POLITECNICO DI MILANO
Corso di Laurea Magistrale in Ingegneria Informatica
Dipartimento di Elettronica e Informazione e Bioingegneria

Cost-effective adaptation plan synthesis for cloud deployments of software applications

Software Engineering Laboratory

Relatore: **Prof. Raffaela MIRANDOLA**
Correlatore: **Diego PEREZ PhD**

Tesi di Laurea di:
Francesco PEGORARO, matricola 782004

Anno Accademico 2012-2013

Ai miei genitori

Ringraziamenti

Ringrazio innanzitutto Prof. Mirandola per avermi offerto la possibilità di svolgere questa Tesi e per la sua costante guida e incoraggiamento.

Ringrazio quindi Diego, che mi ha seguito con costanza, dedizione e simpatia durante tutto lo svolgimento. Non potevo essere seguito meglio.

Ringrazio la mia ragazza, Chiara, e tutti i miei amici di Origgio, Como e Milano che mi hanno aiutato e sostenuto durante tutto il percorso di questi 5 meravigliosi anni.

Infine, un grazie ai miei genitori, ai quali devo tutto.

Francesco Pegoraro, Origgio

Abstract

Cloud Computing has become a feasible alternative to in-house servers, thanks to a series of enabling factors: increase of global Internet connectivity, increase of hardware power, development of new virtualization techniques. Nowadays there are numerous online providers that offer VMs among a variety of cloud services and more providers rise on the market every year.

Unfortunately, deploying appropriately a service on online VMs is not an automatic task. It requires the knowledge of many parameters, which the common user does not know. Concretely, the proper usage of cloud elasticity is a method that can drive the cloud client to save money with respect on in-house deployment or to spend a fortune. This thesis concentrates on cost-effective adaptation plan synthesis for cloud deployments of software applications for a service that has to be deployed using VMs offered by an online provider, as it is the most difficult to estimate.

After modeling the prices of various cloud providers in a standard form and obtaining them through an application specifically developed for this purpose, the plan synthesis process takes care of calculating the VMs that will be required by the service to deal with a variable workload that includes periods of burstiness. The plan synthesis is divided in two sub-processes, one for reserved VMs and one for on-demand VMs, to better exploit the intrinsic characteristics of each type of reservation.

Sommario

Il Cloud computing è diventata una valida alternativa rispetto a mantenere i server interni all'azienda, grazie a una serie di fattori: l'incremento della connettività globale a Internet, l'incremento di potenza dell'hardware, lo sviluppo di nuove tecniche di virtualizzazione. Al giorno d'oggi ci sono vari online provider che offrono, tra i vari servizi cloud, L'uso di VMs online, e ci si aspetta la presenza di nuovi provider sul mercato ogni anno.

Sfortunatamente, eseguire il deploy in modo appropriato di un servizio su VMs online non è un compito automatico. Questo, infatti, richiede infatti la conoscenza di molti parametri, cosa che l'utente comune non sa. Questa tesi si concentra sulla sintesi di un piano di adattamento adatto a gestire l'elasticità delle virtual machines, per un servizio che deve essere installato usando VMs offerte da un provider online, in quanto è uno dei parametri più difficili da stimare.

Dopo aver modellato i prezzi di vari cloud provider e averli ottenuti mediante una applicazione specificatamente creata a questo proposito, il processo di sintetizzazione del piano si occupa di calcolare il numero di VM che saranno richieste per sostenere il carico di lavoro variabile, che include periodi di burst dell'applicazione. Questo processo di sintetizzazione del piano è diviso in due sotto-processi, uno per le VM riservate e uno per le VM on-demand, per meglio sfruttare le caratteristiche intrinseche di ogni tipo di prenotazione.

Contents

List of Figures

List of Tables

List of Appendices

List of Symbols

API (Application Programming Interface) indormation that specifies how some software components should interact with each other.

AWS (Amazon Web Services) a collection of remote computing services that together make up a cloud computing platform, offered over the Internet by Amazon.com.

CPU (Central Processing Unit) hardware within a computer that carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system

DNS (Domain Name System) is a hierarchical distributed naming system for computers. It translates easily memorized domain names to the numerical IP addresses needed for the purpose of locating computer services and devices worldwide.

FIFA (Fédération Internationale de Football Association) international governing body of association football, futsal and beach soccer.

HTML (HyperText Markup Language) markup language for creating web pages that are displayed in a web browser

IIS (Internet Information Services) extensible web server created by Microsoft for use with Windows NT family

Java a programming language, one of the most common in the world.

Javascript a interpreted computer programming language, mostly used in web browsers

JSON (JavaScript Object Notation) is a lightweight data-interchange format

JVM (Java Virtual Machine) executes Java bytecode. It is specific for each computer architecture and is the key technology that allows the portability of Java programs.

Matlab a numerical computing environment that allows matrix manipulation, plotting of

functions and data, and implementations of algorithms

OS (Operating System) collection of software that manages computer hardware resources and provide common services for computer programs

SLA (Service Level Agreement) Contract between a customer one (or more) service provider where the provider agrees to provide the client a service that respects the quality bounds defined in the contract.

XML (eXtensible Markup Language) a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

Chapter 1

Introduction

"Engage"

Capt. Jean-Luc Picard

Cloud computing is a competitive alternative to deploy software applications of a company, principally thanks to the elasticity offered by the cloud infrastructure. It is in fact possible, and should be exploited, to acquire resources only when needed, like in the case of the worst case scenario of the application, and then releasing them when there is no more the necessity of such a quantity of resources. The list of cloud computing providers is long and increasing each year. Choosing one provider is not a trivial task, as many factors come into play in the decision: costs, servers' geographic location, performance, ...

Naturally, cloud providers offer some ways of calculating their VMs' costs, but the tools provided are neither powerful nor flexible enough to properly aid the customer in the selection, as require deep knowledge of how and when the elasticity will be used.

As a matter of fact VMs' costs depends greatly on how the company makes use of the cloud elasticity: a naive adaptation plan can easily lead to spend a fortune, maybe even without reaching the performance desired, while a wise plan for using the elasticity can allow the organization to meet the required performance at reasonable costs. Oddly, the same plan that leads to excellent performance and low costs on a concrete system under a concrete type of workload can perform poorly on a slightly different workload, making the job of deciding the plan that will guide the elasticity very challenging.

Studies on Internet services supported workloads showed that there is a boosty nature in them, meaning that a large amount of requests are received in a small portion

Figure 1.1: Relations between user, service provider and cloud provider

of time. The spikes happen unpredictably. This behavior can, and should, be managed automatically, as the amount of traffic varies very quickly, too quickly to be managed by a human.

In Figure 1.1 are present the three main actors that are present in this thesis. The first one from the right is the *Cloud Provider*, offering cloud services over the Internet. A partial list of online cloud providers is in Appendix A. The *Service Provider* is the actor that uses the cloud resources offered by the *Cloud Provider* to provide a service to its own users. Because of that, it is also called *Cloud Client*. The third actor is the *End User*, which uses the service provided by the *Cloud Client*. For the rest of the thesis will be used consistently the term *client* to indicate the *Cloud Client* and the term *user* to indicate the *End User*.

1.1 Cloud Computing

The world of cloud computing is vast and in rapid development. Every year new providers appear on the scene, while the existing ones consolidate their presence on the market. New products and offers are announced with high frequency and many companies are planning to move part of their systems on online servers. This parading change is the result of the growth of new needs inside the companies. Larger companies tend to create a private cloud, with dedicated server or even datacenters, to be able to fully control the data flow and the security measures. Medium and small companies, however, see in the public cloud a new opportunity to increase the efficiency, reduce the IT costs and give the employees new services.

But what is, concretely, cloud computing? As the National Institute of Standards and Technology[23] states, it is "an expression used to describe a variety of computing concepts that involve a large number of computers connected through a real-time communication network such as the Internet". It continues stating that "the popularity of the term can be attributed to its use in marketing to sell hosted services in the sense of application service provisioning that run client server software on a remote location". Amazon[2], instead, defines cloud computing as "the on-demand delivery of IT resources via the Internet with pay-as-you-go pricing.".

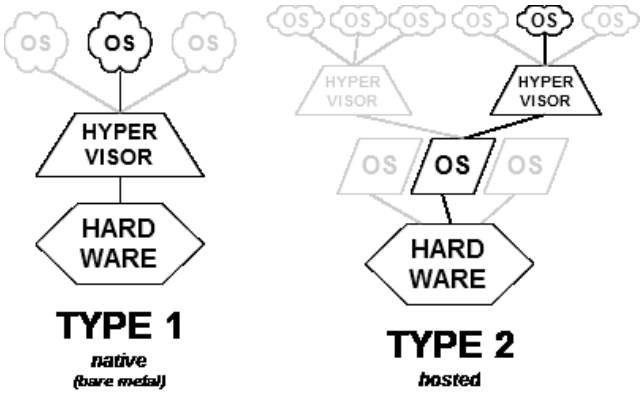


Figure 1.2: Two types of hypervisor. Image taken from [32]

The explosive growth of cloud providers has been partially driven also by the innovations in the hardware department. Computing power has never been cheaper, servers have become more powerful and energy efficient, leading to the reduction of the number of machines used. Services that 10 years ago needed to be deployed on dedicated machines are now installed on the same server, because it is powerful enough to sustain them.

Software innovations also made possible cloud computing. In particular, virtualization technologies have driven the rise of cloud computing since the year 2000, pioneered by Amazon. Their use enabled the usage of a single physical server by multiple independent OS assigned to different users. Before their massive usage, a single OS had to be installed on a physical machine and then users shared the environment, with the isolation provided by the OS. Each physical machine was limited to install a single OS. Nowadays virtual machines works thanks to hardware virtualization, a piece of software that hides the physical characteristics of a computing platform and shows another abstract computing platform on which the OS are installed. This job is done by *hypervisors*. There are two types of hypervisors, as showed in Figure 1.2. Type 1 installs itself directly on the hardware, allowing the execution of multiple OS at the same time and giving the illusion to each OS to own the full machine resources. In type 2, instead, the hypervisor is a software installed inside a OS that has the full hardware control. Naturally, type 1 hypervisors are much more efficient because they don't have the overhead of having an OS layer between them and the hardware resources. Cloud providers virtual machines are on type 1 hypervisors.

Regarding the changes in management for a company that chooses to "go to the cloud": it depends on the service that is moved to the cloud. Let it be, for example, a mail server. In the past, a small company has usually had its private physical mail server, managed by an external consultant because there is no IT dedicated employee. So, there are hardware costs for the physical machines; consultant costs; electricity and general

server maintenance costs; licensing costs for the software installed; security costs. Data is in a single physical location, vulnerable to accidents like a fire that can compromise the work of years in a matter of seconds. Moreover, access to the data is limited by the upload bandwidth of the company's Internet connection. In the event of a company expansion, the adjustment of the mail server is not so automatic and could imply buying new hardware equipment. Finally, protection from Internet attacks is not trivial and it has to be delivered by a specialist.

Moving the mail server to the cloud can certainly help solve some of the problems. First of all, data is always accessible from any browser with just an Internet connection. There is no more need to worry about backups, hardware maintenance and electricity bills. Adding more power is just a matter of minutes, thanks to the pay as you use model. Unfortunately, all that is gold does not glitter. Dimensioning the online servers is not automatic and there are no standard/universal procedures that can accomplish this goal.

Let me present a simile between computing servers performances and water-transportation pipes. Having a service on standard in-house servers that users request over the Internet is similar to owning a pipe where some water has to flow. The water represents the users that connect to the server. The size of the pipe corresponds to the server's capacity/power: the larger the pipe, the higher the capacity/power. If the flow of water a.k.a. users is below the pipe's limit, everything runs smoothly. An interesting parallel with this metaphor is that even with the pipes, as the flow of water is above 80% the pipe's area, it begins to create turbulence that makes it difficult for the transmission of liquid. This is very similar to what is experienced on a server under high loads, as the queue theory states, for high loads of a server, the response time experienced by the user boosts as the queue of requests waiting to be served grows exponentially. If there is, for any reason, a temporary increase in the flow that would need three times the pipe's area to be managed, it is impossible to ensure the flow required because pipe's dimensions are fixed. The same happens with a standard in-house server: it is impossible to sustain a peak of accesses that exceeds its limits. Well, cloud computing can help solve this situation, by temporarily acquiring new servers, just for the peak of requests, and then releasing them when they are not needed anymore. It is like having the ability to resize the pipe's area!

1.2 Thesis contributions

This thesis deals with the challenge exposed in the previous sections. It assists users to use cloud elasticity and to predict their cloud deployment costs. Using a trace log that represents the expected workload (including periods of burstiness), it uses formal theories to calculate an efficient adaptation plan that manages the cloud elasticity in a cost-effective manner. This efficient plan is then offered to the cloud client for its usage. As the consideration at once of every type of elastic resource that cloud providers offer and for which a cloud client is charged would make the study unmanageable, this thesis focus on the charge for the usage of virtual machines (VMs), being the them most unpredictable of all the resources. In fact, predicting the costs of other cloud components like load balancers, storage spaces, network bandwidth, etc. is easier due to the fact that all these components are quantifiable with a high degree of confidence due to their reduced volatility.

1.3 Structure of the thesis

Chapter 2, *Background*, provides the required background needed to understand the topics exposed in the thesis.

Chapter 3, *Proposed approach*, describe the challenge and propose a method to address and resolve them. This method is composed of three steps, two of which have been implemented in this thesis.

Chapter 4, *Cost model retrieving*, explains the first step of the proposed approach, that is what is, how and why it is produced a cost model for a cloud provider. It also introduces the program implemented.

Chapter 5, *Plan synthesis*, describe the second step of the proposed approach. It explains how it the plan is synthesized.

Chapter 6, *Example*, shows a real life example of both cost model retrieving and plan generator programs.

Chapter 7, *Conclusions*, draws conclusions on the work done in this thesis.

Chapter 2

Background

Three Laws of Robotics

1. *A robot may not injure a human being or, through inaction, allow a human being to come to harm.*
2. *A robot must obey the orders given to it by human beings, except where such orders would conflict with the First Law.*
3. *A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.*

Isaac Asimov

In this chapter we describe the theories and the concepts that constitute a necessary prerequisite to understand the problem and developed the solution proposed in this thesis.

2.1 Performance evaluation techniques

Modern computer systems are evolving at a very high pace, increasing their importance every year in all kind of business operation. They are ubiquitous and their presence is almost granted. Just looking back a few years you realize how much computer systems have evolved and how crucial are in everyday life. As a result, the request of tools to understand and predict their behavior raised as well.

Lazowska et al. [20] identifies three approach types to deal with this need:

1. Use of *intuition and trend extrapolation.*

2. Use of *experimental evaluation of alternatives*

3. Use of *models*

The first two approaches represent the extremes, one solely based on human experience and intuition, rapid and inexpensive but inaccurate, while the other relies on laborious trials, accurate but most of the time not cheap. The third approach, the use of *models*, is a sort of compromise between the first two. A model is an abstraction of a system: an attempt to identify those aspects that are essential to the system's behavior. Once defined, the model can be *parametrized* to reflect any of the alternatives and then *evaluated* to determine its behavior.

Creating a model takes a certain effort in term of time and resources spent, but once it is defined, parametrized, stabilized and tested it can predict with accuracy a wide set of situations. It is more reliable than intuition because it is formally defined. It is also more flexible than experimental evaluation (i.e. the so called *what if* scenarios) because, thanks to the parametrization, very different examples can be tested in matter of minutes, while on actual experiments it may take hours or days.

Modeling, then, provides a framework for gathering, organizing, evaluating, and understanding information about a computer system. Among the multiple modelling languages that exist fore performance evaluation, in this thesis has been followed the theories of the *Queueing Networks models*.

2.1.1 Queueing networks models

Queueing networks modelling is a different approach to computer systems modelling, where the computer system is modeled as a network of queues and is then evaluated analytically[20]. Each resource is modeled as a service center, receiving customers that are temporarily stored in a queue, if necessary. The customer is served when is his turn, than it can depart from the service center. A service center is depicted in Figure 2.1.

The simplest possible queueing model consist of two parameters. The first one is called *workload intensity* and it could be of two types: open and closed. In case it is of type *closed*, the number of users in the system is fixed. In case it is of type *open*, the parameter is called *arrival rate*, identified with the symbol λ, and indicates the rate at which customers arrive. For example, customers can arrive at an arrival rate

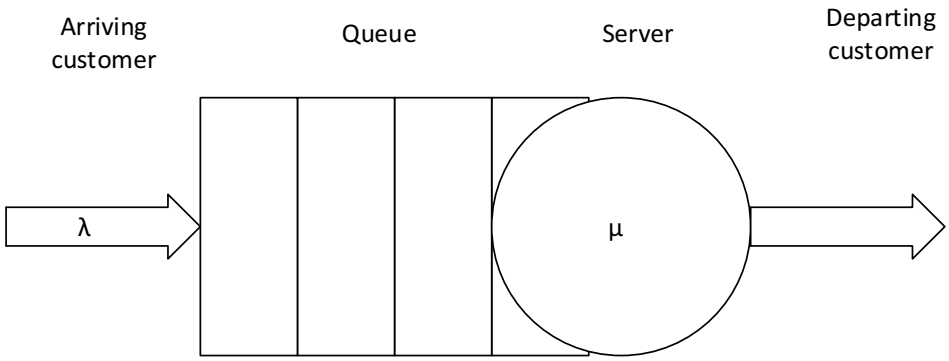


Figure 2.1: A single service center with open workload

$\lambda = 0.5\,^{\text{customers}}/_{\text{s}}$.
The second parameter is the *service demand*, which is the average service requirement of a customer. For example, each request takes 1.25 seconds to be served. From this parameter it is usually derived the *service rate*, which is the number of customers served each second by the server. It is indicated with the symbol μ and is the inverse of the *service demand*; i.e., with the previous *service demand* of 1.25 seconds, the corresponding *service rate* is $\mu = 1.25^{-1} = 0.8\,^{\text{jobs}}/_{\text{s}}$.

This simple yet powerful notation allows the calculation of four performance measures:

1. *service utilization*: proportion of time the server is busy

2. *response time*: average time spent by a customer in the server, both waiting in the queue and receiving service

3. *queue length*: average number of customers in the server, both waiting in the queue and receiving service

4. *throughput*: rate at which customers pass through the service center

For the parameters stated earlier, that is $\lambda = 0.5\,^{\text{customers}}/_{\text{s}}$ and $\mu = 0.8\,^{\text{jobs}}/_{\text{s}}$, these performance values measures are $utilization = 0.625$, $residence\,time = 3.33s$, $queue\,lenght = 1.67\,customers$ and $throughput = 0.5\,^{\text{customers}}/_{\text{s}}$.

The important fact to remark, as done by Prof. Mor Harchol-Balter[17], is that queueing networks theory is a very powerful *predictive* tool. In fact, it can be used to assist the decision when, for example, there is the need to update a computer system. Queueing theory can identify the components that, if updated, can significantly boost the system's performance and also demonstrate how other components, even if updated, do

not speed up the system. But this model does not limit its influence on prediction on existing systems as it can also be a very useful *design* tool. In fact, using this model is not uncommon to find out how some counterintuitive choices can bring increase in the performance in spite of what the experience and the intuition would have suggested. The public available first chapter of Prof. Mor Harchol-Balter's book [17] contains a series of useful examples showing how solutions based on intuition can be mistaken.

In this thesis it is used this type of models to predict the user experience (in terms of response time) with a variable number of servers, each of them with a particular *service rate*.

2.2 Self-adaptive systems

The explosive growth of information and its natural integration with technology require new and innovative approaches for building, running and managing software systems. In addition to the increasing complexity, software systems should also become more versatile, flexible, resilient, dependable, configurable and self-optimizing by adapting to changes in the environment they are build for and modifications of the system requirements. In one word, software systems should become *self-adaptive*, that is able to modify their behavior and/or structure in response to their perception of the environment, the system and the requirements. This is an active research area, as demonstrated in [7, 13, 15, 8].

