in risks of bad performance for the users, mainly during high workload bursts. The average load balancer queue size is another measure that shows the system is not overloaded, since the requests do not wait in the queue for being distributed to the VMs.

The mean response time of the system (*Rsp*) is 9.029 seconds. This is the round-trip time interval elapsed from the dispatch of user request to response arrival. The measure expression on the SPN is $Rsp = NRequests/(P\{\#PReply > 0\} \times (1/TReply))$, where *NRequests* is the average number of requests in the system. *NRequests* is computed through the expression ($E\{\#PSend\} + E\{\#JobAdmission\} + E\{\#Queue\} + E\{\#BusyVMs\} + E\{\#PReply\}$), which sums up the number of tokens in the corresponding places. By summing up the response time of the mashup application, the request and reply network delays, and the time for load balancer distribution, the result is close to the system response time (9.029 s), indicating that requests spend little time in queue. Even then, the mean response time of this system might be shortened by tuning some of its many parameters and components. In order to identify the most effective points for a response time improvement, it is important to assess the measure sensitivity to models' parameters. Therefore, we varied the total time of the submodels (varying the rates of one parameter at a time), and we checked the level of impact on the response time of the SPN main model.

We present a comparison of impact among some parameters through scatter plots. Figure 5.11 depicts the impact of parameters mrt_{ES}, mrt_{SA}, and mrt_{SS} (from the CTMC mashup submodel) on system response time, computed from the SPN main model. We grouped these parameters on Figure 5.11 because they belong to the same submodel. The plot is generated by fixing all parameters at their baseline values (see Tables 2.4, 5.14, and 5.11), except by one parameter that is varied through a specific range in steps of about 10%, enabling the comparison of impact on the system response time. Notice that the slopes of mrt_{ES}, mrt_{SA}, and mrt_{SS} are similar.

Figure 5.12 presents the impact of parameters **TLB**, **TSend**, and **TRep**, which belong to the SPN main model. **TLB** shows a slightly higher impact on system response time than **TRep** and **TSend** do. Note that this analysis is related to the slope of the line (average variance), and not the absolute values of the y axis.

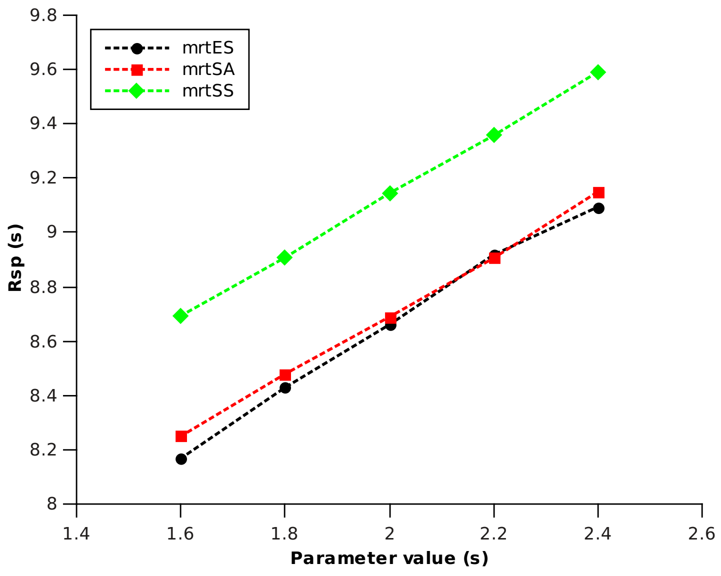


Figure 5.11: Impact of mrt_{ES}, mrt_{SA}, and mrt_{SS} on system response time

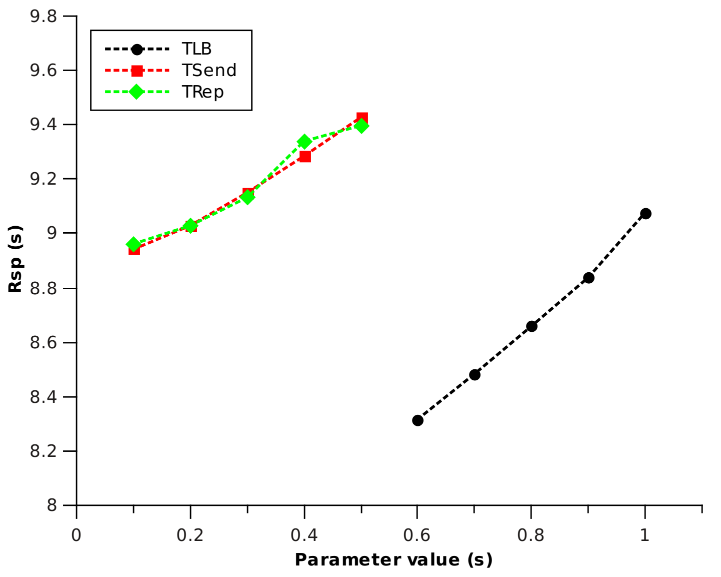


Figure 5.12: Impact of **TLB**, **TSend**, and **TRep** on system response time

Figure 5.13 depicts the impact of parameter **pCache** on system response time, computed from the SPN main model. This is one parameter from the VM instantiation submodel. This relatively lower impact of **pCache** is noteworthy by comparing Figure 5.11 and Figure 5.12 to Figure 5.13. It is worth highlighting that even varying **pCache** with steps larger than 10%, its effect on system response time is limited to about 0.1 second throughout the plot, whereas in Figure 5.11 the impact reaches around 0.8 seconds. On the other hand, **pCache** may be one of the parameters most easily tunable in the system, if compared to specific services response times

(mrt_{ES}, mrt_{SA}, and mrt_{SS}) or parameters related to network latency (**TSend**, **TRep**). Even so, in specific studies on the VM instantiation submodel (Section 5.3) have shown that **pCache** has a higher sensitivity index than many other parameters, and therefore deserves attention from system administrators. This is one reason for the heterogeneous hierarchical modeling, being able to analyze the system-wide behavior without losing details of specific subsystems processes.

Notice that we kept the range of Y-axis (system response time) the same for all plots (Figure 5.11, Figure 5.12, and Figure 5.13), so we can compare the slopes of lines from distinct graphs. Systems administrators can benefit of this approach, that may guide investments and help decision making on which are the high priority components during tune-up and upgrade efforts.