An important remark is that the only common element among the various initiative to explore self-adaptability behavior is usually the software, mostly thanks to its flexibility. Still, the proper implementation of self-adaptive software applications is a formidable intellectual challenge. Cheng et al. [7] and then de Lemos et al. [13] identify four principal research topics:

- design space for adaptive solutions
- processes
- from centralized to decentralized control
- practical run-time verification and validation

The website [15], instead, provides information about software engineering for self-adapt systems and contains a wiki aimed to facilitate communication and idea exchange for adaptive and self-managing systems.

2.3 Cloud Computing

Cloud computing has become an increasingly popular paradigm during the last years. It consists of renting computing power from online providers in exchange of a fee. One of the property of a cloud computing is its *elasticity*, which has been defined by Herbst et al.[**?**] as *the degree to which a system is able to adapt to workload changes by provisioning and deprovisioning resources in an autonomic manner, such that at each point in time the available resources match the current demand as closely as possible.*

As stated in the Introduction, the use of cloud computing is the focal point of this thesis. With an accurate use of the elasticity offered by the cloud infrastructure, it is possible for users to deploy a service using cloud computing, granting excellent user experience at a lower cost than in-house servers.

Cloud providers offer to their user web pages where it is possible to calculate the total cost of ownership of the deployed servers. The web pages for Amazon, Rackspace and Azure calculators are available respectively at [4], [30] and [25]. The common problem in all these pages is that there are requested a set of system usage parameters that are unfeasible to be known by the average cloud client. A non-expert selection of resources can make the client spend a fortune in VMs and the application will not even satisfy its requirements.

With the approach proposed in this thesis, explained in Chapter 3 and detailed in Chapter 4 and Chapter 5, it is possible to calculate these parameters. Moreover, these parameters are tailored for the specific needs of the user. So, once the user knows these values, he can compare the costs of a cloud deployment for his application and draw conclusions of whether it is convenient or not.

2.4 Technologies used

To develop this thesis a series of different technologies has been utilized.

The first software used, in chronological order, was *Matlab*[22]. It is a numerical computing environment that allows matrix manipulation, plotting of functions and data, and implementations of algorithms. It has been very useful because, being an interpreted language, it is quicker to modify the code and to create models with respect to other, compiled, languages. Besides, it is useful to visualize the data and Matlab gives excellent tools to accomplish this job.

After working with Matlab, refining functions and procedures, the job became integrating these functions and procedures in *Java*[26] programs to take advantage of its portability and efficiency. Java is one of the most widespread programming languages in the world, based on the principle WORA (Write Once Run anywhere). To do so, the applications are compiled in a bytecode that can run on any JVM (Java Virtual Machine), regardless of computer architecture. It would seem not efficient having a virtual machine interpreting the bytecode, but this is proven to be not true as benchmarks demonstrate how Java programs are as efficient as C or C++ programs[5]. In fact, a function of this thesis passing from a Matlab implementation to a Java one became 10 times faster. Being programmed in Java, this thesis is easily enabled to be part of a more complex framework that study cloud providers usage.

Together with Java another key technology used was *XML*[10], which is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It has been used to import and export data and to write the configuration files for the programs.

Finally, *JSON*[1], a lightweight data-interchange format that derives from Javascript notation, has been used to retrieve and parse data from online locations.

Chapter 3

Proposed approach

"I did it for me. I liked it. I was good at it. And, I was really... I was alive."

Walter White, Breaking Bad 5x16

In this Chapter, I present the developed approach that deals with the challenges introduced in Chapter 1.

First, it is described a general overview of the challenge that this thesis solves and what is the proposed approach, schematized in Figure 3.1.
Will then be presented all the inputs that have been used in this thesis.

Chapter 4 discusses the role and implementation of the *Cost Model synthesis* block of Figure 3.1, together with the explanation of the Cost model.

Chapter 5 explains the role of the second block of Figure 3.1, *Adaptation Plan synthesis*. Its inputs, (SLA, Workload log and VM characteristics) are introduced in the current chapter and refined in Chapter 5.

3.1 Overview

The developed approach consists of three steps, as shown in Figure 3.1 together with the input data each one requires.

To consider the revenues and costs of the Internet service a cost model of the cloud providers is necessary. This could be done by working directly with the information

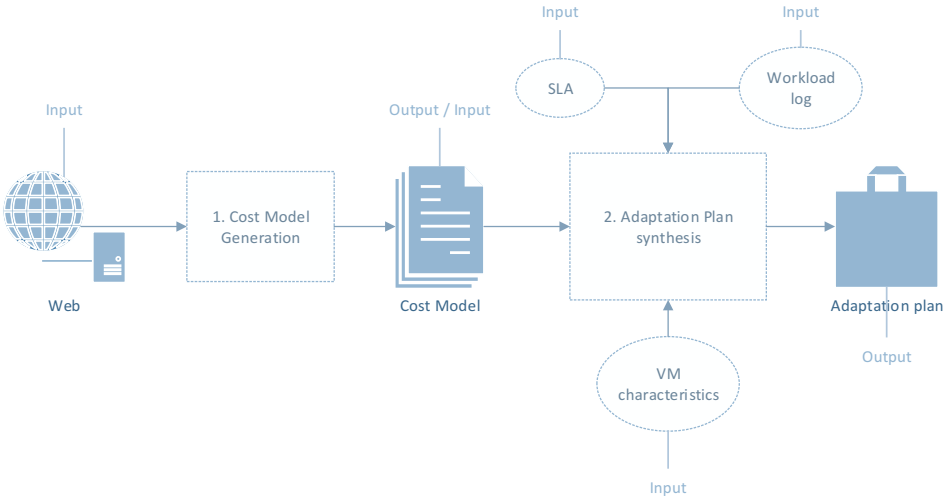


Figure 3.1: Overview of the approach

published on the web by service providers. However, this would make that service the core of our approach, with cost wired to a concrete cloud provider. Instead, using a cost-model that abstracts the different payment implementations done by each provider it is possible to generate a general process that works for any cloud provider.

Usually cloud providers write VMs prices in tables on their web pages, but each provider has its standard and so comparing prices can be not trivial. Sometimes even the tables are not enough, as they need to be completed by a text that explains how and when those prices are applied, text that only a human can understand. A uniform cost model brings homogeneity across different pricing standards, creating a uniformed format that programs can then access, in a completely transparent way.

The first step executes the synthesis of a cost model using data published by cloud providers regarding the billing options for VM usage. The cost-model is compliant with the cost metamodel proposed for representing VM billing characteristics and will be shown in Chapter 4.

The second step synthesizes an efficient adaptation plan for the application. This step requires information regarding: the cost model of the selected cloud provider, the expected workload the application will receive, the SLA of the application and performance characteristics of the application running in a VM of the cloud provider. This step is discussed extensively in Chapter 5.

The third step calculates the profit of the application executing under the synthesized adaptation plan. This step has not been implemented yet.

3.2 Inputs of the approach

In the next section I explain the Inputs required by the developed approach.

As depicted in Figure 3.1 with ovals, the inputs of the approach are:

- SLA
- Workload log
- VM characteristics

3.2.1 SLA

A service-level agreement (SLA) is a contract between a customer one (or more) service provider where the provider agrees to provide the client a service that respects the quality bounds defined in the contract. There exist many manners in which a service provider and a user agree the QoS and the price that the user is willing to pay for it. SLA can also indicate, besides the service level, the penalties that the provider incurs in case of violation of such levels. A SLA is important because it defines on a legal document the required application QoS and revenues; i.e., if the provider fails to sustain them it is possible to proceed legally against it. If a provider does not comply the SLA the client can also ask the termination of the contract.

To decide the type of SLA that my approach would accept, I have studied some different types of SLA used in the scientific community of Performance Evaluation. We describe some of these types of SLA in the next subsections. Finally I present the chosen SLA type and I motivate its election.

3.2.1.1 SLA type 1

A SLA can be defined as a maximum response time R and a percentage P. It would be formulated as "*response time shall be lower than R seconds for the P% of requests*". In this case, the SLA is complied (without paying any penalty) even if up to the $(100-P)\%$ of requests are served in more than R seconds. The penalty may be formulated as "*each request that exceeds the tolerable threshold P pays an amount C*".

To calculate the penalties, it is required to define a time interval $[T_i, T_f]$ in where

count all the requests and their response time. Then, count the "slow" ones, the one served in more than R seconds, and compare this value with the threshold value, the number of allowed "slow" requests, calculated as $numReq \cdot \frac{1-P}{100}$. If the number of slow requests are higher, pay the penalty.

This makes the calculation of penalties very easy once the data is ready. Unfortunately, this formulation has also limits. First of all, the penalty is always C, independently from how much time over R the request has been served. Secondly, it needs the definition of a time interval, and it must wait until the time interval is over to know if and how much penalty has been payed.

3.2.1.2 SLA type 2

It is possible to define a SLA as a maximum response time R, formulate as follows: "*response time shall be lower than R seconds*". There is a penalty of C monetary units for each requests that are above R seconds. This is a particular case of the type 1, where $P = 100\%$. In this case, it is not needed to wait the end of the time interval to pay the penalty, it can be payer as soon as the request is served in more than R seconds.

It is easy as in the previous type to calculate the penalties and moreover in this case the penalties are known as soon as the request is executed, without having to wait the end of the time interval. However, yt also shares a problem of type 1: the penalty C is fixed.

3.2.1.3 SLA type 3

Another possible definition of SLA involves multiple stages with different penalties. It would be formulated as "*the response time for a request shall be lower than R_1 seconds. If response time is between R_1 and R_2, the penalty for the request is C_1 monetary units. If response time is between R_2 and R_3, the penalty is C_2 monetary units... If response time is above R_N the penalty is C_{N-1} monetary units.*". The underling conditions are $R_1 < R_2 < ... < R_N$ and $C_1 < C_2 < ... < C_{N-1}$.

This SLA type is a generalization of type 2, where $N = 1$, so $R_1 = R$ and $C_N = C$.

This way of formulating the SLA has many benefits. First of all, it is still easy to calculate the penalties. Then, they can be calculated as soon as a request is served.

Property	Type 1	Type 2	Type 3	Type 4
Easy to calculate penalties	Y	Y	Y	N
Penalties calculated as soon as the request is served	N	Y	Y	Y
Penalty proportional to the request time	N	N	Y	Y
User can choose easily the parameters	Y	Y	N	N

Table 3.1: Properties comparison of different SLA types

Finally, it takes into account how much time a request has taken to execute: the penalty is not the same for any time over R seconds. The principal limitation of this type is that it is not easy to define the parameters R_i and C_i, especially for a normal user.

3.2.1.4 SLA type 4

Finally, the last type defines the SLA as a dense function, so it is possible to define a different penalty for each response time $R_1 \in \Re^+$. The SLA would be defined as "*response time shall be lower than R seconds. If the response time is $R_1 \geq R$ it is paid a penalty $C = f(R_1)$.*". The function f defines the penalty, and may be stepwise or may not be. This type is a generalization of type 3, where the number of R intervals is infinite.

This definition still have the good properties of returning the penalty as soon as the request is executed and having a the penalty proportional to the response time. Still, it is unlikely that a user can input a proper f function that models perfectly the behavior that should have the penalty. Moreover, it becomes difficult to calculate the penalties.

3.2.1.5 The chosen SLA

In this thesis it is used the *type 3* definition, "stepwise SLA", because is the one that best suites the needs. In Table 3.1 there is a comparison if the properties of the four SLA types.

More formally, it is written in the form

$$sla = ((r_1, m_1), (r_2, m_2), \ldots, (\infty, m_S)) \tag{3.1}$$

being $r_0 = 0$, $r_S = \infty$, $r_0 < r_1 < r_2 < \cdots < r_S$ and $m_1 > m_2 > \cdots > m_S$ where for each $\sigma \in [1, S]$, (r_σ, m_σ) represents that the service provider gets an income (revenue) of m_σ monetary units if the response time is below r_σ but not below $r_{\sigma-1}$. This type of description is yet simple and, since the number of components (r_σ, m_σ) in the SLA is

not restricted, it allows the representation of a wide set of types of agreements. Besides, this SLA definition accepts negative revenues, where provider loses money maybe due to damages in client's (or its own) interests caused by low performance, and it allows knowing the income for a request as soon as it is served.

Example Given $sla = ((1, 0.01), (2, 0), (\infty, -0.01))$, being t the response time of an execution and $f(t)$ the provider income:

$$f(t) \begin{cases} 0.01 & \text{if} \quad t < 1 \\ 0 & \text{if} \quad 1 \leq t < 2 \\ -0.01 & \text{if} \quad t \geq 2 \end{cases} \tag{3.2}$$

It models the situation where, if the response time is less than 1 second the revenue is equal to 0.01. If the response time is between 1 and 2 seconds, the revenue is zero. Finally, if the response time is greater than 2 seconds, the revenue is -0.01. This model is suitable to model a web server that has to return web pages of an online shop. If the page is returned quickly there are more chances that a customer will buy items on the store. If a website takes long time to return its pages it usually leads the customer to leave or change the website, resulting in a potential loss of money for the business owner. This is exactly what is written in the *sla*.

3.2.2 Workload log

The second and third step of the approach need to know the load history of the application in order to generate a plan, if the application to model already exists. To accomplish it, logs that represent the incoming workload to the application are used. In case of a new service, it is possible to use an expected workload. A log contains the incoming workload to the application in terms of count of requests during consecutive time intervals, where all time intervals are of the same length. Formally,

$$log = (I, (n_1, n_2, \ldots, n_N)) \tag{3.3}$$

where I is the length of the time interval and each $n_i, i \in [1, N]$ are the amount of requests during time interval i. This kind of information can be easily acquired from activity records of servers like *Apache http server* or *IIS*.

Example In the exemplification of the behavior of this approach it has been used the workload logs extracted from the information of FIFA 1998 World Cup website accesses

(logs available at [27]). It has been extracted the accesses during three million seconds to the servers of Paris region and counted the accesses over intervals of 10 seconds. The log is $log = (10s, (292, 241, \ldots, 181))$ and it contains $3 \cdot 10^5$ counts of requests. The complete log information is shown in Figure 3.2; it is possible to see that this log shows high variability of the workload over time.

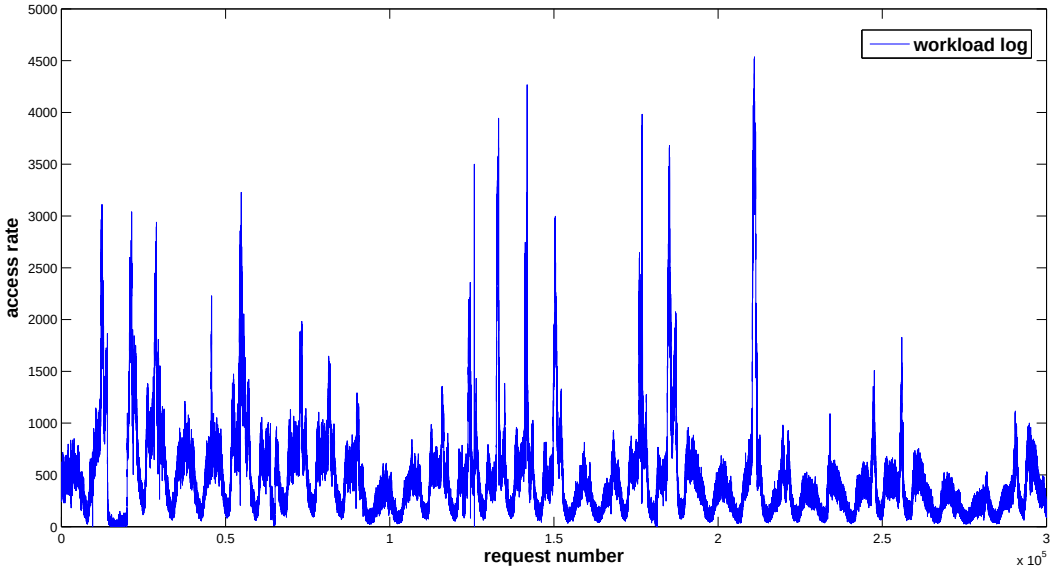


Figure 3.2: Workload log, requests every 10 seconds

3.2.3 VM characteristics

A key component for a correct resource estimation is the knowledge of the application's performance on each VM considered for deploying the software. Formally, assuming M types of VMs,

$$demand = [e_1, e_2, \ldots, e_M] \tag{3.4}$$

where $e_i > 0$ is the execution time in the i-th type of VM (VM_i). Given the fact that the number of VMs offered by the providers is very high, in the order of dozens of different types, trying to measure the performance of the same software on every VM is a tremendous effort from a economical point of view, but it's also time-consuming and not prone to automation. Moreover, the results are non reusable: they work only for the specific software tested at the specific version. Even changing the software version could alter the results. Finally, it is not feasible to count on clients providing this kind of information.

The solution to this problem is to measure only application's *time information* in one of the VMs and then to internally estimate the mean execution time of the application

in the other VMs.

To accomplish this task the program will need to know the relative performance among all different VMs or subcomponents of the VMs, so it will be possible to deduce the execution times e_i from the execution time obtained from a live test. There are numerous studies in this direction [11], dedicated in studying VM performance, as it is a not trivial field, but this is out of the scope of this thesis.

Example For exemplifying this approach, consider a completely CPU intensive application, and therefore consider only the relative CPU speed of VMs. Assume the existence of M different VMs, and the existence of a vector of relative CPU performance of each VM $CPU = [cpu_1; \dots cpu_i; \dots cpu_M]$ with $1 \leq i, j \leq M$. The meaning is that if $\frac{cpu_i}{cpu_j} = 2$, then VM_j requires double of time of CPU for serving a request than VM_i. The input from the client will be the mean time that the application needs to use the CPU to serve a request and the type of VM to which refer that CPU time; call t^j_{CPU} to the CPU time measurement obtained from the j-th type of VM. With this data, it is possible to calculate the values in demand vector by

$$e_i = t^j_{CPU} \frac{cpu_i}{cpu_j}, \forall 1 \leq i \leq M$$

To fill the CPU vector is possible to privately test the capabilities of VM subcomponents or being based on previous research results about cloud resources benchmarking, like those on Cloud Harmony website[9].

To compare an application that is not exclusively CPU intensive it is possible to add the time taken by each subcomponent to the sum of e_i. Usually the subcomponents used for the comparison, in addition to the CPU, are memory and disk, as their performance affect an application total running time. The generic formula to calculate a value i in *demand* vector becomes

$$e_i = \sum_{subcom \in \{CPU, memory, disk\}} (t^j_{subcom} \cdot \frac{subcom_i}{subcom_j})$$

Of course, if there are no subcomponents other than CPU, the calculation of e_i is reduced to the previous case.

Chapter 4

Cost model retrieving

"I don't know half of you half as well as I should like; and I like less than half of you half as well as you deserve"

Bilbo's farewell speech, The Fellowship of the Ring

This chapter will detail the necessity of a cost model retrieving, its implementation and the challenges found in the process.

4.1 Introduction

The need of an automatic *cost model retrieving* raises from the observation that a single provider like Amazon puts in the market more than 500 different VMs. A manual comparison between different VMs is therefore highly impractical and it is discouraged. Besides, there is the possibility of price changes or the introduction of new types of VMs by a vendor, making the cost retrieval a periodic task and not a one time job.

To allow the comparison of VMs from different providers there is the need of a model that could represent all the characteristics for which any cloud provider is interested in charging their clients of each VM. This is achieved with the Cost Metamodel, detailed in Section 4.2.

An automated cost retrieving program has been implemented, allowing the automatic synthesis of cost models for Amazon, Rackspace and HP Cloud Services, ready to be fed to the second step of the process, and it's specified in Section 4.3.