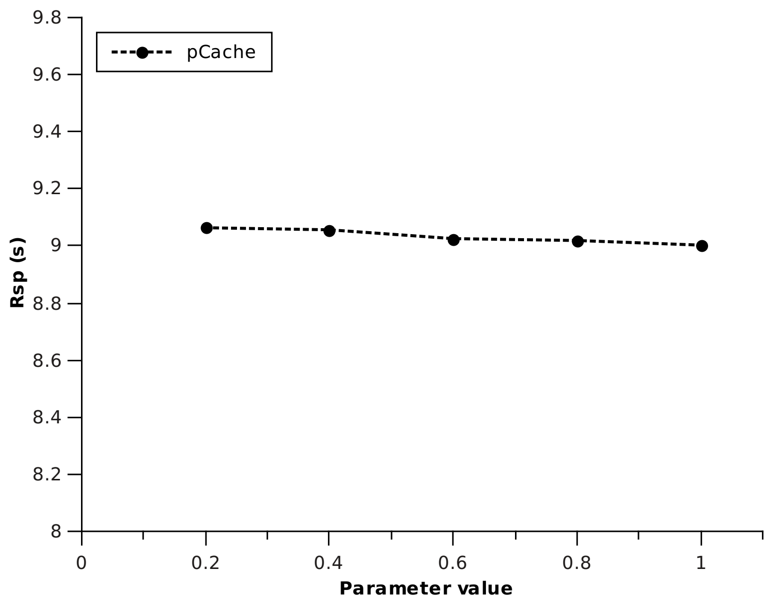


Figure 5.13: Impact of **pCache** on system response time

5.3 Case Study Three

This section presents the results obtained from refined VM instantiation CTMC submodel shown in Figure 4.5. The main objective is verifying the behavior of the VM instantiation time for distinct levels of the parameter *pCache*. Therefore, we carried out a performance evaluation and sensitivity analysis in *pCache* parameter. Sensitivity analysis from design of experiment (presented in Section 5.1) shows that cache is the most impacting factor for the total instantiation time. For this reason, we decided to analyze specifically the *pCache* parameter.

Table 5.13 presents description of parameters and their values, which were obtained from testbed experiments and used to verify the results provided by the CTMC submodel. The times obtained from the measurement of experimental activity is presented in the case study one. In (CAMPOS ELIOMAR; MATOS; SILVA, 2015a) this study is also presented.

Table 5.13: Parameter values for the CTMC submodel of VM instantiation

Parameter	Description	Value
pCache	Probability that EMI is already in cache	1 (100%)
tRI	Mean instance reservation time	0.28 s
tCI	Mean EMI local copy time	7.624 s
tDI	Mean EMI download time	194.79 s
tPV	Mean VM preparation time	10.603 s

The parameter values *tRI*, *tCI*, and *tPV* are based on the time shown in Table 5.8 (case study four) for the first scenario (**Y 10 2**): using node's cache, VM with 10 GB disk, 2 GB sized EMI. The value for *tDI* was obtained in a scenario **N 10 2** similar to the previous one, but with disabled cache. The *pCache* was initially set to 1 to allow comparison of results provided by the submodel with those from experiments with cache enabled. This is an aspect which might deserve characterization in an infrastructure with multiple nodes, and multiple EMIs to instantiate. Even when the cache is enabled, some nodes may not always have the required EMI available in its cache, due to disk space restrictions or simply lack of a previous instantiation.

The Mean Time to Absorption (MTTA) (KOHLAS, 1986), i.e., the mean instantiation time computed from the CTMC with those parameters, is 18.507 s. The time obtained in the experiments is 18.506 s with confidence interval of (15.663 s; 21.349 s) for a confidence level of 95 %, as seen in Table 5.8. When we change the parameters in the CTMC to match the configuration of scenario **N 10 2** (absence of cache), the computed MTTA is 209.670 s, the same value measured for the total instantiation time in the corresponding experiment scenario, whose confidence interval is (209.636 s; 209.703 s) for a confidence level of 95 %. Therefore, our submodel provides consistent results to those observed in the experimental testbed, i.e., we verify the equivalence between the submodels and the experiment.

After having the submodel validated, we go further by using the instantiation CTMC submodel to apply statistical analyses. We have performed parametric sensitivity analyses, in order to predict the impact on time of the instantiation process while the rate of a specific factor varies throughout a given range. For instance, we have carried out a specific research study in which we have checked the performance of the instantiation time to various *pCache* values.

Figure 5.14 depicts the sensitivity analysis (MATOS et al., 2012) of instantiation time with respect to *pCache*, for three levels of EMI size (2 GB, 5 GB, and 8 GB). The plot shows a linear relationship between *pCache* and the instantiation time, which can be explored for example by system administrators searching for a compromise between system performance and efforts to pre-load EMIs in the cache of cloud nodes. For example, assuming that a given application requires an average instantiation time smaller than 300 seconds, from Figure 5.14 we find out that such a requirement is achievable for a 2 GB EMI even without enabling cache, but for 5 GB EMI and 8 GB EMI, the probability of using cache shall be higher than 40 %, and 60 %, respectively.

We mentioned earlier in case study one, that the Figure 5.13 depicts the impact of parameter *pCache* on system response time of the SPN main model. Where *pCache* have relatively lower impact comparing with Figure 5.11 and Figure 5.12. However, now we see that when analyzing the parameter *pCache* comparing or interacting to specific parameters of the instantiation process, such as **EMI**, *pCache* has a higher sensitivity index than many other parameters, and therefore deserves attention from system administrators.

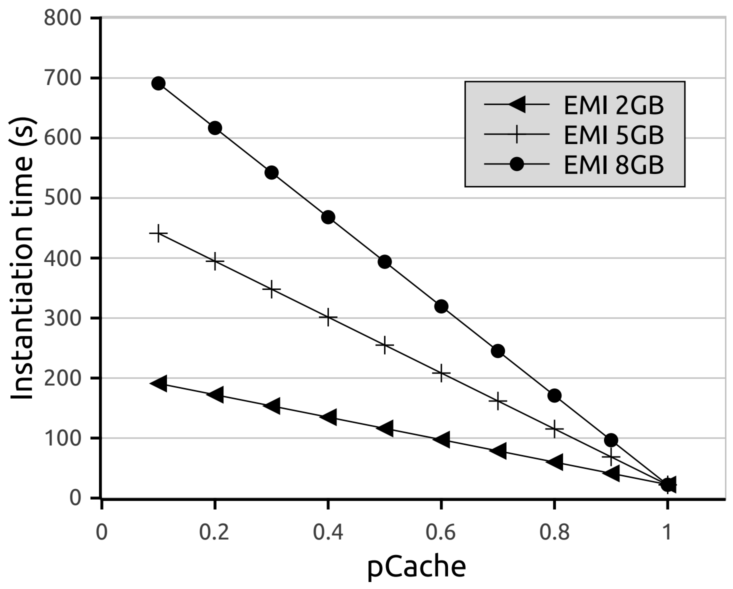


Figure 5.14: Sensitivity analysis of instantiation time with respect to *pCache*

It is worth stressing that the VM instantiation CTMC submodel presented in this work can be extended and used for other analyses not shown here. This CTMC also allows composition with other models to evaluate higher level applications which make intensive use of VM instantiations, what is demonstrated by the hierarchical composition with the SPN model described in Chapter 4.