4.2 Cost metamodel

The three step process shown in Figure 3.1 requires in the second step information regarding how providers charge their clients for VM usage. Cloud providers publish this kind of information in their websites, usually in tables or in natural language. Comparing the published information of VM usage costs of different cloud providers it is easy to see that they do not follow an homogeneous procedure for billing. To make easier an automatic analysis that uses the billing information different cloud providers, it is advisable to have such information represented in the same language. That is why it has been developed an extensible metamodel for representing the billing information of cloud providers.

Studying the websites of cloud providers, such as Amazon[3], Rackspace[31], Azure[24], HP[18] and others [6, 21] we have found different characteristics for which cloud clients are charged. Clients can be charged depending on the region where the datacenter is allocated and on the usage of resources such as: storage, load balancers, monitors, data transfer, virtual machines, databases or cache. Figure 4.1 shows an extensible metamodel that allows the representation of these characteristics.

Since this thesis concentrates on the costs of a cloud deployment cut to the usage of VMs, I have only extended the representation of Computing resources costs for which clients are billed.

As it is evident in Figure 4.1, only the Computing element has been specialized while the other six billing categories have not. The refinement consists of the specification of VM's characteristics and the different manners in which the utilization of a VM with certain characteristics are charged.

Regarding *VMcharacteristics*, the model allow representing that cloud providers offer different types of VMs and a set of Operating System (OS) to run in each VM type. Attribute *typeOfVM* holds the model identification for the VM and *operativeSystem* holds the OS type.

Regarding the *MachineCost*, it defines the catalog of choices offered by the cloud provider for paying for a VM with certain characteristics. It consists of four attributes that allow modeling the VMs costs both when running the VM on-demand and when the VM has been reserved. I found that many providers allowed to reserve VMs, meaning that a client must pay an upfront fee but in return it has a lower running cost for the VM. This means that running a VM becomes cheaper if it is made an upfront payment that reserves it for a long period. Finally, given the moment in time when a concrete

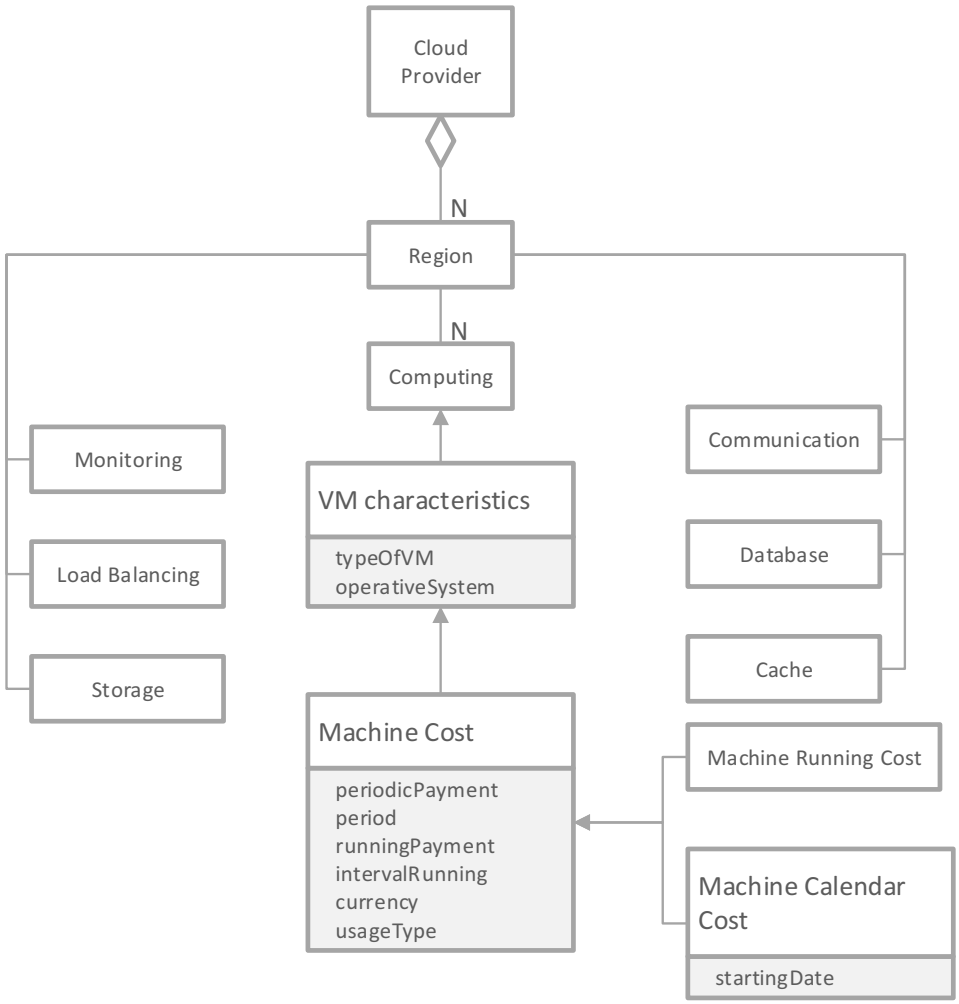


Figure 4.1: Model of a Virtual Machine

VM_x is activated (called T^x_{init}) and the moment when a VM_x is deactivated (called T^x_{end}), the assessment of time that a VM has been running is not unique among cloud providers. Attribute *periodicPayment* holds the amount of money the client has to pay upfront when reserving a VM - in case of an on-demand VM this attribute value is zero. *period* holds the time period for which the reservation is stipulated; usual values are 1 year or 3 years. *runningPayment* is the attribute that contains the amount of money requested by the provider to run the VM for an interval of time equal to *intervalRunning* - e.g., 0.24$ per hour. *currency* identifies the currency used in *periodicPayment* and *runningPayment* and finally *usageType* is the attribute that specifies if the VM is on-demand or is reserved. If it is reserved, it also specifies what type of reservation is.

Classes *MachineRunningCost* and *MachineCalendarCost* allow modelling the differences found in the assessment. An object of MachineRunningCost means that the number of intervals that the machine has been running is $\lceil \frac{T^x_{end} - T^x_{init}}{intervalRunning} \rceil$. This is, for example, the case of Amazon billing policy. An object of *MachineCalendarCost*, instead,

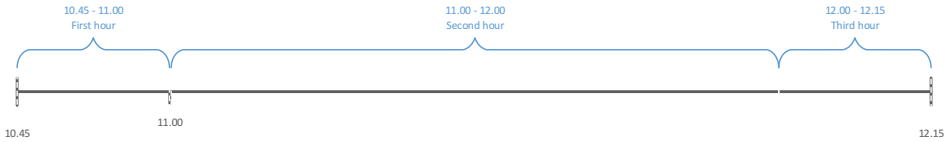


Figure 4.2: Azure timeline example

means that the number of intervals is

$$1 + \sum_{\forall n \in \mathbb{N} \,|\, T^x_{init} < startingDate + n \cdot intervalRunning < T^x_{end}} 1$$

being *startingDate* the moment in time where the cloud provider started charging its clients. An example can clarify the formalism used. Figure 4.2 depicts the situation: a VM has been started at $T_{init} = 10:45$ and ended at $T_{end} = 12:15$ of the same day. *intervalRunning* is 1 hour and *startingDate* is 1/11/2013. The summation result is 2 and represents the second and third hour, as the condition is respected for $n = 11$ and $n = 12$: in the first case $10:45 < 11:00 < 12:15$ and in the second case $10:45 < 12:00 < 12:15$. Other n values does not respect the condition. The first hour is counted by the 1 outside the summation symbol, arriving at a total of three billing hours. This is the case for Azure[24] billing policy, that bills a VM for a full hour even if it is deployed 1 minute before the next hour.

This model takes account of the most common virtual machine setting and configurations possibilities, but can be expanded and modified in case of changes in the way providers offer their products.

4.2.1 Amazon

Figure 4.3 depicts a partial instance of the metamodel that models the cost of two types of instances *small* and *HighCPU-extraLarge* types of VM running Linux in U.S. East region of Amazon AWS (whose billing model follows the *MachineRunningCost* manner in intervals of one hour). These VMs are offered as a reserved VM for three years with different types of upfront payments (e.g., for the case *small* instance, its reservation options are: an upfront payment of $215 and $0.017 more for each hour that is actually running, or an upfront payment of $257 and $0.012 for each hour that it is running) or as purely on-demand VMs (e.g., in the *small* instance type, there is not any upfront payment but it costs $0.06 each hour that it is running).

A peculiarity of this provider is that Amazon's "heavy" reservation VMs have a particular billing characteristic. They are billed for every hour in the month, whether

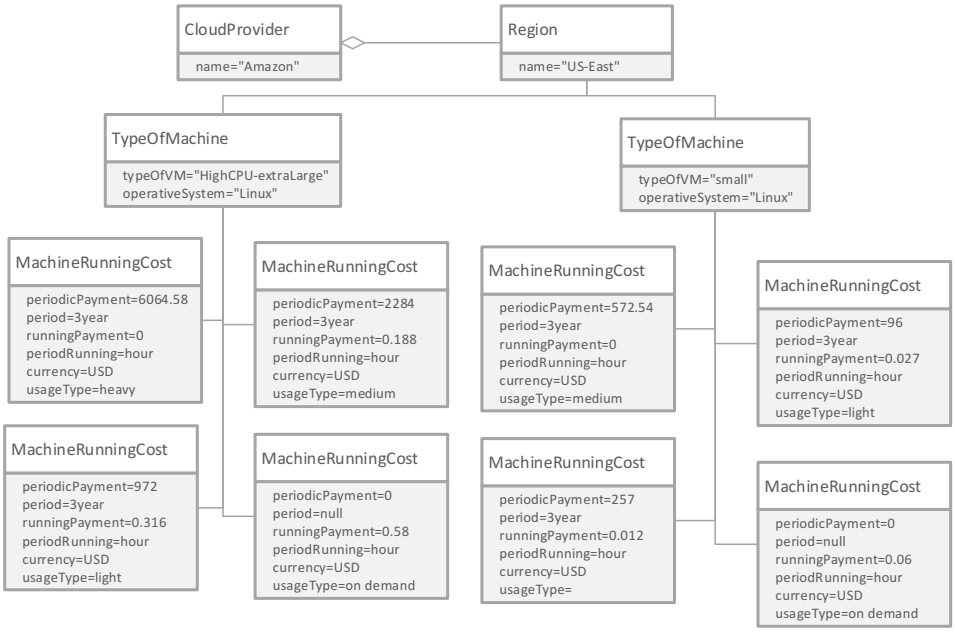


Figure 4.3: Example of Amazon billing model for small instance running Linux

they are used or not, while "light" and "medium" VMs are billed only for the hours they are powered on. This characteristic is only found in “heavy” reservation, and given the amount of people discussing about it on the Internet, it was not explained in a clear manner. In fact, if a client only reads the tables and does not read the particular details for the heavy reservation type it could be tricked into thinking that the three reservation types have the same rules. This problem has been solved by calculating in advance the cost of the heavy reserved VM during the reference period as $periodicPayment + runningPayment \cdot hoursInPeriod$, becoming this the new $periodicPayment$ and setting to zero $runningPayment$, as evident in Figure 4.3.

Example A “Medium” Amazon VM costs 277$ upfront and 0.042$ per hour at the US East (North Virginia) data center for a reservation term of 1 year. The actual cost is obtained multiplying 0.042$ with 8766, that are the number of hours in 1 year, and then adding 277$ to the result: $277\$ + 0.042\frac{\$}{h} \cdot 8766h = 645.172\$$. To reserve this VM for 1 year, whether if it’s used or not, costs 645.172$, so this is the new $upfrontPayment$ while $runningPayment$ is set to zero.

4.2.2 Rackspace Example

Figure 4.4 displays a partial instance of the metamodel that models the cost of the same *1GB* instance type of VM, one running Windows and the other running Linux of Rackspace. Rackspace does not have regions, so the only region used is named “Unique”.

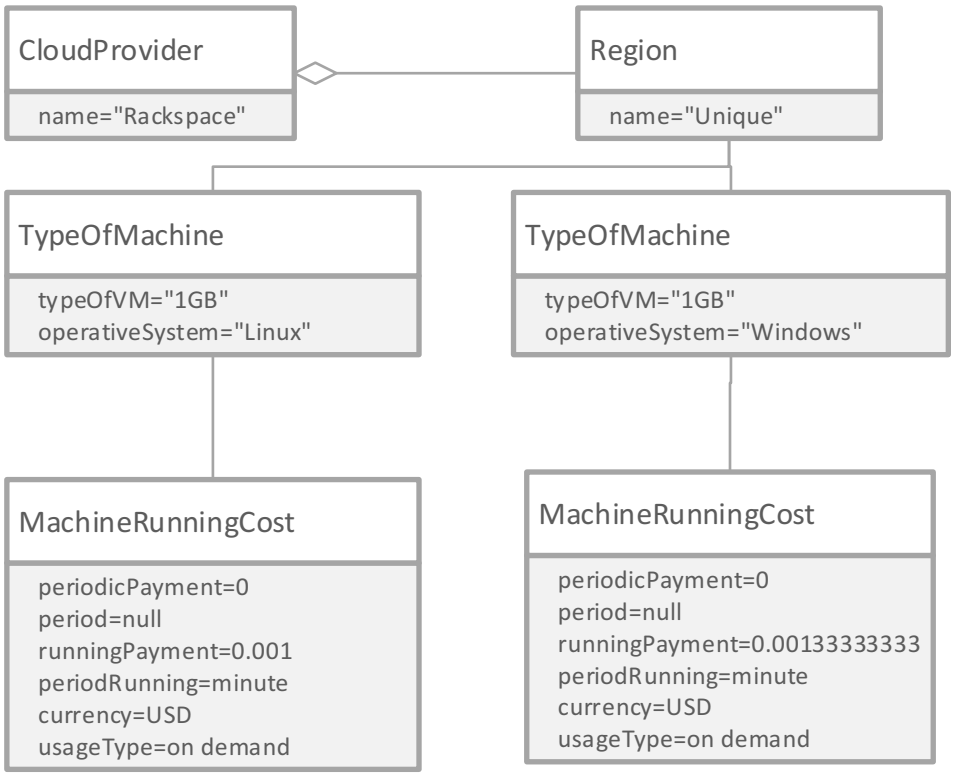


Figure 4.4: Example of Rackspace billing for the same instance on Windos and Linux OS

Moreover, Rackspace offers only on-demand VMs, as the two in the figure. It is small, but noticeable, the increase in hourly cost of the Windows' VM with respect to the Linux one, increase due to the licensing costs.

The prices are so low because are referred to one minute of VM deployment and are obtained simply by taking the hourly price and dividing it by 60. This is made possible by the fact that Rackspace's minimum billing period is one minute, as this paper [11] explains in Section 6.1.

4.3 Implementation

4.3.1 Requirements

Implementing this piece of software, apparently straightforward, posed indeed a series of challenges that had been resolved in the process. First of all, the program had to comply a few design requirements: it had to be modular and portable.

Modularity was the most important requirement, as the program needed to be easy to maintain and expand in the event of a change in the underlying model caused

by providers updates. Portability had to be enforced due to allow the widest possible audience to use this software.

Purpose of this software is to automatically retrieve information about VMs pricing from online provider, to process them and to output the cost model in a homogeneous and consistent format. This software should be operated whenever a cost model is required, if it is not already present. Its results are valid until a provider decides to changes his prices or add VMs to his offer. This event happens randomly but with low frequency, in the order of a few times per year, so maintaining the results weekly seems a reasonable compromise. On the other hand, it is also possible to run this program every time the cost model is needed, but it is most of the time a waste of resources.

4.3.2 Getting the data

The challenge posed by this program was to find a way to fetch information about virtual machines in an autonomous way from the providers' sites. Usually this information is stored in tables or is written in natural language in one or more pages of the provider's website.

Next sections explain how this problem has been addressed with three different providers: Rackspace, HP and Amazon. These three providers have been chosen because they cover a large share of the market, but other providers can be added to the program at any time, as the design of the program is modular and extendible.

The choice to hard-code the first two provider prices and to automatically parse the ones from Amazon derives from a simple cost-benefits comparison: Rackspace and HP have a manageable number of VMs and so the time spent trying to generate an automatic online price parser greatly exceeds the time saved by manually compiling the XML file. Amazon, instead, has a large VMs offer that is hard to manage and update manually, so in that case the time spent building an automatic price retrieving program is well spent.

4.3.2.1 Rackspace

Rackspace is an American based cloud provider since 1998, having in its portfolio public cloud, private cloud and dedicated bare metal computing. Its public cloud prices are available at Rackspace Pricing and at the current date it's composed by 7 different VMs for Linux-based OS and 6 different VMs for Windows-based OS. The pricing information is presented in a table, where the VMs are identified by their RAM quantity, from a

minimum of 512MB to a maximum of 30GB. The only indication of VM's CPU power is the number of virtual cores, ranging from 1 to 8, but it is not specified what a virtual core is equivalent to.

Being the prices stable for more than a year and giving the fact that the number of VMs is low, creating an automatic parser of the price tables was an greater effort with respect to a manual inspection, so these prices have been hard-coded in the program. If in the future they are subject to change, they can be modified directly in the output XML file of the program.

4.3.2.2 HP Cloud

HP is one of the most known companies in the computer field, with thousands of employees worldwide and hundreds of products, but only in the last years they opened to the market of cloud computing. At HP Cloud Pricing is shown their offer, that consist of 6 VMs with either Linux or Windows OS. They range from a "extra small" instance equivalent to 1 HP Cloud Computing Unit to the "double extra large" instance of 32 HP Cloud Computing Unit. One Cloud Computing Unit is a unit of CPU capacity that describes the amount of compute power that a virtual core has available to it. 6.5 CCUs are roughly equivalent to the minimum power of one logical core (a hardware hyper-thread) of an Intel® 2012 Xeon® 2.60 GHz CPU.

The same consideration made for Rackspace is done here: there are so few VMs that create an automatic parser is an overkill. Prices have been hard-coded in the program.

4.3.2.3 Amazon

Amazon is the principal online vendor and offers to its client more than 500 different types of VM billing. It has a very detailed costs page at Amazon EC2, where it is possible to view the VMs grouped by reservation type, that is on-demand, Light, Medium and Heavy. In each panel there is a list of VMs and it is possible to change the region from where the VMs prices are shown. Finally, in the panel there is also the option to choose between different OS. Information about the VMs performance are at Amazon Instance Types. In particular, CPU performance are measured in ECU, where "one EC2 Compute Unit provides the equivalent CPU capacity of a 1.0-1.2 GHz 2007 Opteron or 2007 Xeon processor".

There are also "Spot instances" that are VMs that are temporarily unused and that Amazon sells on a auction, where users bid to get the VM. This thesis' approach is expected to be used by clients who want to deploy a service or application over the internet, to satisfy third-users desires. This can lead to significant price discounts, but it is totally unreliable as the VM won can be reclaimed at any second from another user, so they have not been considered. This option looked as an attractive option to sustain a best-effort task of heavy computation on a cheap position, but not for deploying a stable internet service (at least at this grade of computing maturity).

Noting that the number of VMs is too high to be manageable by a human, an automatic parser is recommended. Analyzing the webpage source it came out that the prices are not embedded in the HTML but they are asynchronously fetched with Javascript from a series of JSON files. It comes natural to exploit this fact, using directly the JSON files to retrieve the prices of the VMs. The software that creates a cost model from the information of this process is available at [27]. Chapter 6 explains its usage.

4.3.3 Design

The program has been developed in Java, to ensure portability and efficiency. Besides, it's one of the most common programming languages nowadays and I was already familiar with it. The program has only a command line interface, no graphical UI is necessary at this point. A GUI can be easily created in a second time.

A class diagram is shown in Figure 4.5. There is a *CloudProvider* class that acts as a bean, containing all the information about the provider's VMs and their pricing. With the *toXML()* method this knowledge is serialized in a XML file, ready to be written into a file. The XML format has been chosen because it's easy to understand for both humans and computers and it's highly customizable and expandable.