5.4 Case Study Four

The next step in our performance evaluation study is using the refined auto scaling CTMC model shown in Figure 4.7 (c). We carried out a performance evaluation and sensitivity analysis in all parameters of the auto scaling process. As explained in Chapter 4, this CTMC model represents the whole auto scaling process, i.e., starts by monitoring the workload arrival, and finishes with VM instantiation (when the incoming workload violates a predefined threshold). Therefore, VM instantiation is the final step of the auto scaling process.

Table 5.14 presents description of parameters and their values, which were obtained from testbed experiments and used to verify the results provided by the CTMC model. The detailed times obtained from the measurement of experimental activity are presented in Section 5.1 in (CAMPOS ELIOMAR; MATOS; SILVA, 2015a,b).

Table 5.14: Parameter values for the CTMC model of auto scaling process

Parameter	Description	Value
tAS	Mean auto scaling monitoring time	188.6 s
pCache	Probability that EMI is already in cache	1 (100%)
tRI	Mean instance reservation time	0.28 s
tCI	Mean EMI local copy time	7.624 s
tDI	Mean EMI download time	194.79 s
tPV	Mean VM preparation time	10.603 s

The value of parameter *tAS* is based on the time shown in Table 5.3 (Section 5.1) for the first scenario (**1 1 1**): with *collection period* of 1 minute, *dimension* of 1 VM, and 1 minute of *window time*. The *dimension* corresponds to the amount of VMs monitored. The factor *window time* is related to the period taken to observe a metric. The factor *collection period* is associated with the time interval for CloudWatch collecting information from all NCs repositories.

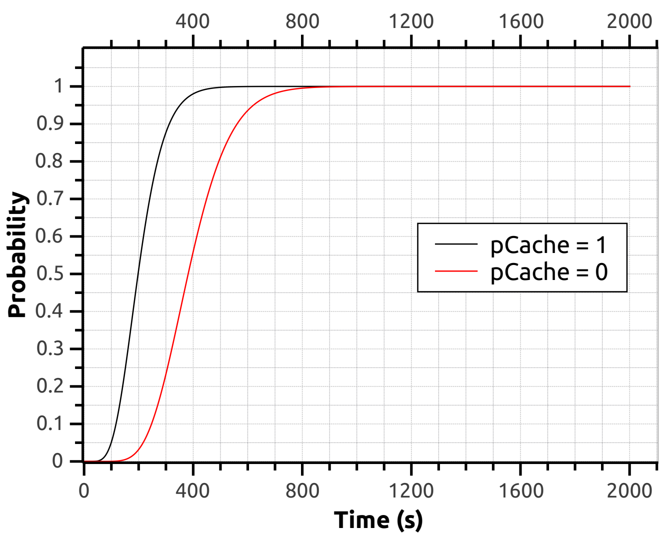
The parameter values *tRI*, *tCI*, and *tPV* are based on the time shown in Table 5.8 (Section 5.1) (CAMPOS ELIOMAR; MATOS; SILVA, 2015a) for the first scenario (**Y 10 2**): using node's cache, VM with 10 GB disk, 2 GB sized EMI. The value for *tDI* was obtained in a scenario **N 10 2** that is similar to the previous one, but with disabled cache. The *pCache* was initially set to 1 to allow comparison of results provided by the submodel with those values experiments with cache enabled. Note that the values of these parameters are the same used in the previously case study.

The MTTA (KOHLAS, 1986) of the CTMC is the mean time of auto scaling monitoring followed by VM instantiation process. The value of MTTA with those input parameters is 207.107 s. It is exactly the same result from the experiments, if we sum up the values from Table 5.3 without the parameter *tDI*. Changing parameters in CTMC to match the scenario without cache, the MTTA is 394.273 s, exactly the same result if summing up the values in Table 5.3 without the parameter *tCI*. Therefore, our submodel provides consistent results with the experimental data. The Mercury tool was used for the creation and analysis of all models of this work (SILVA et al., 2013; CALLOU et al., 2013).

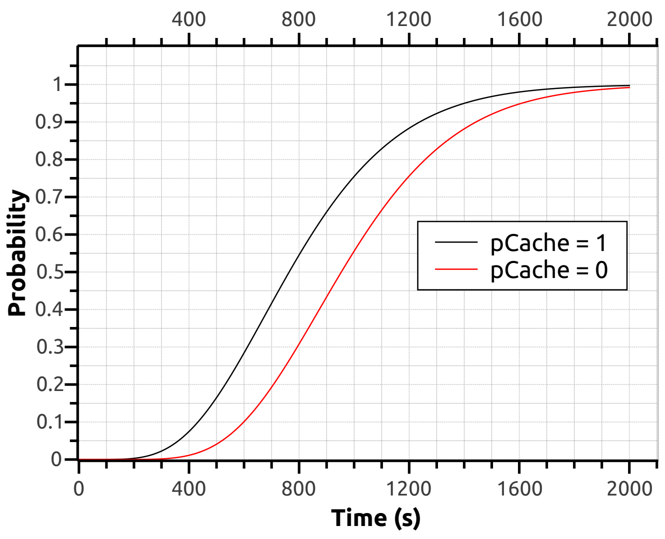
The next step in our performance evaluation study is to use the CTMC submodel for further analysis. We computed the probabilities of absorption. Such value means the probability of finalizing the auto scaling monitoring and instantiation process by a given time t. These probabilities were calculated from $t = 0$ s until $t = 2000$ s, and plotted as cumulative distribution function as shown in Figure 5.15.

In Figure 5.15 (a), we consider the scenario (**1 1 1**) presented in Table 5.3, where *tAS* = 186.6 s. When cache is enabled on nodes, with $pCache = 1$ for example, the 100 % probability completing the auto scaling monitoring and VM instantiation process is achieved around 500 s. But when cache is not enabled on nodes, with $pCache = 0$ for example, the 100 % probability is achieved around 800 s.

In Figure 5.15 (b), we consider the scenario (**5 1 1**) (i.e., now with *collection period* of 5 minutes) presented in Table 5.3, where *tAS* = 788.6 s. When cache is enabled, the 100 % absorption probability is achieved around 1900 s. But when cache is not enabled, the 100 % probability is achieved around 2000 s.



(a) Probabilities for the scenario (**1 1 1**): *collection period* = 1, *dimension* = 1, and *window time* = 1



(b) Probabilities for the scenario (**5 1 1**): *collection period* = 5, *dimension* = 1, and *window time* = 1

Figure 5.15: Graphs of absorption probabilities of the auto scaling process.

Therefore there is a significant impact on system performance when *collection period* factor is modified from 1 minute to 5 minutes (in this case, considering the *dimension* of 1 VM and the *window time* of 1 minute as fixed levels). This is demonstrated by the difference between two scenarios was at least 1400 s when $pCache = 1$, and at least 1200 s when $pCache = 0$. We can also notice that parameter *pCache* modified considerably the time to complete the process, especially for scenario (**1 1 1**), increasing 300 s when cache was not enabled.

Next, we examined the behavior of the MTTA (KOHLAS, 1986), i.e., the mean time to complete the auto scaling monitoring followed by VM instantiation, for different values of *tAS*, *tRI*, *tCI*, *tDI*, and *tPV*. This sensitivity analysis can allow system administrators to better prioritize efforts to reduce time of each phase. Figure 5.16 shows the variation of MTTA with respect to *tAS*, *tRI*, *tCI*, *tDI*, and *tPV*, considering three levels of *pCache* (0.25, 0.5, and 0.75). The graphs show a linear relationship between each of those factors and MTTA, which is expected because they are stages of a sequential process. For such an analysis, values found in Table 5.14 were used as the baseline configuration. We varied one parameter at a time, keeping values of other parameters unchanged.

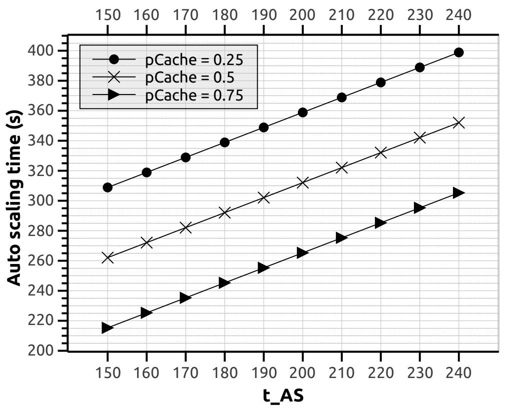
In Figure 5.16 (a), the parameter *tAS* varied around its average in a range from 150 s to 240 s with steps of 10 s. In this graph we confirm that variations on parameter *tAS* have a large effect on MTTA, for all levels of *pCache*.

In Figure 5.16 (b), we varied *tRI* around its average in a range from 0.1 s to 1 s with intervals of 0.1 s. Since the instance reservation time was quite small, we noticed that decreasing it would have little effect on MTTA, for all *pCache* levels.

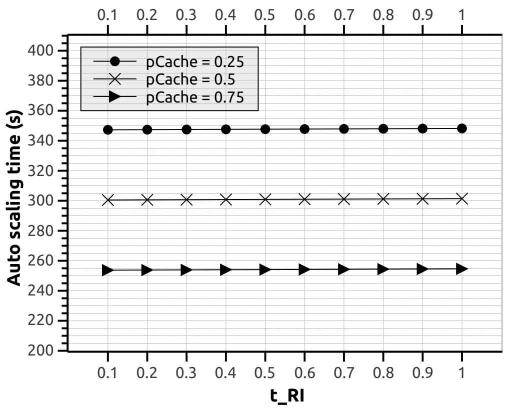
In Figure 5.16 (c), we varied the parameter *tCI* around its average in a range from 5 s to 14 s with intervals of 1 s. Please notice that MTTA was slightly affected by the time of copying the EMI locally on node. We can particularly emphasize that changes in *tCI* had an almost negligible impact on MTTA for $pCache = 0.25$ and $pCache = 0.5$, while this impact was larger for $pCache = 0.75$.

In Figure 5.16 (d), we varied parameter *tDI* around its average in a range from 150 s to 240 s with steps of 10 s. Importantly, the range here was similar to the range adopted for *tAS*, and larger than for other parameters because we adopted steps based on percentage values around the average of each parameter. Figure 5.16 (d) shows that by varying *tDI*, the instantiation time changed very significantly for all *pCache* levels. Moreover, the smaller the value of *pCache* was, the greater was the impact of *tDI* in MTTA. Therefore, efforts to decrease *tDI* produced more benefits when there was a low probability of finding the VM image in node's cache.

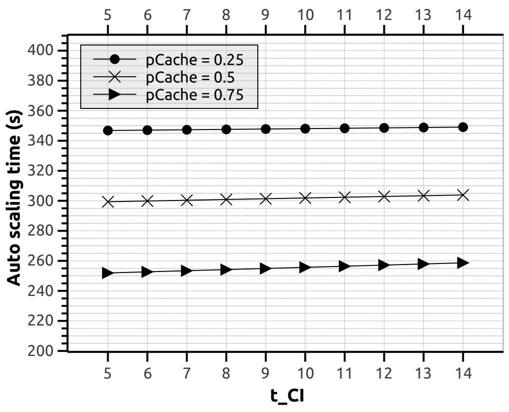
Finally, in Figure 5.16 (e), we varied parameter *tPV* around its average in a range from 5 s to 14 s with intervals of 1 s. The graph shows that the variation of *tPV* lightly impacted on MTTA, and the level of this impact was almost the same for all *pCache* levels, something which was expected, since the VM configuration occurs only after copying or downloading the EMI.



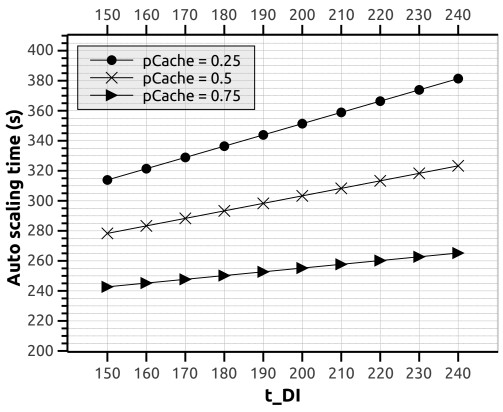
(a) Variation of *tAS* (auto scaling monitoring time)



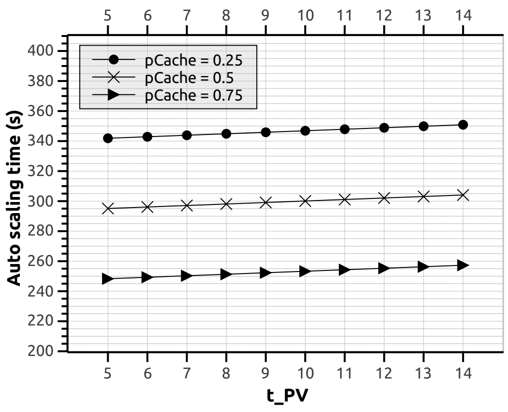
(b) Variation of *tRI* (instance reservation time)



(c) Variation of *tCI* (EMI local copy time)



(d) Variation of *tDI* (EMI download time)



(e) Variation of *tPV* (VM preparation time)

Figure 5.16: Sensitivity analysis of auto scaling process with respect to parameters *tAS*, *tRI*, *tCI*, *tDI*, and *tPV*

6 Conclusion

This study investigated the performance of a composite web service (event recommendation mashup), running on a private cloud infrastructure with auto scaling mechanisms.