CloudParser is an abstract class that holds a method interface that each provider class must implement *getPrices()* and two common function, *writeToFile()* and *getDataFromURL()*. A abstract class is a type of class that cannot be instantiated and contains an incomplete implementation, as in this case. For each provider there will be a class implementing this abstract class that will retrieve and parse the prices and return them in a CloudProvider object. *getPrices()* is the method that will return a CloudProvider object containing the billing information of the provider parsed by the specific class. *writeToFile()* is a utility method that writes a string to a file, while *getDataFromURL()* reads the contents of a web page and returns them in a string.

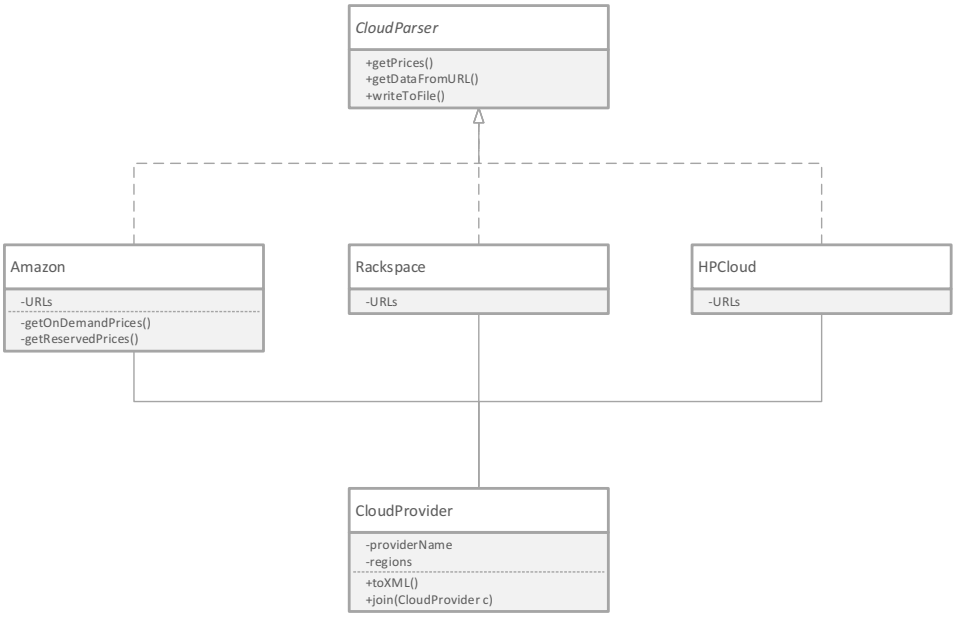


Figure 4.5: Cost model class design

The metamodel explained in Section 4.2 had to be converted into a class model. Figure 4.6 shows the classes that has been created. Starting from the top, there is a *CloudProvider* class containing the name of the cloud provider and a list of *Region* objects. Each *Region* holds the region name and a list of *TypeOfMachines* objects. Each *TypeOfMachines* is identified by its *machineCode* and holds a list of *OperatingSystem* objects. Each *OperatingSystem* contains the name of the OS and a list of *MachineRunningCost* objects. Finally, a *MachineRunningCost* contains the following attributes:

- *periodicPayment*: also known as upfront payment, is the amount payed only once in the specified *period* to run a single instance of the VM, in the *currency* specified.
- *runningPayment*: amount payed to run a single instance of the VM in a *period running* time interval, in the *currency* specified.
- *currency*: currency used in the pricing. Can be USD or EUR.
- *period*: period of time where the virtual machine is reserved.
- *periodRunning*: minimum amount of time in which a virtual machine can be billed.
- *usageType*: type of virtual machine reservation that some providers, like Amazon, offer can be one of the following states: on_demand, light, medium, heavy.

The only method common to each class is *toXML()* that has the task to serialize the information contained inside the class and output it in a XML string. *purge()* is used to

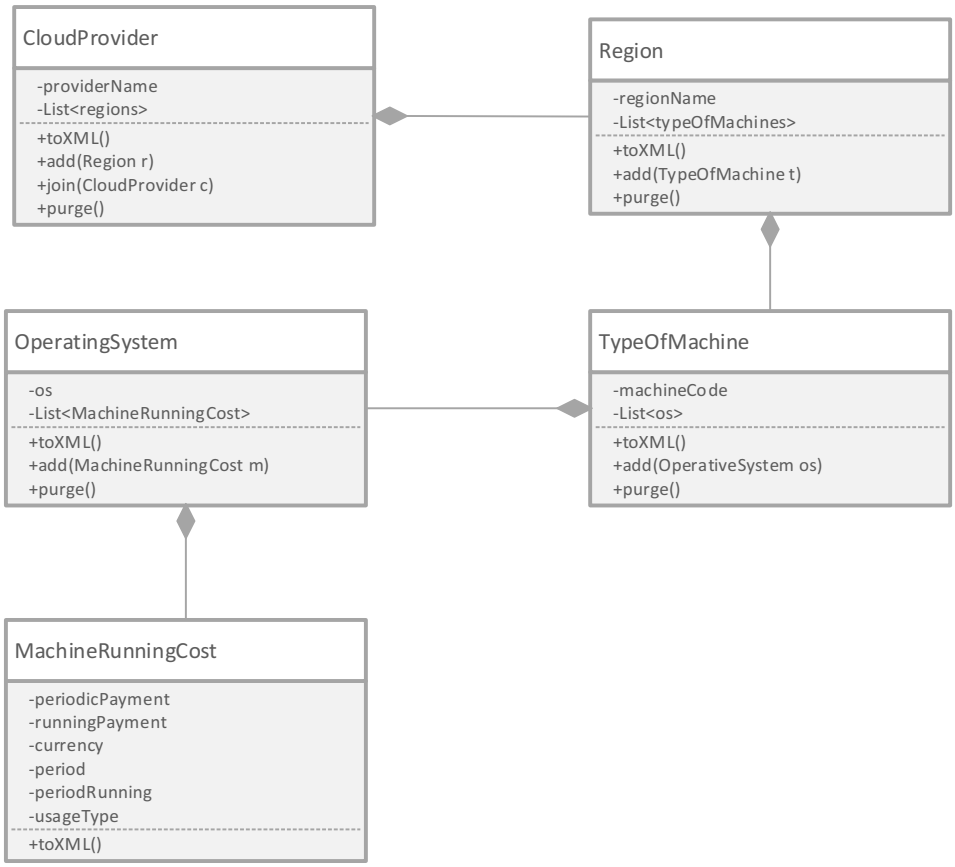


Figure 4.6: Cloud Provider class model

check the presence of empty or useless objects, such as a Region with no TypeOfMachine objects or a MachineRunningCost instance with both periodicPayment and runningPayment set to zero, and to remove them. Each *add()* method adds the passed parameter to its list of objects, checking for duplicates recursively. In case of a duplicate, the duplicate is not added. The *join()* method in CloudProvider class has the same behavior, but first checks that the joining CloudProvider object has the same *providerName* property.

4.3.3.1 XML

Having chosen XML as output file it was necessary to define the equivalent XML representation of each object in Figure 4.6.

It has been decided to create a single tag for each class, while each classes' properties are written as attributes. This is a very compact and efficient way of writing information, with low text redundancy. Another possibility was to create for each attribute the corresponding tag, but it would have been difficult for a human to read the file as the information is dispersed in many ramifications.
Figure 4.7 shows an extract of the Amazon cost model XML output by the final program.

```
<cloudProvider name="Amazon">
    <region name="us-east-1">
        <typeOfMachines name="m1-small">
            <os name="linux">
                <machine currency="USD" runningPayment="0.06" periodicPayment="0.0" period="null" periodRunning="hour" reservationType="not_set"/>
                <machine currency="USD" runningPayment="0.034" periodicPayment="61.0" period="1 year" periodRunning="hour" reservationType="light"/>
                <machine currency="USD" runningPayment="0.027" periodicPayment="96.0" period="3 years" periodRunning="hour" reservationType="light"/>
                <machine currency="USD" runningPayment="0.021" periodicPayment="139.0" period="1 year" periodRunning="hour" reservationType="medium"
                />
                <machine currency="USD" runningPayment="0.017" periodicPayment="215.0" period="3 years" periodRunning="hour" reservationType=
                "medium"/>
                <machine currency="USD" runningPayment="0.0" periodicPayment="291.71" period="1 year" periodRunning="hour" reservationType="heavy"/>
                <machine currency="USD" runningPayment="0.0" periodicPayment="572.54" period="3 years" periodRunning="hour" reservationType="heavy"/>
            </os>
            <os name="windows">
                <machine currency="USD" runningPayment="0.091" periodicPayment="0.0" period="null" periodRunning="hour" reservationType="not_set"/>
                <machine currency="USD" runningPayment="0.059" periodicPayment="69.0" period="1 year" periodRunning="hour" reservationType="light"/>
                <machine currency="USD" runningPayment="0.051" periodicPayment="106.3" period="3 years" periodRunning="hour" reservationType="light"
                />
                <machine currency="USD" runningPayment="0.044" periodicPayment="160.0" period="1 year" periodRunning="hour" reservationType="medium"
                />
                <machine currency="USD" runningPayment="0.039" periodicPayment="250.0" period="3 years" periodRunning="hour" reservationType=
                "medium"/>
                <machine currency="USD" runningPayment="0.0" periodicPayment="510.54" period="1 year" periodRunning="hour" reservationType="heavy"/>
                <machine currency="USD" runningPayment="0.0" periodicPayment="1167.735" period="3 years" periodRunning="hour" reservationType=
                "heavy"/>
            </os>
        </typeOfMachines>
        <typeOfMachines name="m1-medium">
            <os name="linux">
                <machine currency="USD" runningPayment="0.12" periodicPayment="0.0" period="null" periodRunning="hour" reservationType="not_set"/>
                <machine currency="USD" runningPayment="0.068" periodicPayment="122.0" period="1 year" periodRunning="hour" reservationType="light"/>
                <machine currency="USD" runningPayment="0.054" periodicPayment="192.0" period="3 years" periodRunning="hour" reservationType="light"
                />
                <machine currency="USD" runningPayment="0.042" periodicPayment="277.0" period="1 year" periodRunning="hour" reservationType="medium"
```

Figure 4.7: Amazon cost model in a XML file

4.3.3.2 Amazon JSON

Amazon publishes its prices on webpages as tables that load the data dynamically. After some investigation, it has been found that tables get their data from asynchronously fetched JSON files which contains all the pricing information. Identified the JSON files location, they were exploited to retrieve Amazon's prices automatically.

The structure of a Amazon JSON prices file, regarding prices of Linux on-demand VMs, is shown in Figure 4.8.

Being the JSON objects formatted in a slightly different model with respect to the one presented in Figure 4.6, there is the need to traduce from objects in this notation to the objects shown in Figure 4.6.

Being this not a formal API it could be subject to sudden changes in case Amazon decides to alter its prices webpage. In that case it will be sufficient to update the traducer code to reflect the changes made. This happened only once in one year and the change made was not in the file structure but in the content of an attribute, so the traducer code has been quickly updated.

```
{
  "vers":0.01,
  "config":{
    "rate":"perhr",
    "valueColumns":[
      "linux"
    ],
    "currencies":[
      "USD"
    ],
    "regions":[
      {
        "region":"us-east",
        "instanceTypes":[
          {
            "type":"stdODI",
            "sizes":[
              {
                "size":"m1.small",
                "valueColumns":[
                  {
                    "name":"linux",
                    "prices":{
                      "USD":"0.060"
                    }
                  }
                ]
              },
              {
                "size":"m1.medium",
                "valueColumns":[
                  {
                    "name":"linux",
                    "prices":{
                      "USD":"0.120"
```


Figure 4.8: Part of a JSON file containing Amazon Linux on-demand VMs prices

Among the utilities that are used for parsing JSON files, I decided to use the Google JSON Java Library[16] for the program, because it is a lightweight standalone library with no external dependencies, excellent performance and easy to use. With just one line of code it reads and parses a JSON files and returns a Java object populated with the JSON file's data. Other parsers, like JSONLib[12] and FlexJSON[19] were discarded due to the excessive tuning required to properly work with the Amazon JSON files.

4.3.4 Implementation

The program has been implemented in Java, following the design directives. The most challenging part has been the parsing of Amazon's JSON file, because it wasn't documented as the JSON files are not parts of a public API. The programming has been done with the Eclipse editor.

4.4 Running program

The program can be invoked by this command

```
java -jar cloudParser.jar [Amazon | Rackspace | HP] [destination
    file xml]
```

It is possible to specify two arguments. The first one is the name of the provider that will have its prices parsed. The second is the name of the destination XML file, where the prices will be written.

It is also possible run the program without arguments; in this case, the program will interactively ask for a provider and a destination file.

Program's running time computing Amazon prices is, on average, 5057ms; to write the Rackspace cost xml file the running time is 237ms. The reason to this discrepancy is in the fact that Amazon prices need to download 8 different JSON files, while Rackspace have its prices hard-coded. These running times are perfectly reasonable for the running frequency this program has been designed to.

Chapter 5

Plan synthesis

"I'm feeling lucky"

Google

This chapter explains the process of the adaptation plan synthesis. It details concepts as what the prerequisites are, what the process consists of, the rationale behind the separation between reserved and on-demand processes, what "buckets" are.

Once we have a cost-model of a cloud provider that can be easily managed automatically, it can be synthesized an adaptation plan that takes carefully into account the cost of VMs. Such adaptation plan will guide the allocation and de-allocation of VMs, exploiting the elasticity offered by the cloud. The plan is divided in two processes: the first one deals with the most predictive type of trace and uses only reserved VMs to generate a plan; the second part of the process, instead, works on the workload trace remaining from the first process, containing mainly bursty accesses, and uses exclusively on-demand VMs. Each process generates an adaptation plan where it is specified the number and type of VMs to activate depending on the input access rate.

5.1 Overview

The plan synthesis process has the job of calculating the number and, if possible, the reservation type of VMs needed to run a service, given its expected workload and SLA to meet.

Figure 5.1 details this stage, for reasons that will be explained subsequently: the

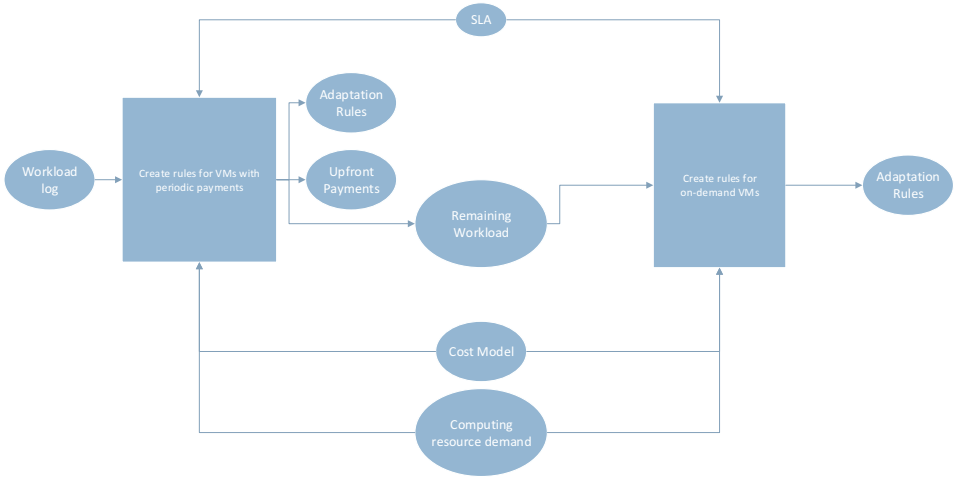


Figure 5.1: Two-stage plan synthesis

process has been separated into two parts, one only for on-demand VMs and one for reserved VMs. Inputs of this stage are the workload log, SLA, the cost model and the computing resource demand. The outputs of the first part are a set of adaptation rules, the upfront payments and a remaining workload. In turn, outputs of the second part are only the set of adaptation rules for the on-demand stage.

A plan is a set of rules that automatically indicate when to deploy or remove VMs according to the incoming access rate of the service.

5.1.1 Separation between on-demand and reserved plan synthesis

As it was exemplified in the cost model in Section 4.2, some providers offer the option to book VMs in order to obtain a less hourly cost. Reserved instances must be booked ahead and constitute a finite set of available VMs, while on-demand VMs can be theoretically infinite and so there is no limit in deploying them. This is a core difference: reserved VMs are a finite number as their number must be known ahead, while on-demand VMs are not planned and at any moment of time a large number of them can be powered on.

Due to this distinction two distinct procedures and the different kind of workload that are expected to support have been developed to best exploit the potentiality of each instance type.

5.2 Inputs and outputs

The second stage of the process, the *plan synthesis*, has four different inputs and generates two or one output, depending which module has been used. These inputs have already been shown in Section 3.2, so will be briefly refreshed.

5.2.1 Cost Model

The cost model is the output of the program discussed in Chapter 4. It consists of a XML file containing the prices of all the relevant VMs of a provider, in the format shown in Figure 4.6.

5.2.2 SLA

The SLA follows the specification detailed in Section 3.2.1 and specifies the service performance, together with the "reward" obtained for each step of the SLA. This SLA identifies and regulates the performance that the service provider has to guarantee in order to satisfy its user requirements. The Service Provider is in his turn a client of the the Cloud Provider: it makes use of cloud resources and its agreements to satisfy its users needs and maximize its benefits. Figure 1.1 exemplified the situation.

5.2.3 VM Characteristics

Information about VM performance have to be fed to the program. Since VM performance are *application dependent*, there could not be added anything generic to the cost model that allows the plan synthesis process to compare a set of VMs.

Other research groups are currently benchmarking and ranking VM performance of different cloud providers[11]. In this thesis it is assumed as known just the common information in Performance evaluation research, the service rate (μ) of each VMs used in the plan synthesis, together with the information to identify and retrieve the correct VMs from the cost model.

5.2.4 Workload log

The adaptation plan synthesis step creates a plan tailor-made for an existing software application or service that is intended to offer its services over the Internet. Information about the expected load of the service is provided through the workload log, as explained in Section 3.2. Formally, the log has been defined in equation 3.3 as $log = (I, (n_1, n_2, \ldots, n_N))$, where $I > 0$ is the length of the time interval and each $n_i, i \in [1, N]$ is the amount of requests during time interval i.

This definition is useful for mathematical and theoretical reasoning. In the implemented process, there is a log file that contains for each line the number of request n_i, while the length of the time interval I is given separately in a configuration file.

5.3 Workload analysis and its impact on resources cost

5.3.1 Burstiness

One of the common things of services accessible through Internet, like mail servers, web servers or database servers, is that they all have variable and bursty access rates. It is a well known fact in the IT world and it's even more evident if we plot the access requests of any of these servers on a plot, as did in Figure 3.2. On top of a normal day-night cycle, there is a great number of unpredictable access peaks. To deal with the peaks on a user managed server the only solution is to dimension the server capacity in order to handle the maximum number of requests, plus a margin as safeguarding; i.e., set it up to be able to cope with the worst-case scenario. This leads to server underutilization because usually the worst-case scenario rarely happens but it shows a much higher arrival rate of requests than the normal usage. This is the first reason to switch from in-house servers to cloud deployment: the possibility to own and pay only for the resource needs in each moment, rather than for the worst case scenario.

Example Figure 3.2 shows the access rates of the FIFA World Cup 1998 webserver during 2 consecutive weeks. The highest access rate registered was of $453.5\,^{\text{req}}/_{\text{s}}$ while the mean access rate was of $46.9\,^{\text{req}}/_{\text{s}}$. This behavior can be traced to the nature of the website: during a world series football match, people who had no access to television wanted to know the result of the match, probably constantly refreshing the page to get the last updates. This is the reason for the access peaks in this particular case.