A *General Full Factorial Design of Experiments* was performed for the auto scaling monitoring. The effect and relevance in three factors were analyzed – *collection period*, *dimension*, and *window time* – considering as target metric the total time of auto scaling monitoring process. The experimental results showed that the most influential factors were: *collection period*, the *window time*, and the interaction between these two factors. The *collection period* of 1 minute caused a significant decrease in mean time for auto scaling monitoring. The size of *window time* also had a great effect, especially when the *collection period* was 1 minute.

A *Full Factorial Design of Experiments* was performed for the VM instantiation process. We analyzed the effects and relevance of three factors – cache, VM type, and EMI size – considering the total time of instantiation. The times for completion of intermediate phases of the instantiation were also measured and analyzed. The experimental results pointed out that the most influencing factors are the cache and EMI size, including the interaction between these factors. The use of cache causes a significant decrease in the instantiation time. The size of EMI also has a large effect, mainly when the cache is not used. The presented analysis lead to the suggestion of some good practices for cloud infrastructures administrators. Due to the high impact of EMI size, the customization of virtual machine images to produce small EMIs is recommended, as well as the employment of network equipment and configuration to provide high bandwidth and throughput. Preloading EMIs on a fraction of the available nodes might also improve significantly the instantiation time, since it may increase the probability of using the cache.

The hierarchical heterogeneous modeling approach enabled an analysis that included the main high-level components as well as details of the auto scaling monitoring, VM instantiation process and each specific web service call within the mashup application.

Through the SPN main model, metrics such as average VMs utilization, number of idle VMs, and length of load balancer queue demonstrated that the system is balanced, but partially due to the usage of VMs created by the auto scaling mechanism. The system response time, from end user perspective, confirms that the system is able to process requests without adding excessive delay due to queuing mechanisms. Moreover, a sensitivity analysis technique tailored for hierarchical models indicated effective points to be tuned in this system in order to achieve even better performance. The most influential factors for the system are the response time of *Event Search* and *Similar Artists* services (parameters of the mashup CTMC submodel), as well as the *Load Balancer* of the private cloud. Considering the system setup (e.g.: workload and VMs capacity) evaluated here, the time to instantiate VMs –and its inner variables – is not among the parameters that have a big impact on overall performance.

This study investigated the auto scaling process, an important activity in cloud computing systems, and intensively used by web services that deal with unexpected request peaks. For this reason, was develop a CTMC model that represents the auto scaling process from threshold violation until to complete instantiation of a new VM. The analysis presented are helpful for cloud infrastructure administrators, that can properly tune parameters such as the *collection period*, as well as give especial attention to the benefits of populating the cache of VM images on nodes, and keeping VM images as small as possible to avoid long EMI download times. The approach for sensitivity analysis presented here contributes with guides to prioritize efforts and might be applied in similar models of cloud computing infrastructures and their applications.

6.1 Statement of the Contributions

As a result of the work presented in this dissertation, the following contributions can be highlighted:

- Approach of a performance evaluation methodology of a main system and its subsystems. This methodology makes use of hierarchical modeling and experimental design. This approach enables to represent details of specific processes using parametric sensitivity analysis;
- Proposal of a SPN main model to represent the functioning of a composite web service with elasticity mechanisms in a private cloud platform, in order to apply parametric sensitivity analysis;
- Proposal of a CTMC submodel to represent the functioning of event recommendation mashup, in order to apply parametric sensitivity analysis;
- Proposal of a CTMC submodel to represent the functioning of the VM instantiation in a private cloud platform, in order to apply parametric sensitivity analysis.
- Proposal of a CTMC model to represent the functioning of the all auto scaling process in a private cloud platform, in order to apply parametric sensitivity analysis. The auto scaling CTMC model presented in this work can be extended and combined with

other models, in order to assess the higher-level applications that might benefit from intensive usage of auto scaling mechanism and VM instantiation;

- Performance evaluation of a General Full Factorial Design Experiment of the auto scaling monitoring, which aims to analyze the impact of different factors on subsystem performance;
- Performance evaluation of a Full Factorial Design Experiment of the VM instantiation process;
- Assist system administrators to properly configure the auto scaling mechanism in private clouds frameworks. Such results can also aid in the development of techniques or algorithms to improve performance of auto scaling functions in private clouds frameworks from the Scripts used on testbeds this work.

In addition to the contribution mentioned, some papers presenting the findings of this dissertation were produced:

- Eliomar Campos, Rubens Matos, Paulo Maciel, Igor Costa, Francisco Souza, and Francisco Airton Silva. Performance Evaluation of Virtual Machines Instantiation in a Private Cloud. In: Proceedings of the IEEE 11th World Congress on Services (IEEE SERVICES 2015). June 27 - July 2, 2015 - New York - USA (CAMPOS ELIOMAR; MATOS; SILVA, 2015a).
- Eliomar Campos, Rubens Matos, Paulo Maciel, Francisco Souza, and Francisco Airton Silva. Stochastic Modeling of Auto Scaling Mechanism in Private Clouds for Supporting Performance Tunning. In: Proceedings of the 2015 IEEE International Conference on Systems, Man, and Cybernetics (SMC2015). October 09-12, 2015 - Hong Kong - China (CAMPOS ELIOMAR; MATOS; SILVA, 2015b).

Other papers to be published:

- Eliomar Campos, Rubens Matos, Paulo Maciel, Francisco Souza, and Francisco Airton Silva. Performance Evaluation of Auto Scaling Mechanism on Private Cloud. In Performance Evaluation Review. (to submit)
- Rubens Matos, Eliomar Campos, Paulo Maciel, and Artur Henriques. Performance Modeling and Sensitivity Analysis of Scalable Web Service on Private Cloud. In IEEE Transactions on Services. (submitted)

6.2 Future Works

Below, we list the extensions of the current work to be carried out in future work:

- Performance evaluation considering other factors related to the elasticity mechanisms such as scaling policy, scaling size, workload, threshold (e.g.: CPU utilization, memory utilization, requests);
- Performance evaluation considering other important factors related to the VM instantiation such as network bandwidth, disk throughput, type of discs, file systems, and CPU load/speed;
- Future works may verify the feasibility of EMI compression before transmission to the node and the subsequent decompression, or considered keeping the EMIs in flash storage;
- Other studies might evaluate other private cloud environments, such as OpenStack and OpenNebula;

References

ABDALLAH, H.; HAMZA, M. On the sensitivity analysis of the expected accumulated reward. **Perf. Eval.**, Amsterdam, The Netherlands, The Netherlands, v.47, n.2, p.163–179, 2002.