Without the extreme outlier values, the server could have been dimensioned for approximately 120 $^{req}/_{s}$, but instead it had to be dimensioned for 500 $^{req}/_{s}$. Even worse, without a careful study of bursty periods, one could only use the expected mean arrival rate of requests (46.9 $^{req}/_{s}$) and naively feel that its deployment can deal with the workload of the service. It is 4 times more powerful, and only to sustain an inbound traffic that happens in a small fraction of the total time. Indeed, only the 5.3% of the time the requests are more than 120 $^{req}/_{s}$, so for the 94.7% of the time the server is underutilized. However, the requests during such period represent the 21.56% of the total.

5.3.2 Cloud Providers

An online cloud provider offers to its clients the possibility of buying various services, including but not limited to VMs, load balancers, monitors, backup & replication, dns, databases. For these services there could be engineered an adaptation plan that manages their usage in each moment in order to compute the client costs. This thesis concentrates on the VMs management, abstracting away from the rest of artifacts offered by the cloud providers.

Comparing cost web pages for VMs across different providers it is evident how some of them, especially the largest ones, offer different types of purchase. The different types of purchases have been grouped into two sets: on-demand and reserved.

On-demand is the famous “pay as you go”, the user pays only for the resources effectively used, and he can request as many resources as he can afford. There is no limit in purchasing on-demand resources and the performance delivered are the same respect a reserved instance.

For reserved instances, instead, the user will have to pay an amount upfront, with the benefit of a discounted hourly price for the VMs. Among the different levels of VM reservation that this set includes, the common rule is “the more the user pays upfront the less the VMs costs hourly”.

A partial list of online providers is present in Appendix A.

5.3.3 Cost study

Let me now show a study of the price of the same VM on different reservation prices. The VM used for this study is from Amazon, region US East, OS Linux, type Large,

reservation for 1 year and for 3 years, prices of October 2013. Prices have been extracted from the cost model generated with the cost model retrieving.

reservation	upfrontPayment	periodicPayment	period
on-demand	0	0.24	-
light	243	0.136	1 year
medium	554	0.084	1 year
heavy	1166.84	0	1 year
light	384	0.108	3 years
medium	860	0.067	3 years
heavy	2237.57	0	3 years

Table 5.1: Costs for a Amazon Linux VM, US East region, as of October 2013

In Figure 5.2 it is shown the cost graph for the VM at 1 year reservation period. The red line represents the cost for a heavy reservation VM, which is constant as these VMs are billed whether they are powered on than not, so it is parallel to the x axis.[1]

Costs are obtained by the formula

$$cost(h) = upfrontPayment + periodicPayment \cdot h$$

where h is the number of hours the VM has been running while *upfrontPayment* and *periodicPayment* are the attributes of the cost model presented in Section 4.2 and summarized in Table 5.1.

To pay the minimum cost possible, the only variable to know is how many hours the VM will be active. Once this variable has been set, it is easy to choose the best type of VM purchase just looking at the graphic and selecting the reservation associated to the lower line. In a more systematic way, it's easy to find the point where the lines intersect and use that value as a threshold for the usage of the current reservation type.

In this case, the limit value for being the on-demand VM usage the most economic is 2337 hours (26.67%). In turn, for light VM reservation is 5981 hours (68.23%) and for medium VM reservation is 7296 hours (83.23%). Remember that the number of hours in 1 year is 8766. This information can be condensed in system 5.1.

[1] Pre-reviewers of this document noticed that the line seems inclined downwards. It is merely an optic illusion.

$$\begin{cases} on\,demand & h < 2337 \\ light & 2337 \leq h < 5981 \\ medium & 5981 \leq h < 7296 \\ heavy & 7296 \leq h < 8766 \end{cases} \tag{5.1}$$

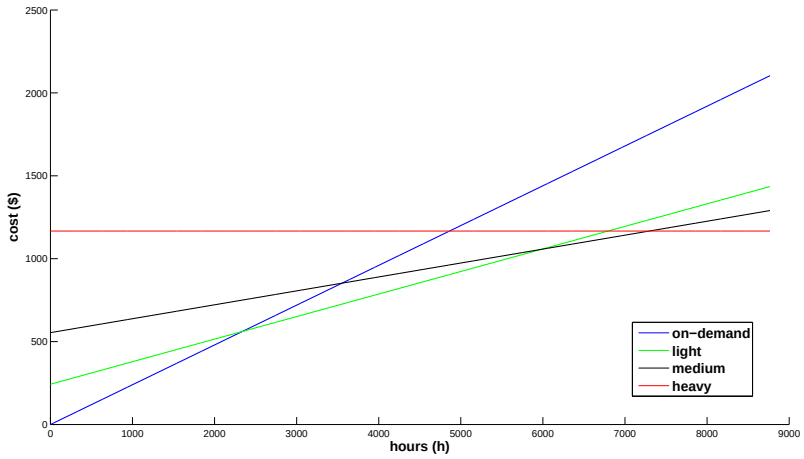


Figure 5.2: Cost of the same Amazon VM at different reservations - 1 year period

Figure 5.3 shows the cost graph for the VM at 3 years reservation period, or equivalently 26298 hours. On-demand VM is convenient until 2910 hours of usage (11.07%), light reservation VM is convenient until 11610 hours (44.15%) and medium VM is convenient until 20561 hours (78.18%). This information is summarized in the system 5.2.

$$\begin{cases} on\,demand & h < 2910 \\ light & 2910 \leq h < 11610 \\ medium & 11610 \leq h < 20561 \\ heavy & 20561 \leq h < 26298 \end{cases} \tag{5.2}$$

It is evident how on-demand VM should be considered only for seldom usage, while the other reservation types can help containing costs. This statement is in agreement with the burstiness of Internet services workloads; the bursty periods rarely happen but, when they do, they require a large number of resources for a short period.

From Figure 5.3 it is clear how a naive approach that uses only on-demand VMs costs spends in about a year the amount needed to run the same VM at a different

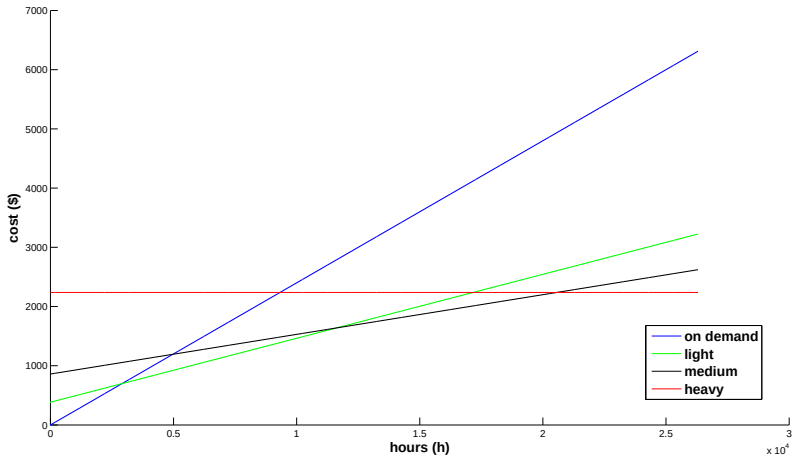


Figure 5.3: Cost of the same Amazon VM at different reservation types, all of them considering the 3 years period

reservation for 3 years.

This analysis is fundamental to understand what type of reservation choose, knowing the number of hours the VM will be deployed. It is clear also the usefulness of the study performed on this thesis, since usually a service provider that deploys its service in the cloud can hardly calculate without help from an expert the proportions of time that VMs will be active.

5.4 Plan synthesis for Reserved VMs

This stage creates a subset of rules of the adaptation plan, those rules that will be in charge of activating and deactivating VMs that are reserved. Since the cost of VMs reservation should be payed before running the service, the number of VMs to reserve and the type of reservation of each of these VMs should be known before running the services. Therefore, in this stage it has also been implemented a method that returns such information, i.e. the type of VMs to reserve and how many VMs of each type of reservation available in the cost model, as shown in Figure 5.1. A flowchart that describes this process in a more detailed manner is provided in Figure 5.4.

Before being used, the workload log *log* has to be normalized. The *log* provided by the user is, as explained in Section 5.2.4, a series of N numbers where each value represents the amount of requests received in I seconds. The plan synthesis works with units in seconds, so the *log* trace is normalized by dividing each value by I. Having stated previously that $I > 0$ this division doesn't pose any problems. After this preliminary

operation, the plan synthesis can start.

This stage works on the most stable part of the workload, as the bursts will be analyzed and managed with on-demand VMs. Due to this fact, it is not made use of the variety of VMs offered by the cloud providers but it has been decided to use only one of the types. Otherwise, it would make the application's performance less predictable and a slight change in the workload could entail a big difference in the computing infrastructure, which will turn into an excessive adaptation rate. It has been selected the most powerful type of VM for the type of application's computing resource demand. For example, if the application is memory intensive, it will be selected a VM with high performance for memory-intensive executions; if the application is CPU-intensive, it will be selected the type of VM with high computing capacity, and so on. So for this let's call it VM_i.

The rule synthesis starts by calculating the optimal usage thresholds u_n for each reservation type of VM_i and store them in a vector. A usage threshold $u_n \in [0, 1]$ is the limit where one type of reservations stops being convenient and another type starts being the optimal choice; i.e., the generalization of the example illustrated in Section 5.3.3. The thresholds are found by intersection of the cost lines, where a cost line is given by

$$periodicPayment + runningPayment \cdot hours \tag{5.3}$$

$periodicPayment$ and $runningPayment$ are obtained from the cost model and $hours \in [1, period]$ is the variable number of hours the VM has been used during the $period$ (for example, for Amazon a $period$ is 1 year or 3 years, equivalent to 8766 and 26298 hours, while for Azure a $period$ is 6 months or 1 year). As $hours$ is the only variable, it is easy to find the intersection between two functions as an equation between the two lines; afterward, the usage threshold is obtained by dividing the $hours$ found with the referring $period$. Saying *reservation1* has $periodicPayment_1$ and $runningPayment_1$ and *reservation2* has $periodicPayment_2$ and $runningPayment_2$, then it is found the value $hours$ that makes

$$periodicPayment_1 + runningPayment_1 \cdot hours = periodicPayment_2 + runningPayment_2 \cdot hours$$

So,

$$hours = \frac{periodicPayment_2 - periodicPayment_1}{runningPayment_1 - runningPayment_2}$$

Finally, it is derived the usage proportion, independently of the time unit used:

$$u_1 = \frac{hours}{period}$$

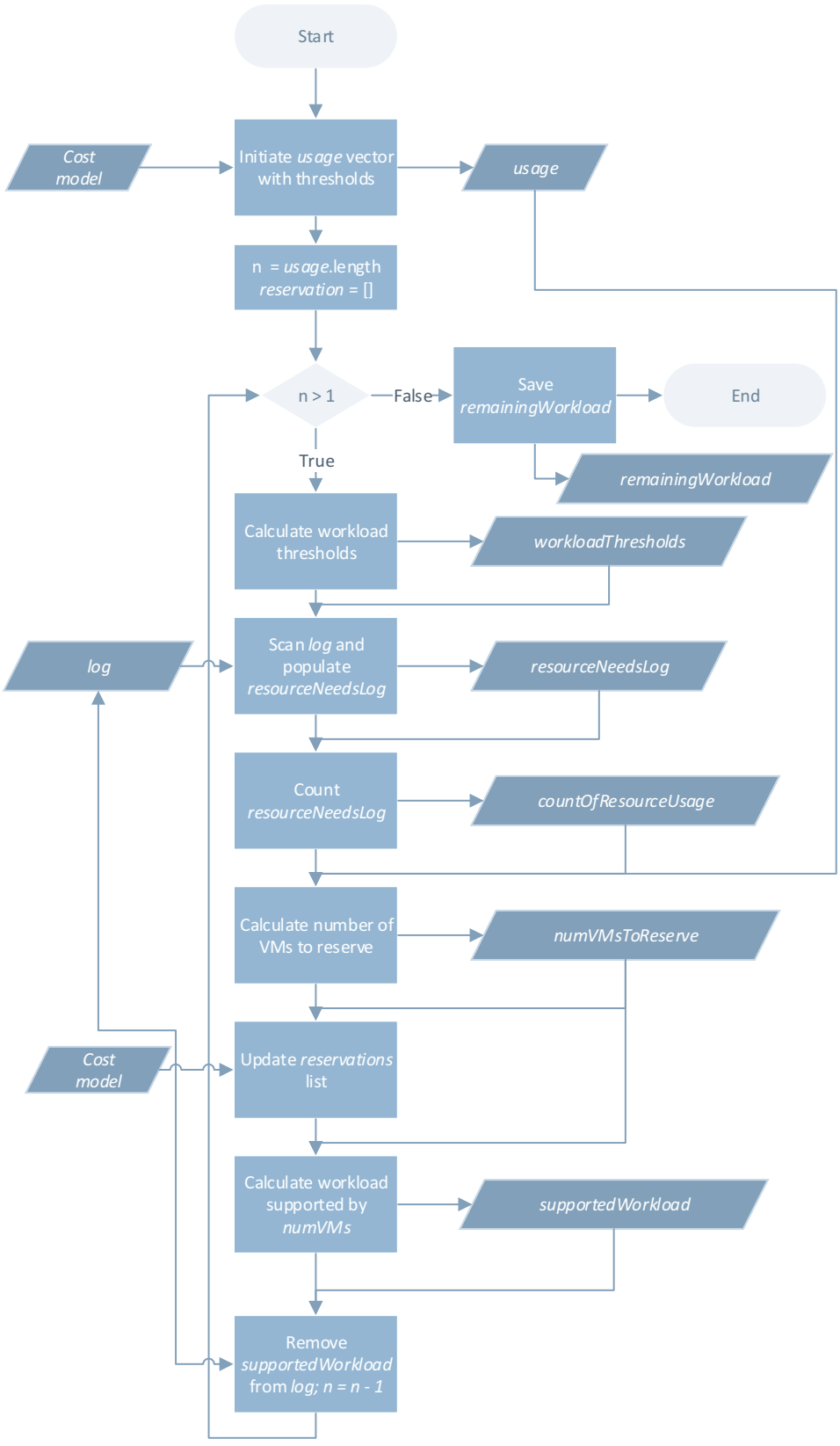


Figure 5.4: Reserved VMs plan synthesis flowchart

This information is structured in a list called *usage*. The *usage* list is composed by pairs in the format $(cost_n, u_n)$, where $cost_n$ is an object of the cost model with the cost information of the type of reservation that is the optimal up to usage u_n and u_n is the usage thresholds for that cost. In the rest of the thesis for identifying the two elements that compose the n-th list item I use the notation $usage_n.cost$ and $usage_n.u$.

Example Here it will be shown the steps to calculate the thresholds for the the first threshold is obtained by comparing on-demand and light reservation types for a Amazon VM, region US East, OS Linux, type Large, reservation for 3 years. The first threshold is found by comparing on-demand and light reservation costs:

$$0 + 0.24 \cdot h = 384 + 0.108 \cdot h \Rightarrow h_1 = \frac{384 - 0}{0.24 - 0.108} = 2909$$

Having $period = 3\,years = 26298\,hours$,

$$u_1 = \frac{h_1}{26298} = 0.1106$$

Comparing light and medium reservation,

$$h_2 = \frac{860 - 384}{0.108 - 0.067} = 11610$$

that becomes

$$u_2 = \frac{11609}{26298} = 0.4415$$

Finally,

$$h_3 = \frac{2237,57 - 860}{0.067 - 0} = 20561$$

and

$$u_3 = \frac{20561}{26298} = 0.7818$$

So, the usage threshold is

$$usage = ((cost_{on-demand}, 0.1106);\ (cost_{light}, 0.4415),\ (cost_{medium}, 0.7818),\ (cost_{heavy}, 1))$$

Curiosity

The formula used to calculate usage thresholds has the benefit to always give back a result, unless Figure 5.2 showed two lines in parallel, in which case the denominator of the fraction becomes zero. In this case it is easy to see that a kind of VM purchase is always better than another, and this situation could be easily

recognizable by the software tool. Interestingly, it can also happen that the number of hours returned is negative, meaning again that one cost line is always better than the other. This fact indeed happened during the tool implementation. The tool I implemented founded 3 cases in the Amazon AWS[2] public prices where prices were inconsistent, I reported my findings to Amazon, they recognized the inconsistency and they corrected their worldwide public prices[28].

Once the *usage* list is initialized, it is initiated a iterative process that unveils the amount of VMs to contract with each type of cost, as shown in Figure 5.4 and Algorithm B.1.

It is created an empty array, called *reservation*, that will hold the calculated number of VMs for each type of reservation. Each position in the array will store the pair $(cost_n, numVMs_n)$. Therefore, the array will have the form

$$reservation = [(cost_1, numVMs_1),\ (cost_2, numVMs_2), ...]$$

being $cost_n$ an object of the cost model with the cost information of a VM for a type of reservation and $numVMs_n$ a natural value with the number of VMs to reserve at a cost $cost_n$. The size of *reservation* array is N (i.e., the same of *usage*) the idea is to cycle through each *usage* element and allocating to each payment type a number of VMs. The number of VMs for each payment type depends on the frequency with which they will be allocated. The process starts considering the amount of machines that are expected to be active the highest proportion of the time. Thus, the process starts considering the information in the last position of *usage* (i.e. those ones with the highest upfront payment).

5.4.1 Calculate the workload thresholds

Into the loop of Figure 5.4, the first operation is to *calculate the workload thresholds* (in terms of arrival rate of requests) λ_j for which, taking into account the *sla*, the profit of the system becomes the same either using j instances of VM_i or $j+1$ instances. This thresholds are stored in the *workloadThresholds* array, which is an output of the activity (as depicted in Figure 5.4). The thresholds mean that, ideally, the best moment to have the j+1-th VM activated is just when the arrival rate of requests to the application exceeds λ_j. Figure 5.5 shows the $profit$ function with *sla* , service rate of requests μ constant, and $numVM$ variable. The parameters used are $sla = ((1, 0.01), (2, 0), (\infty, -0.01))$, $\mu = 5\ ^{req}/_{s}$

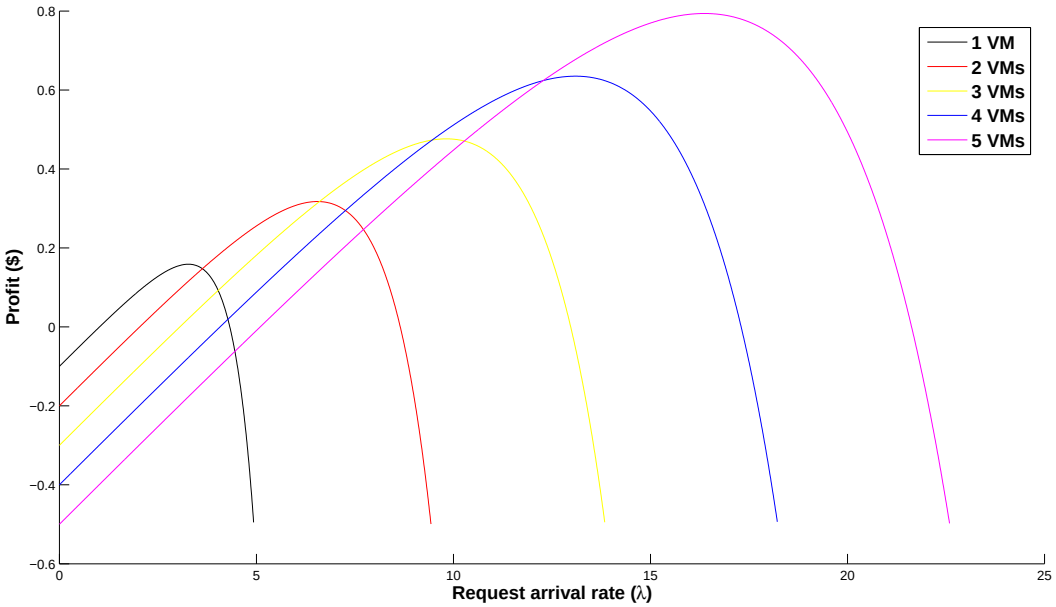


Figure 5.5: Profit in function of the workload and amount of active VMs

and $costVM = 0.1$, while $numVM \in [1, 5]$. The intersection between the first curve and the second is located at $\lambda_1 = 3.63\,^{\text{req}}/_{\text{s}}$.