AL-HAIDARI, F.; SQALLI, M.; SALAH, K. Impact of CPU Utilization Thresholds and Scaling Size on Autoscaling Cloud Resources. In: CLOUD COMPUTING TECHNOLOGY AND SCIENCE (CLOUDCOM), 2013 IEEE 5TH INTERNATIONAL CONFERENCE ON. **Anais. . .** [S.l.: s.n.], 2013. v.2, p.256–261.

AMAZON. **Amazon Simple Storage Service (S3)**. Available on `http://aws.amazon.com/s3/`.

AMAZON. **Amazon Elastic Compute Cloud**: user guide. [S.l.: s.n.], 2014.

AMAZON. **Auto Scaling**. Available on `http://aws.amazon.com/autoscaling/`.

AMAZON. **What is Cloud Computing?** Available on `http://aws.amazon.com/what-is-cloud-computing/`.

BAUER, E.; ADAMS, R. **Reliability and Availability of Cloud Computing**. [S.l.]: Wiley-IEEE Press, 2012.

BLAKE, J. T.; REIBMAN, A. L.; TRIVEDI, K. S. Sensitivity analysis of reliability and performability measures for multiprocessor systems. In: ACM SIGMETRICS CONFERENCE ON MEASUREMENT AND MODELING OF COMPUTER SYSTEMS, 1988., New York, NY, USA. **Proceedings. . .** ACM, 1988. p.177–186.

BOLCH, G. et al. **Queuing Networks and Markov Chains**: modeling and performance evaluation with computer science applications. 2.ed. [S.l.]: John Wiley and Sons, 2001.

BOTRAN, L. et al. A Review of Auto-scaling Techniques for Elastic Applications in Cloud Environments. **Journal of Grid Computing**, [S.l.], v.12, n.4, p.559–592, 2014.

CALLOU, G. et al. Estimating sustainability impact of high dependable data centers: a comparative study between brazilian and us energy mixes. **Computing**, [S.l.], p.1–34, 2013.

CAMPOS ELIOMAR; MATOS, R. M. P. C. I. S. F.; SILVA, F. A. Performance Evaluation of Virtual Machines Instantiation in a Private Cloud. In: SERVICES (SERVICES), 2015 IEEE WORLD CONGRESS ON. **Anais. . .** [S.l.: s.n.], 2015. p.319–326.

CAMPOS ELIOMAR; MATOS, R. M. P. S. F.; SILVA, F. A. Stochastic Modeling of Auto Scaling Mechanism in Private Clouds for Supporting Performance Tunning. **Systems, Man, and Cybernetics (SMC), 2015 IEEE International Conference on**, [S.l.], 2015.

CARON, E. et al. **Auto-scaling, load balancing and monitoring in commercial and open-source clouds**. INRIA. Research report N. 7857. Available on `http://hal.inria.fr/docs/00/66/87/13/PDF/RR-7857.pdf`.

DANTAS, J. et al. Models for Dependability Analysis of Cloud Computing Architectures for Eucalyptus Platform. **International Transactions on Systems Science and Applications**, [S.l.], v.8, p.13–25, Dec 2012.

EUCALYPTUS. **AWS and Eucalyptus Compatibility**. [S.l.: s.n.], 2013. Available on `http://www.eucalyptus.com/aws-compatibility`.

EUCALYPTUS. **Eucalyptus 3.4.0 User Guide**. [S.l.: s.n.], 2013.

EUCALYPTUS. **What are private clouds?** Available on `https://www.eucalyptus.com/cloud-topics/what-are-private-clouds`.

EUCALYPTUS. **What is cloud platforms?** Available on `https://www.eucalyptus.com/cloud-topics/what-is-cloud-platforms/`.

EUCALYPTUS. **What is cloud computing?** Available on `https://www.eucalyptus.com/blog/2014/04/21/cloud-101-what-cloud-computing`.

EUCALYPTUS. **Official Documentation for Eucalyptus Cloud**. Available on `https://www.eucalyptus.com/docs/eucalyptus/4.0/`.

EUCALYPTUS. **Eucalyptus 3**: design, build and manage. [S.l.: s.n.], 2014.

EUCALYPTUS. **CloudWatch Troubleshooting**. Available on `https://github.com/eucalyptus/eucalyptus/wiki/CloudWatch-Troubleshooting/`.

EVENTFUL. **Overview**. Available on `http://about.eventful.com/`.

FERRARIS, F. et al. Evaluating the Auto Scaling Performance of Flexiscale and Amazon EC2 Clouds. In: SYMBOLIC AND NUMERIC ALGORITHMS FOR SCIENTIFIC COMPUTING (SYNASC), 2012 14TH INTERNATIONAL SYMPOSIUM ON. **Anais...** [S.l.: s.n.], 2012. p.423–429.

FOURSQUARE. **Foursquare**. Available on `https://foursquare.com/`.

FRANK, P. M. **Introduction to System Sensitivity Theory**. [S.l.]: Academic Press Inc, 1978.

FURHT, B. Cloud Computing Fundamentals. In: FURHT, B.; ESCALANTE, A. (Ed.). **Handbook of Cloud Computing**. [S.l.]: Springer US, 2010. p.3–19.

GERMAN, R. **Performance Analysis of Communication Systems with Non-Markovian Stochastic Petri Nets**. New York, NY, USA: John Wiley & Sons, Inc., 2000.

GOOGLE. **Google Maps**. Available on `http://www.google.com/maps/about/`.

GUIMARÃES, A. P.; MACIEL, P. R.; MATIAS JR, R. An Analytical Modeling Framework to Evaluate Converged Networks Through Business-oriented Metrics. **Reliability Engineering & System Safety**, [S.l.], 2013.

GUSEV, M. et al. CPU Utilization while Scaling Resources in the Cloud. In: CLOUD COMPUTING 2013, THE FOURTH INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, GRIDS, AND VIRTUALIZATION. **Anais...** [S.l.: s.n.], 2013. p.131–137.

HAMBY, D. M. A Review Of Techniques For Parameter Sensitivity Analysis Of Environmental Models. **Environmental Monitoring and Assessment**, [S.l.], p.135–154, 1994.

HAVERKORT, B. R. **Lectures on formal methods and performance analysis**. New York, NY, USA: Springer-Verlag New York, Inc., 2002. p.38–83.

HOFFMAN, F.; GARDNER, R. Evaluation of Uncertainties in Environmental Radiological Assessment Models. In: TILL, J.; MEYER, H. (Ed.). **Radiological Assessments**: a textbook on environmental dose assessment. Washington, DC: U.S. Nuclear Regulatory Commission, 1983. Report No. NUREG/CR-3332.

JAIN, R. **The Art Of Computer Systems Performance Analysis**: techniques for experimental measurement, simulation and modeling. [S.l.]: Wiley India Pvt. Ltd., 2008.

JMETER, A. **Overview**. Available on `http://jmeter.apache.org/index.html`.

KLEINROCK, L. **Queueing Systems**. New York: Wiley, 1975. v.1.

KOHLAS, J. Numerical computation of mean passage times and absorption probabilities in Markov and semi-Markov models. **Zeitschrift für Operations Research**, [S.l.], v.30, n.5, p.A197–A207, 1986.

LAST.FM. **Last.fm**. Available on `http://www.lastfm.com/`.

LEITNER, P.; HUMMER, W.; DUSTDAR, S. Cost-Based Optimization of Service Compositions. **Services Computing, IEEE Transactions on**, [S.l.], v.6, n.2, p.239–251, Nov. 2011.

LI, Y.; FANG, J.; XIONG, J. A Context-Aware Services Mash-Up System. In: SEVENTH INTERNATIONAL CONFERENCE ON GRID AND COOPERATIVE COMPUTING, 2008., Washington, DC, USA. **Proceedings. . .** IEEE Computer Society, 2008. p.707–712. (GCC '08).

LIN, C.-C. et al. Automatic Resource Scaling Based on Application Service Requirements. In: CLOUD COMPUTING (CLOUD), 2012 IEEE 5TH INTERNATIONAL CONFERENCE ON. **Anais. . .** [S.l.: s.n.], 2012. p.941–942.

LOOKBUSY. **Lookbusy – a synthetic load generator**. 2013.

MA, H. et al. QoS-Driven Service Composition with Reconfigurable Services. **Services Computing, IEEE Transactions on**, [S.l.], v.6, n.1, p.20–34, 2013.

MACIEL, P. et al. **Performance and Dependability in Service Computing**: concepts, techniques and research directions. [S.l.]: IGI Global, 2011.

MARSAN, M. A.; CONTE, G.; BALBO, G. A class of generalized stochastic Petri nets for the performance evaluation of multiprocessor systems. **ACM Trans. Comput. Syst.**, New York, NY, USA, v.2, p.93–122, May 1984.

MARSAN, M. A. et al. **Modelling with Generalized Stochastic Petri Nets**. 1st.ed. New York, NY, USA: John Wiley & Sons, Inc., 1994.

MATOS, R. et al. Sensitivity analysis of a hierarchical model of mobile cloud computing. **Simulation Modelling Practice and Theory**, [S.l.], v.50, p.151–164, 2014.

MATOS, R.; MACIEL, P.; SILVA, R. QoS-driven optimisation of composite web services: an approach based on grasp and analytical models. **International Journal of Web and Grid Services**, [S.l.], v.9, n.3, p.304–321, 2013.

MATOS, R. S. et al. Sensitivity analysis of server virtualized system availability. **IEEE Transactions on Reliability**, [S.l.], v.61, n.4, p.994–1006, 2012.

MELL, P.; GRANCE, T. The NIST definition of cloud computing. , [S.l.], 2011.

MINITAB, I. **Meet Minitab 16**. 2013.

MOLLOY, M. K. Performance Analysis Using Stochastic Petri Nets. **IEEE Trans. Comput.**, Washington, DC, USA, v.31, p.913–917, September 1982.

MONTGOMERY, D. C. **Design and Analysis of Experiments**. 8th.ed. New York: John Wiley and Sons, 2012.

MUPPALA, J. K.; TRIVEDI, K. S. GSPN models: sensitivity analysis and applications. In: ACM-SE 28: PROCEEDINGS OF THE 28TH ANNUAL SOUTHEAST REGIONAL CONFERENCE, New York, NY, USA. **Anais...** ACM, 1990. p.25–33.

MURATA, T. Petri nets: properties, analysis and applications. **Proceedings of the IEEE**, [S.l.], v.77, n.4, p.541–580, Apr 1989.

NIST. **NIST Cloud Computing Standards Roadmap**. 2013.

OU, Y.; DUGAN, J. B. Approximate Sensitivity Analysis for Acyclic Markov Reliability Models. **IEEE Trans. on Reliab.**, [S.l.], n.2, June 2003.

REN, Y. et al. Reliability Prediction of Web Service Composition Based on DTMC. In: THIRD IEEE INTERNATIONAL CONFERENCE ON SECURE SOFTWARE INTEGRATION AND RELIABILITY IMPROVEMENT, 2009., Washington, DC, USA. **Proceedings...** IEEE Computer Society, 2009. p.369–375. (SSIRI ’09).

ROSS, S. **Introductory Statistics**. [S.l.]: Elsevier Science, 2010.

SATO, N.; TRIVEDI, K. S. Stochastic Modeling of Composite Web Services for Closed-Form Analysis of Their Performance and Reliability Bottlenecks. In: SERVICE-ORIENTED COMPUTING, 5., Berlin, Heidelberg. **Proceedings...** Springer-Verlag, 2007. p.107–118. (ICSOC ’07).

SILVA, A. et al. Avaliação de Desempenho da Composição de Web Services usando Redes de Petri (in Portuguese). In: PROCEEDING OF THE BRAZILIAN SYMPOSIUM ON COMPUTER NETWORKS AND DISTRIBUTED SYSTEMS - SBRC 2006, Curitiba. **Anais...** [S.l.: s.n.], 2006.

SILVA, B. et al. ASTRO: an integrated environment for dependability and sustainability evaluation. **Sustainable Computing: Informatics and Systems**, [S.l.], v.3, n.1, p.1–17, 2013.

SOTOMAYOR, B. et al. Virtual Infrastructure Management in Private and Hybrid Clouds. **Internet Computing, IEEE**, [S.l.], v.13, n.5, p.14–22, Sept 2009.

SOUSA, E. et al. Evaluating Eucalyptus Virtual Machine Instance Types: a study considering distinct workload demand. In: CLOUD COMPUTING 2012, THE THIRD INTERNATIONAL CONFERENCE ON CLOUD COMPUTING, GRIDS, AND VIRTUALIZATION. **Anais...** [S.l.: s.n.], 2012. p.130–135.

STEWART, W. J. **Introduction to the Numerical Solution of Markov Chains**. [S.l.]: Princeton University Press, 1994.

SULEIMAN, B.; VENUGOPAL, S. Modeling Performance of Elasticity Rules for Cloud-Based Applications. In: ENTERPRISE DISTRIBUTED OBJECT COMPUTING CONFERENCE (EDOC), 2013 17TH IEEE INTERNATIONAL. **Anais...** [S.l.: s.n.], 2013. p.201–206.

TRIVEDI, K. S. **Probability and Statistics with Reliability, Queuing, and Computer Science Applications**. New York: John Wiley and Sons, 2001.