***profit* function** Curves in Figure 5.5 were obtained by using the well-known queueing theory that states that the probability that the response time τ of a request is less than r is:

$$P(\tau < r) = 1 - e^{(\lambda - \mu)r} \tag{5.4}$$

when each VM holds its own queue of requests and assuming than requests to the application are randomly forwarded to one of the VMs[3]. The arrival rate of requests is assumed to follow the exponential distribution, with an arrival rate of λ, which in this case is the workload of the *log* trace. μ, instead, is the *service rate*, the number of requests handled per second by the VM (assuming again that the time to serve a request follows the exponential distribution); its value can be obtained from the *demand* array of Section 3.2.3, and the relation is $\mu_i = e_i^{-1}$

Under this assumptions, the function *profit* is defined as

$$profit(\lambda, \mu, cVM, numVM, SLA) = revenue(\lambda, \mu, numVM, SLA) - numVM \cdot cVM \tag{5.5}$$

[3]The assumption of random workload forwarding allows to have in the queue of each VM an exponentially distributed inter-arrival time of requests

where cVM is the cost of having powered on a VM, $numVM$ is the number of VM used and SLA holds the SLA information in the format explained in Section 3.2.1. *revenue* is the function that calculates the revenue of the system with the parameters specified, and is defined as

$$revenue(\lambda, \mu, numVM, SLA) = \lambda \cdot \sum_{\forall (r_i, m_i) \in SLA} (m_i \cdot (e^{(\frac{\lambda}{numVM} - \mu) r_{i-1}} - e^{(\frac{\lambda}{numVM} - \mu) r_i})) \quad (5.6)$$

This function uses queuing theory to consider the probability of finishing the execution f a request in each of the intervals considered in the *sla*.

As said earlier, each threshold λ_n is found at the intersection of two consecutive *profit* functions, or else when

$$profit(\lambda, \mu, cVM, n, SLA) = profit(\lambda, \mu, cVM, n+1, SLA)$$

5.4.2 Scan *log* and populate *resourceNeedsLog*

Once even the $workloadThresholds$ array is populated, it is time to use it to scan the *log* resource trace in order to find, for each point of the *log* trace, what is the optimal number of VMs that should be used for dealing with that amount of traffic. This operation is executed by activity *Scan log and populate resourceNeedsLog* in Figure 5.4 and its results are stored in $resourceNeedsLog$, which is an array of size N, the same as the workload log. This is accomplished by scanning $workloadThresholds$ and finding the index j such as $workloadThresholds_{j-1} < log_l \leq workloadThresholds_j$, where log_l is the current trace element. Formally, $resourceNeedsLog$ is populated with Equation 5.7

$$\forall l \in [1, N] \, \forall j \in [1, W] \\ resourceNeeds_l = j \,|\, workloadThresholds_{j-1} < log_l \leq workloadThresholds_j \quad (5.7)$$

having $N = log.length$ and $W = workloadThresholds.length$. For example, using the *log* of Figure 3.2 and $workloadThresholds$ calculated as in the example shown just above, the first iteration of the loop gives as result the $resourceNeeds$ array shown in Figure 5.6.

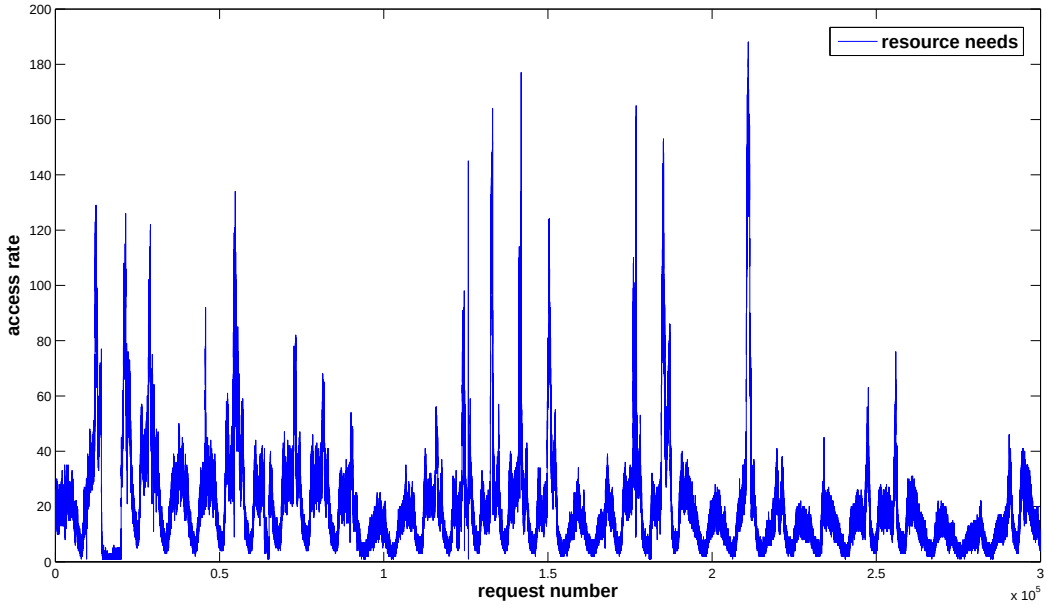


Figure 5.6: Number of required resources that maximize the profit in function of the variable workload for the first iteration of the algorithm

5.4.3 Count resourceNeedsLog

The third operation inside the loop, called "*Count resourceNeedsLog*", is intended to study $resourceNeeds$ array. In particular, the function *count* is used upon $resourceNeeds$ in order to know the proportion of time each number of VMs is active. The result is a new array, called $countOfResourceUsage$ and it is defined in Formula

$$\forall i \in [1..P] \, \forall l \in [1..N] \, countOfResourceUsage_i = \frac{count(log_l \geq i)}{N}$$

P is the maximum value stored in $resourceNeedsLog$, called $P = resourceNeeds.max$; $N = resourceNeedsLog.length$ and the function *count* simply returns the number of element of the $resourceNeeds$ array that have values higher or equal i. The result of this operation can be seen in Figure 5.7. Frequency equal to 1 means that there are always needed at least this amount of VMs, frequency 0 means that there is never required that amount of resource. Naturally, at position 1 the frequency is 1 because there is always at least 1 VM active, but the curve drops rapidly and at around 28 VMs the frequency is around 20% - it means that more than 28 VMs are needed for less than 20% of the time. The fact that in Figure 5.7 a large number of VMs are used for a very small fraction of the time means that seldom is required a high number of VMs, confirming that the *log* trace has a bursty nature.

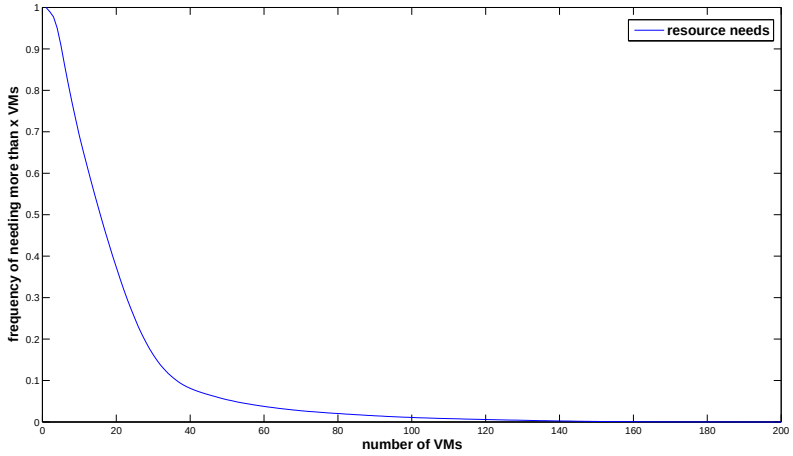


Figure 5.7: Distribution of the VM requirements

5.4.4 Calculate number of VMs to reserve

Next, it is executed the fourth operation inside the loop of Figure 5.4 called "*Calculate number of VMs to reserve*", which decides the number of VMs to reserve at the cost considered in the current iteration of the loop, i.e. at $usage_n.cost$. To do so, it is used the freshly calculated $countOfResourceNeeds$ array and the $usage$ array. The aim of this operation is to decide what is the best amount of VMs to reserve at a cost $usage_n.cost$. The number of VMs to reserve, called $numVMs$, is the maximum index i of $countOfResourceNeeds$ where $countOfResourceUsage_i$ is greater than the usage thresholds $usage_{n-1}.u$. In this manner each reservation type can reserve a number of VMs that will be deployed in their most cost effective part of the cost model.

Example For the first round of the loop, $n = 4$ and so $usage_{n-1}.u = 0.7818$. The maximum index i of $countOfResourceNeeds$ that holds the inequality $countOfResourceNeeds_i > 0.7818$ is $numVMs = 7$, with a frequency equal to $countOfResourceNeeds_7 = 0.8156$; in fact, $countOfResourceNeeds_8 = 0.7716$.
This means that 7 VMs are required (at least) 81.56% of the time. Being the current type of reservation convenient if and only if the VMs are used more than 78.18% of the time (that is $usage_{n-1}.u \cdot 100$), these 7 VMs are correctly allocated with this type of reservation.

5.4.5 Update *reservations* list

The next step, the fifth block inside the loop called "*Update reservations list*", simply adds the calculated number of VMs and the current cost object in the *reservation* list:

$$reservation \leftarrow add(reservation, (cost_n, numVMs))$$

At this point the objective of the process iteration has been achieved, to decide all number of VMs to reserve at a cost $usage_n.cost$. Now it is prepared the used data of the iterative process for the next iteration.

5.4.6 Calculate workload supported by *numVMs*

First, the amount of requests that are expected to be served by the calculated $numVMs$ VMs should not be considered anymore. This is the work of the sixth block, called "*Calculate workload supported by numVMs*". It calculates the amount of workload that will be supported by $numVMs$ VMs and store it in $supportedWorkload$. At first sight, it could seem that it is simply $workloadThresholds_{numVMs}$, that is the maximum amount of workload that is optimally handled by $numVMs$ VMs, but it would be incorrect. In fact, $workloadThresholds$ array is calculated when all the curves sharing the same cost model. Here, instead, the first $numVMs$ VMs will have the same cost model $cost_n.cost$ and the next VM deployed, $numVMs + 1$, will have the cost model $cost_{n-1}.cost$ because will be reserved at the next cycle with a different cost model. So, the workload effectively supported by the new reserved VMs is calculated by a new function also based on Queuing Networks theory called $calculateEquivalentProfit$. The result of this function is stored in $supportedWorkload$ variable.

The mentioned function *calculateEquivalentProfit* finds the workload value for which the profit is equivalent either using $numVMs$ at cost $usage_n.cost$ or using $numVMs$ VMs at cost $usage_n.cost$ and one VM at cost $usage_{n-1}.cost$. The new threshold should be marginally higher than the old, because the $costVM$ is higher for the VM $numVMs + 1$ and accordingly the gain curve starts at a lower y-axis point.

***calculateEquivalentProfit* function** The purpose of this function is to find the maximum amount of workload that $numVMs$ VMs can sustain maintaining maximum the profit, knowing that the $numVMs + 1$ VM will have a different cost model. To accomplish this goal, it is not possible to use directly the $profit$ function passing as pa-

rameters $numVMs$ and $numVMs + 1$ otherwise the result will be the same obtained in $workloadThresholds$. The first curve is obtained from the function

$$profit(\lambda, \mu, cVM,\ numVMs, SLA)$$

with $\lambda \in [0, numVMs \cdot \mu_n]$, while for the second curve has been used a slightly modified version of the $profit$ function. In fact, the $numVMs + 1$-th VM has the same μ value of the previous VMs as all the VMs equals in terms of performance, but differs for its cost. So, the $profitEq$ function becomes

$$profitEq(\lambda, \mu, cVM1, cVM2, numVMs + 1, SLA) = \\ revenue(\lambda, \mu, numVMs, SLA) - (numVMs) \cdot cVM1 - cVM2$$

In this function, $numVMs$ is the number of VMs used at the current cost $cVM1$ and only the last one is at cost of the next reservation $cVM2$. The revenue generated by all the VMs is reduced by the cost of the $numVMs - 1$ VMs of the same cost model and by the cost of the single VM with a different cost model.
The intersection between $profit(\lambda, \mu, cVM, numVMs, SLA)$ and
$profitEq(\lambda, \mu, cVM, cVM2, numVMs + 1, SLA)$ is the $supportedWorkload$.

Example Let's say that after the first cycle $numVMs = 7$, where the VMs are from Amazon, type "medium", heavy reservation with $\mu = 5\ ^{req}/_{s}$. Thresholds have already been calculated and $workloadThresholds[7] = 6.01$. $costVM$ for this VM is 0 as it is a VM at heavy reservation, so the periodic payment is zero and therefore so is $costVM$. For the next type of reservation, medium, $costVM = 0.00034$. The $calculateEquivalentProfit$ function needs to find the intersection between one curve that represents the profit obtained using $numVMs$ "large" VMs at reservation heavy and $numVMs$ "large" VMs at reservation heavy plus one "large" at reservation medium. Being the medium reservation VM at a higher cost, its curve is slightly worse - and so lower - with respect to a curve with all heavy reservation VMs. For this reason, the $workloadSustained = 18.23$ obtained by $calculateEquivalentProfit$ is marginally higher: 18.23 respect to 16.47 of $workloadThresholds_7$. Although it has been observed that the difference is not huge, it has theoretically been known that this is the precise method to calculate the thresholds and so it has been implemented to test the differences between the precise method and the inaccurate one that would directly use the $workloadThresholds_{numVMs}$ value.

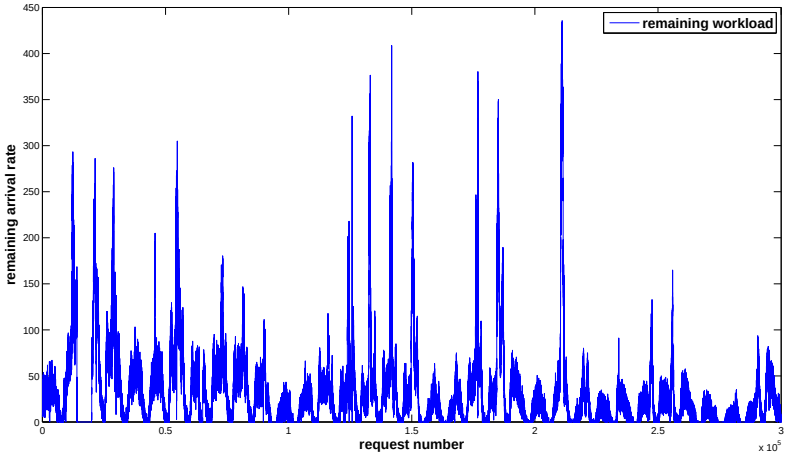


Figure 5.8: Remaining workload after the first cycle of the Reserved plan synthesis

5.4.7 Remove *supportedWorkload* from *log*

Now that it is calculated the equivalent workload supported by $numVMs$ VMs and these VMs have been added to the *reservation* array, it is safe to remove this amount of workload to the *log* trace. This operation is done in the action called "*Remove supportedWorkload from log; n = n - 1*" of Figure 5.4. This is accomplished by subtracting from each element of the *log* trace the value $supportedWorkload$. If the result is negative, the element value is set to zero. Formally,

$$log = max(log - supportedWorkload, 0)$$

Figure 5.8 depicts the modified *log* trace after the first cycle of the loop. It would seem that the plot is the same as Figure 3.2, but it is not. In fact, each element has been scaled down by the value $workloadSustained = 18.23$

After the *log* trace has been updated, n is decremented by one and the loop restarts until $n > 1$.

When $n = 1$ the process exits from the loop and ends. *log* becomes $remainingWorkload$, as it contains the workload that has not been addressed by the reserved VMs and will be the input for on-demand plan synthesis. It can be viewed in Figure 5.9. The *reservation* list can be returned to the user as it is completed.

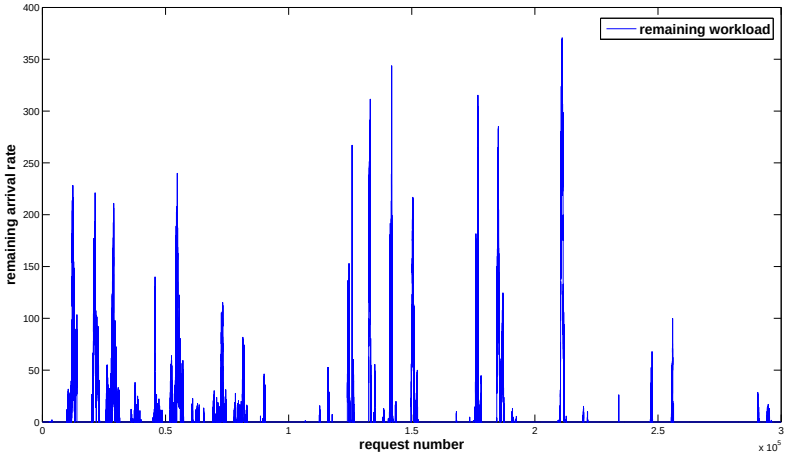


Figure 5.9: Workload remaining after the complete Reserved plan synthesis

5.5 On-demand Plan synthesis

At this point, it has been calculated the number of VMs to reserve, the type of reservation of each of those VMs and the arrival rate of requests thresholds where each VM should be ideally activated or deactivated. These calculated VMs are expected to deal with an important part of the workload, for example 90% of the time (as it was unveiled in Section 5.3.3) these reserved VMs can deal with the received workload. There are however some parts of the expected workload that should be managed by VMs activated purely on-demand; these are the behaviors of the workload that rarely happen but require a much larger quantity of resources than the usual workload.

This is the second stage depicted in Figure 5.1 and it is dedicated to calculate the part of the plan that guides the plan synthesis using only on-demand VMs. Input of this stage is the *remaining workload* of the previous stage (Reserved Plan synthesis), *SLA*, the *cost model*, VMs' *booting time* and the *VM characteristics*. Practically, they are the same inputs of the previous stage, except for the addition of VMs' *booting time* and the fact that the *log* arrives from the previous stage stage. In case the provider does not offer reserved instances it is possible to use only this stage directly with the workload log coming from the application. In any case, the ideal *log* input is *workloadRemaining* because this stage is designed to deal with bursty loads. Having as natural input the *workloadRemaining* generated by the previous process, the input *log* does not require any normalization. This means that, differently from the reserved plan synthesis where each value is divided by the interval length I, the *log* trace in this process is untouched and goes straightforward to the analysis process.

This process is expected to create the adaptation rules that will deal with the

sudden increments and decrements in the workload (i.e, the *bursts* of requests). The speed with which these increments take place makes unfeasible the creation of adaptation rules that modify the computing infrastructure by only one unit, as it was done in the previous process. Instead, adaptation rules that activate or deactivate buckets of VMs are created. On one hand, the concurrent activation of a bucket of VMs will create a temporal over-provisioning of resources. On the other hand, the activation one by one of VMs would cause a temporal under-provisioning of resources.