VOAS, J.; ZHANG, J. Cloud Computing: new wine or just a new bottle? **IT Professional**, [S.l.], v.11, n.2, p.15–17, March 2009.

WATSON J.F., I.; DESROCHERS, A. Applying generalized stochastic Petri nets to manufacturing systems containing nonexponential transition functions. **Systems, Man and Cybernetics, IEEE Transactions on**, [S.l.], v.21, n.5, p.1008–1017, Sep 1991.

YANG, J. et al. Workload Predicting-Based Automatic Scaling in Service Clouds. In: CLOUD COMPUTING (CLOUD), 2013 IEEE SIXTH INTERNATIONAL CONFERENCE ON. **Anais...** [S.l.: s.n.], 2013. p.810–815.

YIN, B. et al. Sensitivity analysis and estimates of the performance for M/G/1 queueing systems. **Perform. Eval.**, Amsterdam, The Netherlands, The Netherlands, v.64, n.4, p.347–356, 2007.

ZHANG, Q.; CHENG, L.; BOUTABA, R. Cloud computing: state-of-the-art and research challenges. **Journal of Internet Services and Applications**, [S.l.], v.1, n.1, p.7–18, 2010.

ZHONG, D.; QI, Z.; XU, X. **Petri Net, Theory and Applications**. [S.l.]: I-Tech Education and Publishing, 2008. Available on: `http://www.intechopen.com/books/petri_net_theory_and_applications`.

Appendix

A

Script to Measure Auto Scaling Monitoring Time

```bash
#!/bin/bash
lookbusy(){
    beginTimeMetric=$(($(date +%s%N)/1000000))
    ssh -i /home/frontend/Downloads/acesso.pem 192.168.0.171 "
        lookbusy -c 50" &
    #checking VM Pending
    verifyVmPending $beginTimeMetric
}
verifyVmPending(){
   booting=TRUE
   while [ $booting = TRUE ]
   do
     pending=`euca-describe-instances | grep pending | wc -l`
     if [ $pending -gt 0 ]
     then
       beginTimeVmPending=$(($(date +%s%N)/1000000))
       ssh -i /home/frontend/Downloads/acesso.pem 192.168.0.171 "
           killall -9 lookbusy"
       durationDetectionAction=$(($beginTimeVmPending-
           $beginTimeMetric))
       echo $durationDetectionAction >> timess.txt
       booting=FALSE
     fi
  done
}
#****MAIN****
read -p "Number of samples: " numSamples
num=0
while [ $num -lt $numSamples ]
do
```

```
   #checks before if there are any VM pending and if the alarm status is OK, only to then run the next result
   booting=TRUE
   while [ $booting = TRUE ]
   do
     pending=`euca-describe-instances | grep pending | wc -l`
     statusAlarm=`euwatch-describe-alarms | grep ALARM | wc -l`
     if [ $pending -eq 0 ] && [ $statusAlarm -eq 0 ]
      then
       euscale-set-desired-capacity scale -c 9
       #invokes the synthetic load generator lookbusy
       lookbusy
       booting=FALSE
      fi
   done
   #to always keep the number of instances required to be monitored, in this case 1
   euscale-set-desired-capacity scale -c 1
   #wait for the alarm to return to its status OK and only then run the next result
   booting=TRUE
   while [ $booting = TRUE ]
   do
     statusAlarm=`euwatch-describe-alarms | grep ALARM | wc -l`
     if [ $sta  tusAlarm -eq 0 ]
      then
       booting=FALSE
      fi
   done
   #wait some random time to start new trigger the load generator lookbusy
   sleeping=$((($RANDOM % 10)*60))
   sleep $sleeping
   num=$(($num+1))
done
```

B
Script to Measure VM Instantiation Time

```
#!/bin/bash
killVMs(){
   instances=`euca-describe-instances | grep INSTANCE | awk '{print
      $2}'`
   euca-terminate-instances $instances
   sleep 10
   booting=TRUE
   while [ $booting = TRUE ] do
     running=`euca-describe-instances | grep running | wc -l`
     shutting=`euca-describe-instances | grep shutting-down | wc -l`
     kvm=`ssh 192.168.0.5 ps aux | grep  /usr/libexec/qemu-kvm | wc -
        l`
     if [ $running -eq 0 ] && [ $shutting -eq 0 ] then
        if [ $kvm -eq 0 ] then
           instances=`euca-describe-instances | grep INSTANCE | awk
              '{print $2}'`
           euca-terminate-instances $instances
           sleep 10
           booting=FALSE
   fi fi done
}

verifyVmUp(){
   booting=TRUE
   while [ $booting = TRUE ] do
      running=`euca-describe-instances | grep running | wc -l`
      if [ $running -gt 0 ] then
         beginTimeUp=$(($(date +%s%N)/1000000))
         echo "Begin Time VM UP:  "$beginTimeUp >> log_times.txt
         duration_kvm=$(($beginTimeUp-$beginTimeKvm))
         echo "Duration Preparation KVM: "$duration_kvm >> log_times.
            txt
         booting=FALSE
   fi done
}

verifyKvm(){
  c=TRUE
  while [ $c = TRUE ] do
    filter=`ssh 192.168.0.5 ps aux | grep  /usr/libexec/qemu-kvm | wc
        -l`
    if [ $filter -eq 1 ] then
       beginTimeKvm=$(($(date +%s%N)/1000000))
       echo "Begin Time KVM: "$beginTimeKvm >> log_times.txt
       duration_download=$(($beginTimeKvm-$beginTimeInstance))
       echo "Duration Download: "$duration_download >> log_times.txt
       c=FALSE
  fi done
  echo "Checking VM UP" >> log_times.txt
  verifyVmUp $beginTimeKvm
}

instancesVms(){
  beginTime=$(($(date +%s%N)/1000000))
  echo "Begin Time: "$beginTime >> log_times.txt
  echo "Instantiating" >> log_times.txt
  euca-run-instances -t m3.xlarge -z CLUSTER01 -k default emi-4
      F4B3CE2
  beginTimeInstance=$(($(date +%s%N)/1000000))
  echo "Begin Time Instance CC: "$beginTimeInstance >> log_times.txt
  duration_instance=$(($beginTimeInstance-$beginTime))
  echo "Duration Creation Instance CC: "$duration_instance >>
      log_times.txt
  echo "Checking if the KVM started" >> log_times.txt
  verifyKvm $beginTimeInstance
}

#************ MAIN ************
read -p "Number of samples: " numSamples
num=0
while [ $num -lt $numSamples ] do
  echo "Sample Number: "$(($num+1)) >> log_times.txt
  killVMs
  instancesVms
  echo $duration_instance";"$duration_download";"$duration_kvm >>
      durations.txt
  echo "END" >> log_times.txt
  num=$(($num+1))
done
```