The activation of VMs one by one during bursty periods and the consequent under-provisioning has much more harmful consequences for the service than the activation of VMs by buckets and the resulting over-provisioning. There are two main reasons that support the fact that under-provisioning is more harmful than over-provisioning:

- the over-cost due to over-provisioning of resources is linear to the amount of the VMs over-provisioned and the amount of time they are active. Regarding the amount of time they are active, it is known that it will be for short times since, due to the manner in which the rules in the previous section have been created, none of the current VMs is expected to be active more than $usage_1.u$ proportion of time. Regarding the other term, i.e., the amount of machines over-provisioned, in the following paragraphs adaptation rules are created taking into account this issue; so, having under control the number of VMs over-provisioned

- the over-cost due to under-provisioning of resources is very harmful in this case because it is function of the application's performance loss and number of requests received. In case of under-provision of VMs, when the workload becomes bursty the application may collapse, so requests will show poor performance during that time and therefore losses from the SLA will boost. Moreover, even if the length of the interval of time of the spike of request is short, the number of requests received during this interval is much higher than the average (even an order of magnitude higher), as shown in Subsection 5.3.1.

As a consequence, to propose a safe adaptation plan, adaptation rules have been created that consider some over-provisioning of VMs by means of activating buckets of VMs at the same time, so it is avoided remaining for long time periods under the optimal computing capacity when the arrival rate increases fast. Figure 5.10 compares the two approaches. It depicts a burst of accesses: in less than 50 minutes requests rise from zero to 110 $^{req}/_{s}$. The burst is already extrapolated on top on the normally sustained workload by the reserved VMs, i.e. 0 $^{req}/_{s}$ in Figure 5.10 does not mean that the system received zero connections as they have already been handled by the deployed VMs.
On the left of Figure 5.10, to deal with the incoming burst a new on-demand VM is

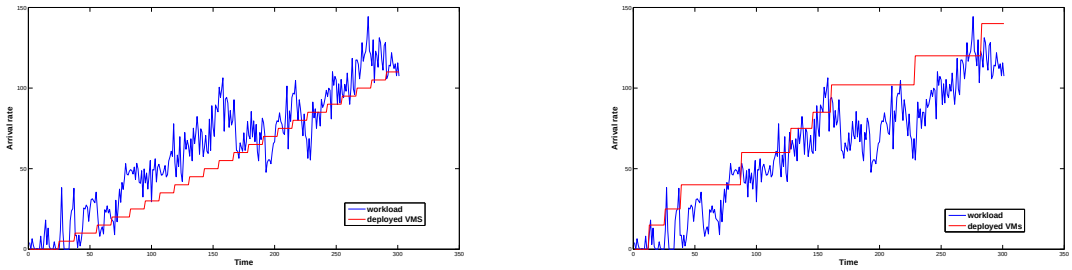


Figure 5.10: Comparison between single VM deployment and bucket deployment.

deployed each time the incoming arrival rate exceeds the number of requests that the system handles maintaining maximum profit. However, before being ready to reply at incoming connections, the new VM needs to boot and this operation takes an amount of time called *bootingTime* that in this case has been set to 60 seconds. After *bootingTime* seconds, the VM is ready to accept workload. As the burst increases rapidly the arrival rate, the deployment of one VM at a time is not able to keep up. This leads to worse user experience and lower service profit due to the higher response times.
On the right, instead of single VM, buckets of VMs are deployed when the incoming workload rises. Also in this case buckets become active *bootingTime* seconds after their deployment, but thanks to the temporary over-provisioning it is rare to find cases where the arrival rate (in blue) overcomes the amount of requests handled by the VMs (in red). The response times are kept within optimal values.

Therefore, in order to avoid remaining for long time periods under the optimal computing capacity when the arrival rate increases, it has been decided to create rules that activate a bucket of VMs concurrently, although it could case some temporal over-provisioning of resources. To calculate the bucket size two different paths have been explored: one of them is based on finding "special" arrival rate values and another is based on the difference in the workload between the moment in which it is launched a VM activation order and the moment in which the VM is effectively serving requests. Next paragraph describe these two methods.

Decision of buckets of VMs based on *resistance values* for the arrival rate

It has been studied in the *remaining workload* how the arrival rate usually increases. It was expected to find some arrival rate values that acted as *resistance* (i.e., a value of arrival rate for which the workload "bounces" off this value rather than breaking through it). In case of finding such arrival rates, an adaptation plan would be devised where each rule would activate a bucket that contained the optimal number of VMs for dealing with the arrival rate difference between two consecutive *resistance* values. For

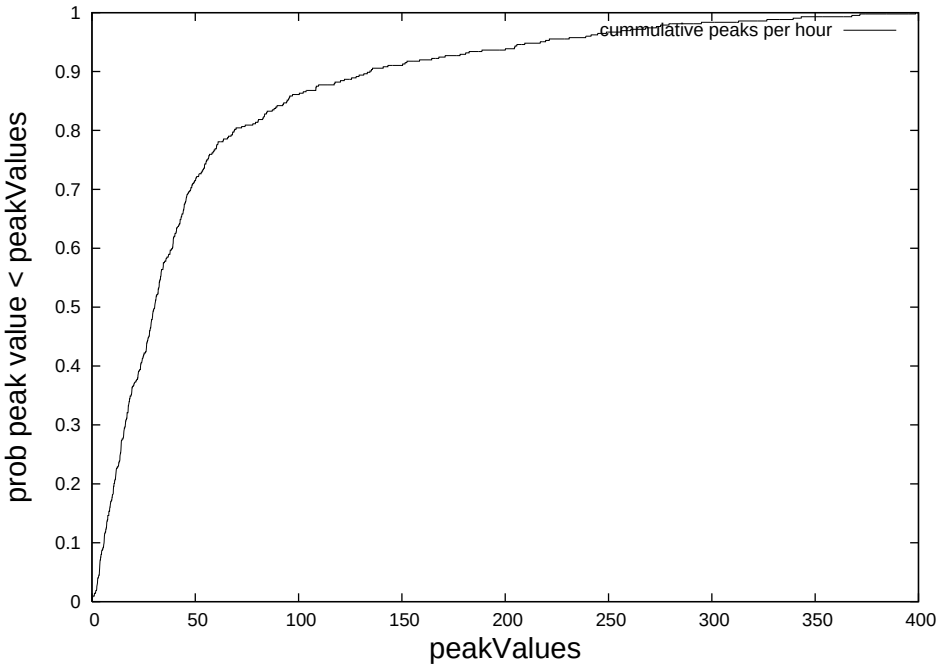


Figure 5.11: Cumulative function of peak values

doing this study *peaks* arrival rate values have been extracted from the *remaining workload* trace. To calculate the *peaks* values, it has been considered a *potential peak* if it was the maximum arrival rate value between two zero values of the *remaining workload* (most of the positions in the *remaining workload* array are zero since it stores only the peaks above the usual workload that rarely happen). After, among the *potential peaks* found, it has been kept only the maximum value during each *periodRunning* (e.g., maximum one peak for each hour in the case the provider is Amazon) as a *peak*. Using the created set of *peak* values, it has been calculated the probability of a *peak* to be lower than a value; i.e., $P(peak \leq x)$, which is analog to the cumulative function of *peak* values. Figure 5.11 shows such a plot. Looking at the figure, there is no arrival rate that acts as a *resistance* value, because otherwise the plot would show a shape like an irregular stair. Therefore, this method for creating buckets of VMs has been discarded.

Decision of buckets of VMs based on the VM *bootingTime*

This method is based on the calculation of the number of VMs that are expected to be required to maximize the client's profit when the system adaptation finishes. This is, when a VM activation order is launched, there will be activated as many VMs as they should be required when the adaptation process finishes. Therefore, the bucket size of each adaptation rule should allow the deployed computing capacity to join the optimal computing capacity once the VMs are effectively serving requests. This method entails a workload prediction for the system in the near future. The term *bootingTime* has been used to indicate the time required for VMs to adapt, which includes for example the VM

activation time and the time for application initialization. Now the interesting values are the expected workload when the system finishes its adaptation. These values have been called *workloadAfterBoot*.

The flowchart describing this stage's steps is shown in Figure 5.12 and it is described in detail in the next sections.

The on-demand plan synthesis receives a set of M VMs, the cost-model that describes the cost of the VM when its $periodicPayment = 0$, i.e. VMs are purely on-demand, and the *demand* array containing the computational power of each VM. Say, for VM_i, $\mu_i = demand_i.e^{-1}$.

Other research groups are working on characterization the performance of VMs offered in the cloud [11]. Such work is out of the scope of this thesis. Instead, it is assumed as input the service rate μ_m of the application on each type of VM, assuming that eventually those research groups offer confident results.

5.5.1 Calculate thresholds of each VM

The first operation is to calculate $workloadThresholds$ array for each VM, with the same procedure explained in the previous section.
Each array is stored in $workloadThresholds_m$, a list of arrays where $m \in [1, M]$, from the most powerful at position 1 in descending order (a VM VM_1 is more powerful than a VM VM_2 if $\mu_1 > \mu_2$). So, $workloadThresholds_1$ is an array containing the thresholds of the most powerful VM, $workloadThresholds_2$ contains the thresholds of the second most powerful VM, and so on. The list of arrays $workloadThresholds$ is depicted in Figure 5.13.

The *buckets* list that will hold the *bucket* arrays is initialized. Each *bucket* is an array of size M where in each position of the array is saved the number of VMs of a type of VM that compose the bucket. So,

$$buckets = (bucket_1, bucket_2, ..., bucket_B)$$

where each $bucket_i$, $i \in [1..B]$ is an array of size M, holding in each position the number of VMs of the respective type that compose the bucket. For example,

$$bucket_1 = [3\,2\,1\,0]$$

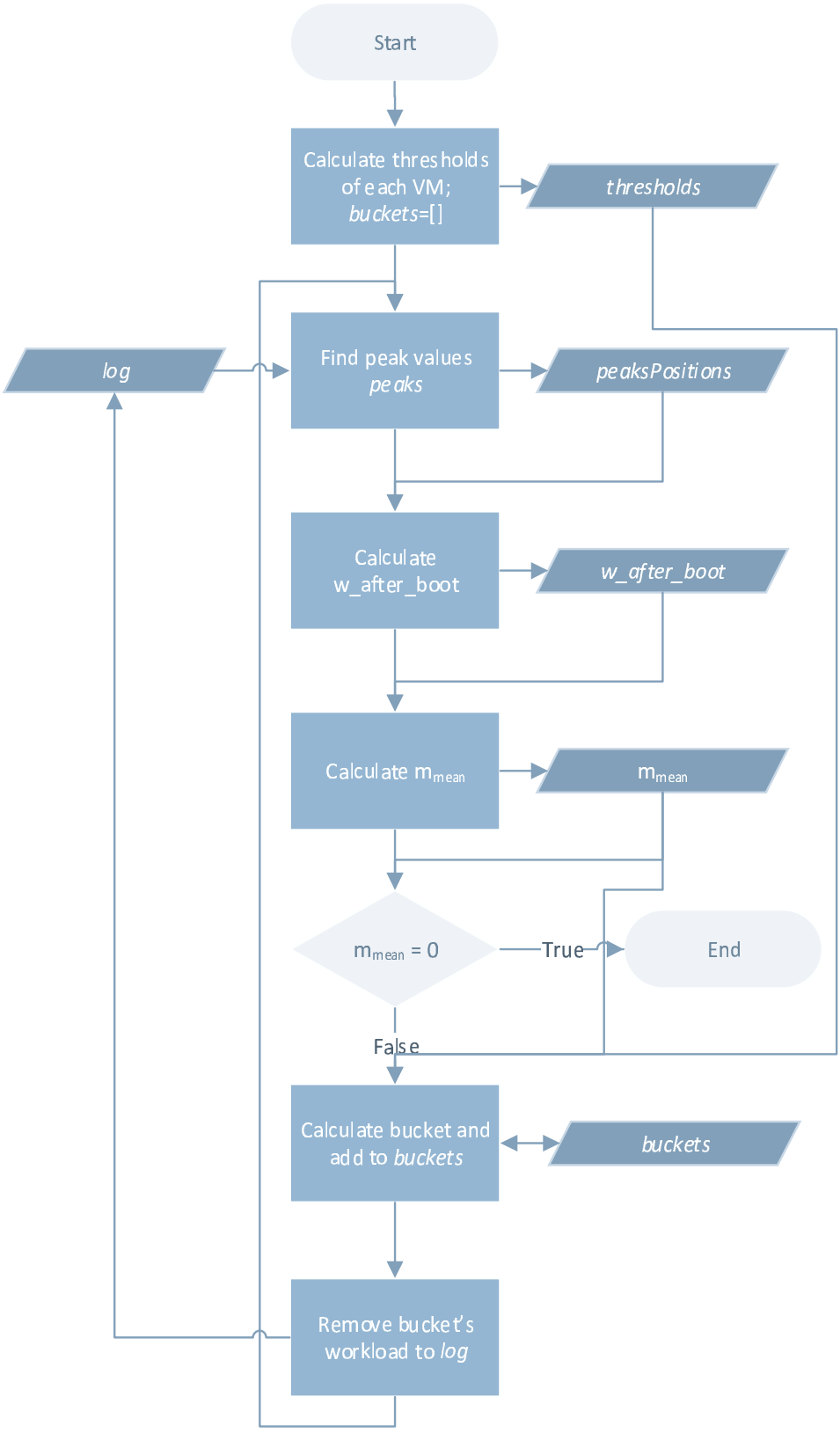


Figure 5.12: On-demand VMs plan synthesis flowchart

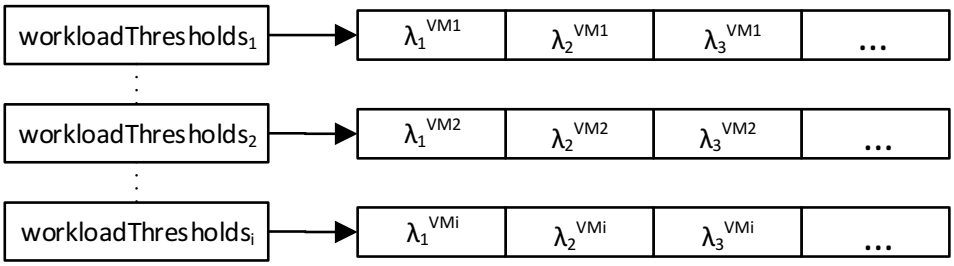


Figure 5.13: Organization of the *workloadThresholds* list of arrays

meaning that the first bucket is composed by 3 VMs of the first VM type, 2 VMs of the second VM type, 1 VM of the third VM type and 0 VM of the fourth type. Inside a bucket, there are no restrictions for the order in which the VM types are written, but once it has been chosen it must be the same for all the buckets. Although, it is recommended to sort the type of VMs from the least powerful to the most powerful. i.e., in the example written before the least powerful type of VM had 3 VMs in the bucket, while the most powerful had 0 VMs.

5.5.2 Find peak values *peaks*

The second step is to find the *peaks* values in *remainingWorkload* trace. As it was stated earlier, as definition of *peak* that has been adopted is "the maximum arrival rate value between two zero value of the *log* trace". With this definition, the smallest peak possible is a single positive value preceded and followed by a zero.

After some testing of finding *peaks* values in the *remainingWorkload* trace, it has been found that there were too many single points with very low values that brought nothing but noise in the process. Therefore I changed the definition the following: "a *peak* is the maximum arrival rate value in a window of values of the *remainingWorkload* trace different from zero".

With this new definition, all the noise points are eliminated or anyway have a lesser impact in the process. The *window* size is set to the maximum value between the *intervalRunning* and the *bootingTime*. For example, let it *bootingTime* be equal to 10 minutes. In case the provider is Amazon, *intervalRunning* is equal to 1 hour and the *window* size is consequently set to 1 hour; in case the provider is Rackspace, which has *intervalRunning* equal to 1 minute, the window size is set to 10 minutes, as the *bootingTime* period is greater than *intervalRunning*. The peaks are found by searching in the *remainingWorkload* trace, staring from the beginning, in intervals of *window* size the maximum value and storing its position in the *peaksPosition* array. If the maximum

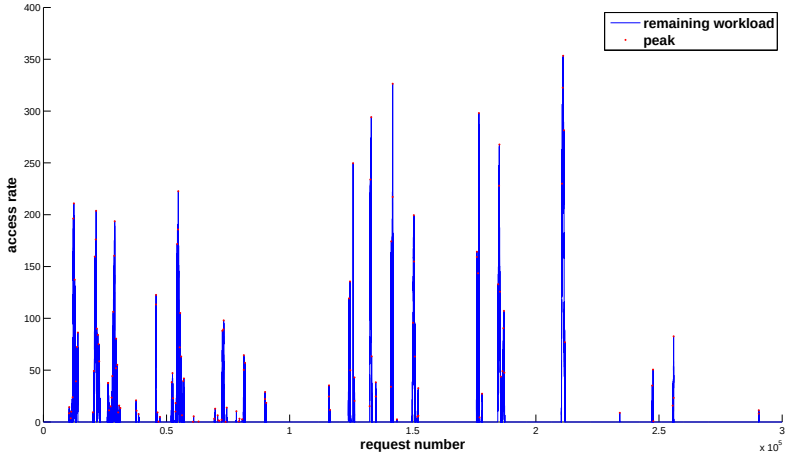


Figure 5.14: *Peaks* found in the *remainingWorkload* trace, marked in red

value of an interval is zero, the position is not added. If in an interval there are multiple peaks of the same arrival rate value, only the first one is considered. In Figure 5.14 we show the peaks found on the *remainingWorkload* trace, marked in red.

5.5.3 Calculate *workloadAfterBoot*

After finding all the peaks, it is possible to calculate the *workloadAfterBoot* array. For each peak, it is found the last zero element, i.e. the closest zero that precedes in time the peak, and from that position it is taken the value present $bootTime$ time after the moment of that zero value. The rationale is to predict the workload that will be present in the system when the activated VMs finished their booting period and are ready to serve client's requests.

Example Let's say that the first peak is at position 300 of the *log* trace and the last zero value is at position 240 (it means that all the values between 241 and 300 are positive). $bootingTime$ is $60s$ and it is known that between each element of the trace passed $5s$. It means that, to move forward of $60s$ in the trace, it is necessary to shift of $\frac{60s}{5s} = 12$ elements, arriving at element 252. The *log* value at position 252 is $workloadAfterBoot$: $workloadAfterBoot_1 = log_{252}$.

5.5.4 Calculate m_{mean}

The next step simply extract the average of the *workloadAfterBoot* array:

$$m_{mean} = \frac{1}{L} \sum_{i=1}^{L} workloadAfterBoot_i$$

where $L = workloadAfterBoot.length$.

If $m_{mean}$is zero, it means that the array *workloadAfterBoot* contains only zero values or that it is empty. In both cases, the loop breaks and the buckets are given to the user, otherwise the algorithm continues creating the bucket. Given the set of VMs to utilize, it is needed to find the combination of VMs that offer the highest profit for this arrival rate m_{mean}.

5.5.5 Calculate bucket and add to *buckets*

To find the combination of VMs that is best suited to sustain the arrival rate m_{mean}, an iterative approach is used.
First of all, an array *bucket* of size M, where M is the number of VMs and the size of *demand* array, is initialized with all the elements to zero. It will contain the number of combination of VMs that best sustain the arrival rate m_{mean}.
The list of arrays $workloadThresholds$ is involved in this procedure. It is worth remembering that in this array each threshold $\lambda_n^{VM_i}$ indicates that from $\lambda_{n-1}^{VM_i}$ $^{\text{req}}/_{\text{s}}$ and up to $\lambda_n^{VM_i}$ $^{\text{req}}/_{\text{s}}$ the ideal number of VMs to use is n, because the profit is maximum. Moreover, this list is sorted in such a way that the $workloadThresholds$ arrays are sorted from the most powerful VM to the least powerful. So, starting from $n = 1$, the most powerful VM, it is scanned $workloadThresholds_n[x]$ up until it is found the first threshold that has a greater value than m_{mean}. Analytically,

$$x \in \mathbb{N} \,|\, workloadThresholds_n[x] > m_{mean} \wedge workloadThresholds_n[x-1] < m_{mean}$$

Naturally, $x \leq workloadThresholds_n.size$. This formulation leaves out the case where $x = 1$, but this is not a limitation as this case would be discarded anyway, as explained at the end of the paragraph. If a value x is found, that means that using x VMs of the current type exceeds the input requests, as $workloadThresholds_n[x] > m_{mean}$, leading to some underutilization. So, using $x - 1$ VMs it is certain to exploit the full potential of the VMs. Consequently, $bucket_n = x - 1$, that means the current bucket, at position n, utilize $x - 1$ VMs. That is also the reason for which $x = 1$ is not an acceptable solution:

with just 1 VM m_{mean} has been exceeded, probably a smaller and cheaper VM is best suited to take care of that burst of requests.
The workload allocated to the $x-1$ VMs is removed from m_{mean} by doing

$$m_{mean} = m_{mean} - workloadThresholds_n[x-1]$$

The m_{mean} remaining is positive because it holds the inequality

$$m_{mean} > workloadThresholds_n[x-1]$$

Then, n is increased by one unit and the procedure restarts, terminating when $n = workloadThresholds.size$.

Once the VMs that compose the bucket have been calculated, it can be added to the list of buckets, after a simple check. If, at the end of the loop, the bucket is empty, that is $\forall n \in [1, M]\, bucket_n = 0$, it means that every VM was too powerful and did not sustain optimally the access rate m_{mean}. This happens only if m_{mean} is lower than the first threshold of the smallest VM, nominally $m_{mean} < \lambda_1^{VM_M}$, which is in position $workloadThresholds_M[1]$.

If the bucket is not empty it is added to the *buckets* list:

$$buckets = add(buckets, bucket)$$

5.5.6 Remove bucket's workload from *log*

The procedure continues by removing from *remainingWorkload* the workload supported by the current *bucket*. The workload supported by the bucket is obtained by summing all the $workloadThreholds_n[x]$, where $n \in [1, M]$ and x is $bucket_{M-n+1}$, i.e. the supported workload of each VM that compose the bucket. This notation is valid only if in the bucket the VMs are sorted from the least powerful to the most powerful VM, while it is worth to remember that the list of array *workloadThresholds* is sorted from the most powerful to the least powerful VM. So, if s is the supported workload of the bucket and it is initialized to zero, it is calculated by

$$\forall n \in [1, M] \quad s = s + workloadThresholds_n[bucket_{M-n+1}]$$

Example The current iteration found, for example, the bucket $bucket = [4\,3\,2\,1]$ and now it is needed to calculate the workload supported by this bucket.
First of all, it is initialized s at zero: $s = 0$. Then the procedure starts by adding the *workloadThreshold* corresponding to the first VM, the most powerful:

$$n = 1 \Rightarrow s = s + workloadThresholds_1[bucket_4] = workloadThresholds_1[1] = 15.64$$

Then, it moves to the second VM of the *workloadThreshold* array, the third of the bucket:

$$n = 2 \Rightarrow s = s + workloadThresholds_2[bucket_3] = 15.64 + workloadThresholds_2[2] = 41.56$$

At this point, it is added the third VM of the *workloadThreshold* array, the second of the bucket:

$$n = 3 \Rightarrow s = s + workloadThresholds_3[bucket_2] = 41.56 + workloadThresholds_3[2] = 66.26$$

Finally, it is added the last VM of the *workloadThreshold* array, the first of the bucket, the least powerful VM:

$$n = 4 \Rightarrow s = s + workloadThresholds_4[bucket_1] = 66.26 + workloadThresholds_4[1] = 74.94$$

So, the workload sustained by the bucket $bucket = [4\,3\,2\,1]$ is $s = 74.94$.

After calculating the supported workload, this is removed from the *remainingWorkload* trace because the newly created bucket will support it. This is done by the same function as in reserved plan synthesis:

$$remainingWorkload = max(remainingWorkload - s, 0)$$

The procedure that calculates the buckets terminates also in case the *remainingWorkload* trace has all values equal to zero, because in that case there are no more bursts to analyze.

Chapter 6

Experiment

"If you want to know what a man's like, take a good look at how he treats his inferiors, not his equals."

Sirius Black

In this chapter will be shown an experiment that executes the developed approach using a real-world scenario.

The service to model is a public web server, serving web pages for a series of important sport events.

It has been experimented the approach using Amazon AWS as cloud provider as it is one of the most important and complete among the cloud providers.

6.1 First stage

This stage is a preparatory phase where all the future inputs are collected, checked and ready to be fed to the programs.

It consists of preparing the cost model XML files that will be used in the next stages, along with the preparation of the other inputs that are needed later on: the workload trace, the SLA and the computing resource demand.

6.1.1 Cost models

The program reads the web to create automatically a cost-model, following the metamodel described in Section 4.2. The cost model contains the billing details of each VM of the provider in the format displayed in Figure 4.7 and is contained in a XML file.

The provider used for this experiment is Amazon. Their cost model has been obtained with the Cost Model Retrieving program described in Chapter 4.

To generate the cost model XML for Amazon, the command issued was

```
java -jar cloudParser.jar Amazon amazon.xml
```

The program *cloudParser.jar* can be found at [27].

Part of the *amazon.xml* file can be seen in Figure 4.3 - it has not been shown its integral version because it is 2402 lines long.

> Although the experiment concentrates on Amazon ,there could be easily obtained the other two cost models, for Rackspace and HP. They are much smaller due to the lower number of VMs offered by them with respect to Amazon.

6.1.2 Workload trace

The workload trace used to plan adaptations is the one described in Section 3.2. It is the log of the accesses to servers located in the Paris region for the 1998 FIFA World Cup. The trace represents the arrival rate during 3.000.000 seconds, each element represents the accesses registered during 10 consecutive seconds, and so the log consists of 300.000 elements. The trace is showed in Figure 3.2.

6.1.3 SLA

The SLA models the penalty/reward system explained in Section 3.2.1.

For a user writing the SLA is not trivial because it asks to decide a revenue on the basis of the service response time of each request.

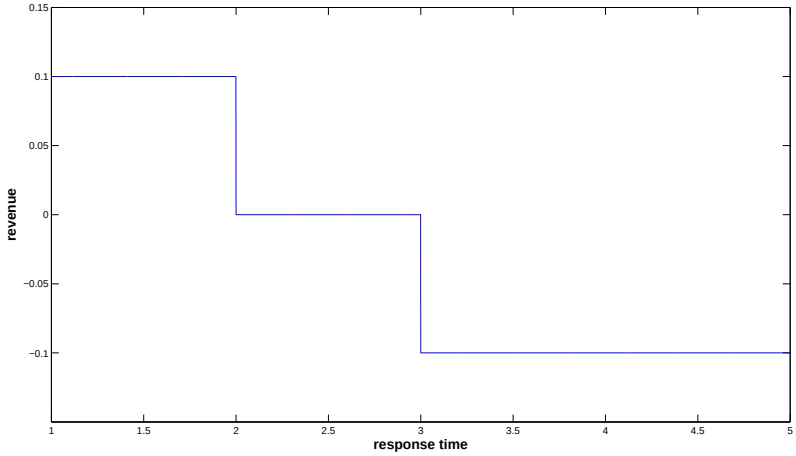


Figure 6.1: Graphical representation of the SLA

The SLA used is the example is

$$SLA \begin{cases} 0.1 & \text{if} \quad t < 1s \\ 0 & \text{if} \quad 1s \leq t < 2s \\ -0.1 & \text{if} \quad t \geq 2s \end{cases}$$

This SLA assign a revenue of 0.1 for each service response that is received by the user in less than 1 second. The profit drops to zero in case the response is received between 1 and 2 second and becomes negative if the service response time is above 2 seconds. Figure 6.1 shows the SLA.

6.1.4 VM characteristics

For the VM performance characteristics, this thesis relies on studies that compare VM performance already published on the web. Concretely, it has been used Cloud Harmony[9]. This data is publicly available and is the source of the μ value used.

Given the fact that the service object of the analysis is a web server, benchmark data about its performance on VMs offered by the provider used in this comparison is obtained from the "LAMP" benchmark. It has been selected the "LAMP" benchmark because it measures performance using Linux, Apache, MySQL, PHP, which are the core components that are typically used to run a website, although they are non the only ones. The VMs whose benchmark were present are Amazon's small, medium, large and xlarge VMs from the servers located in Ireland. Their values are displayed in Table 6.1.

As the μ values depend on the concrete application to deploy, part of the work

Provider	Type	Result	μ
Amazon	ec2-eu-west.linux.m1.small	35.37	7.07
Amazon	ec2-eu-west.linux.c1.medium	72.25	14.45
Amazon	ec2-eu-west.linux.m1.large	94.89	18.98
Amazon	ec2-eu-west.linux.c1.xlarge	101.71	20.34

Table 6.1: Benchmark data from Cloud Harmony website for the "LAMP" benchmark

that is completely out of the scope of this thesis, it has been kept the ratio shown in Table 6.1 regarding VM performance. For exemplifying the approach, it has devised an application that required 50ms to execute in the most powerful VM. So, the results in Table 6.1 have been divided by 5, then obtaining the μ values for each VM while keeping their real performance ratio.

6.2 Second stage

After completing the first stage, all the input files should be ready to be used by this stage. There is still the need to complete the configuration file for the plan synthesis program.

As Figure 5.1 showed, this stage is divided in two parts: the first one generates the rules for reserved VMs, while the second one uses only on-demand VMs to create a plan.

6.2.1 Reserved plan synthesis

The execution of the experiment for the reserved instances follows the configuration explained in the next Section. In order to be reproducible, the execution performed is explained in Section 6.2.1.2. Both intermediate and final result of this stage are described in Section 6.2.1.3.

6.2.1.1 Configuration file

The configuration file for the plan synthesis program contains all the required information to successfully generate a valid plan and needs to be filled carefully. An example of a valid configuration file is in Figure 6.2.

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
    <element name="mu" value="20.34" />
    <element name="vmName" value="c1-xlarge" />
    <element name="secondsBetweenWorkloadValues" value="10" />
    <element name="regionsToAnalyze" value="eu-west-1" />
    <element name="os" value="linux" />
    <element name="period" value="3 years" />
    <element name="verbosityLevel" value="2" />
    <element name="allowOverwrite" value="1" />
    <element name="writeToStdout" value="1" />
    <element name="costModelFileName" value="C:\Thesis\amazon.xml" />
    <element name="workloadLogFileName" value="C:\Thesis\trace.txt" />
    <element name="sla" value="[0 0; 0.1 1; 0 2; -0.1 inf]" />
    <element name="workloadLogOutputFileName" value="C:\Thesis\workloadLogOutput" />
    <element name="vmReservedOutputFileName" value="C:\Thesis\vmReserved" />
</config>
```

Figure 6.2: Example of a valid configuration XML file for a reserved only plan synthesis

Let's review the parameters of the configuration file.

- *mu*: it's the μ value, the number of requests per second that the VM can deliver. It is a single value because for a reserved plan synthesis it is used only a single VM, and this is its μ value.

- *vmName*: it's the name of the VM that will be used for the reserved plan synthesis. It should be the most powerful VM suitable for the application under study, and the name should be the same as the one used in the cost model, corresponding to the attribute *name* of the *typeOfMachines* tag.

- *secondsBetweenWorkloadValues*: this parameters tells the program how much time passed, in seconds, between two consecutive values in the workload log trace file. It is implied that each value of the log trace holds the cumulative number of consecutive requests received in this amount of seconds. Is the value called I in Section 3.2.2.

- *regionsToAnalyze*: list of regions where the VM to analyze is located. Each region must contain the VM specified in *vmName* and a plan will be generated for each region's VM independently. Each region's name must be present in one of the values of attribute *name* of the *region* tag in the cost model. Different regions are separated by a comma. If the desire is to generate a plan for each region present in the cost model, it is sufficient to set this parameter to "all".

- *os*: indicates the VM's Operative System. If set to "all", all the OS of the chosen VM will be used to generate a plan. Each OS will generate a different plan because

identifies a different VM.

- *period*: indicates the period of time for which the VMs are reserved. For Amazon provider, it can be "1 year" or "3 years". This parameters allows the program to select the right VM from the cost model, as the same VM displays different costs depending on the reservation period.

- *verbosityLevel*: with this parameter it is possible to choose the verbosity level of the application, i.e. what is the degree of detail of its operations that the program will show to the user. Level 2 is the most verbose, where the program outputs all kinds of internal information, useful only in case of debug. Level 1 is intermediate, where general information messages are displayed. Level 0 is the less verbose, where only a handful of messages are showed regarding the starting or termination of procedures. General users should keep this parameter to 0.

- *allowOverwrite*: if this parameter is set to "1" (or "true", is equivalent), the program is allowed to overwrite the output files in the case they already exist. This can happen when the plan synthesis is run multiple times having selected the same VM. In case this parameters is set to "0" (or "false"), the program won't write the output file if a file with the same name is already present.

- *writeToStdout*: if is set to "1" or "true", the application can write messages to the console. If is "0" or "false", it will not write messages to the console. This parameter is independent from *verbosityLevel*, that is even with *verbosityLevel=2* the program won't output messages to the console.

- *costModelFileName*: this parameter locates the cost model file. It can be specified an absolute path to the file or a relative one from the application's folder. The cost model is a XML file generated with the *cloudParser.jar* application, executed in Section 6.1.1.

- *workloadLogFileName*: this parameter locates the workload log file, the application's trace. This file should be a text file where each line contains the number of request received in the same interval I of time. The interval of time I is specified in the parameter *secondsBetweenWorkloadValues*.

- *sla*: this parameter defines the application's SLA. For a more detailed explanation of SLA, see Section 3.2.1. It is defined in a Matlab-like format: each line contains first the reward obtained if the response time is below R seconds, and the second parameter indicates the R seconds. The end of a line is delimited by a semicolon.

The first line should be "0 0;". The last line should contain, in place of R, "inf", to indicate that is the last step of the SLA.

- *workloadLogOutputFileName*: with this parameter the user chooses the location and the initial file name of the workload remaining after the program's elaboration. In fact, an output of plan synthesis is the workload that has not been allocated by the reserved VMs and it is written in the same format as the workload log. To distinguish between different remaining workload logs, at the end of the file name is appended, in this order, the region, the OS and the type of VM that has been used to generate the plan. An example of output file name is *"workloadLogOutput_eu-west-1_linux_c1-xlarge.txt"*

- *vmReservedOutputFileName*: this parameter specifies the location and the initial file name of the XML file that will contain the result of plan synthesis. The XML file will hold an XML tag named *single_virtual_machine* with all the information about the VM that has been used to generate the plan, a tag named *workload_log_file* indicating the full path to the workload log file used for the plan and a tag named *vm_reserved* that contains the number of VM calculated for each reservation type. At the end of the XML file name is appended, in this order, the region, the OS and the type of VM that has been used to generate the plan. An example of output file name is "*vmReserved_eu-west-1_linux_c1-xlarge.xml*".

These are the parameters for a reserved-only plan synthesis. As it will be explained in the following sections, on-demand parameters are almost identical but are present in a separated section of the XML file.

6.2.1.2 Program execution

After setting the parameters in the configuration file, called *config.xml*, it is the time to run the program. The configuration is shown in Figure 6.2.

To generate the plan for Amazon, the command issued was

```
java -jar planGenerator.jar config.xml
```

The *planGenerator.jar* and *config.xml* files are in the same folder. The whole process takes around 2 seconds. The output of the program is two files, *workloadLogOutput_eu-west-1_linux_c1-xlarge.txt* and *vmReserved_eu-west-1_linux_c1-xlarge.xml*

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
<workload_log_file filename="C:\trace.txt" />
<single_virtual_machine>
    <name>m1-large</name>
    <mu>15.0</mu>
    <region>us-east-1</region>
    <provider>Amazon</provider>
    <reservation_period>_3year</reservation_period>
    <os>linux</os>
    <payments>
        <vm_payment periodic_payment="0.0" running_payment="0.24" reservation_type="not_set" />
        <vm_payment periodic_payment="384.0" running_payment="0.108" reservation_type="light" />
        <vm_payment periodic_payment="860.0" running_payment="0.067" reservation_type="medium" />
        <vm_payment periodic_payment="2237.57" running_payment="0.0" reservation_type="heavy" />
    </payments>
    <vm_thresholds>
        <vm_threshold position="0" threshold_frequency="0.11062" />
        <vm_threshold position="1" threshold_frequency="0.44147" />
        <vm_threshold position="2" threshold_frequency="0.78184" />
    </vm_thresholds>
</single_virtual_machine>
<vm_reserved>
    <reserved num="5" reservation_type="light" />
    <reserved num="3" reservation_type="medium" />
    <reserved num="3" reservation_type="heavy" />
</vm_reserved>
</root>
```

Figure 6.3: Reserved plan synthesis XML output file

A .zip file with the program and a sample configuration can be downloaded from [27].
The first file contains the *remaining workload* explained in the previous chapters, that is the amount of arrival rates from the workload log file that the plan synthesis process did not allocate to the reserved VMs and is to study with the on-demand plan synthesis. The *remaining workload* trace should be the most highly variable (i.e. *bursty*) fragment of the workload trace, containing bursts and spikes of different intensities. A plot of a workload remaining is displayed in Figure 5.9. The workload remaining is saved in the folder specified in the configuration file with the name *workloadLogOutput_eu-west-1_linux_c1-xlarge.txt*
The second file is the XML containing the results of the plan synthesis. An example is shown in Figure 6.3. This file contains a tag, *workload_log_file*, indicating the full path to the workload log file used for the plan. Then, the *single_virtual_machine* XML element contains all the details about the VM used to generate the reserved plan. In particular, this tag contains provider, region, os, name, μ and reservation period of the VM, so it can be uniquely identifiable in the cost model. Moreover, it contains the payments information for each type of reservation of this VM and for the on-demand one. Finally, it contains also the usage thresholds calculated, which have been explained in Section 5.4. The last tag, *vm_reserved*, contains the result of the plan synthesis. In fact, each child element holds the number of VMs that have been calculated to be reserved for each reservation type.

6.2.1.3 Results

The program calculated that, for the input trace and with the configuration provided, the optimal number of VMs to reserve are:

1. 2 VMs at reservation *light*
2. 2 VMs at reservation *medium*
3. 2 VMs at reservation *heavy*.

It is now explicated the input arrival thresholds that should trigger the activation of each VM.

Virtual Machines *heavy* reservation:

- 1 powering on when input arrival rate is greater than $0\,^{req}/_{s}$
- 1 powering on when input arrival rate is greater than $13.16\,^{req}/_{s}$.

handling a total of $28.39\,^{req}/_{s}$.

Virtual Machines *medium* reservation:

- 1 powering on when input arrival rate is greater than $28.39\,^{req}/_{s}$
- 1 powering on when input arrival rate is greater than $43.23\,^{req}/_{s}$

handling a total of $29.12\,^{req}/_{s}$.

Virtual Machines *light* reservation:

- 1 powering on when input arrival rate is greater than $72.35\,^{req}/_{s}$
- 1 powering on when input arrival rate is greater than $87.55\,^{req}/_{s}$

handling a total of $29.98\,^{req}/_{s}$.

All the 6 VMs reserved can sustain totally $117.53\,^{req}/_{s}$.

Figure 6.4 shows the evolution of the *log* workload during the reserved plan synthesis process.

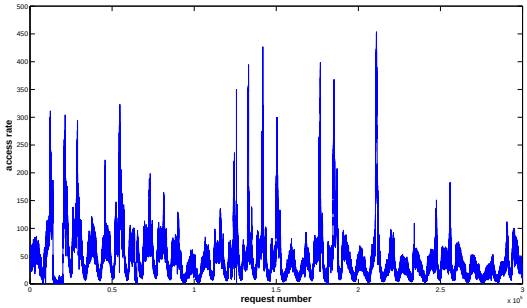
The first sub-figure 6.4a shows the initial workload trace, already normalized, i.e. each trace value has been divided by the interval I which, in this case, amounted of 10 seconds. It almost doesn't contain any values equal to zero; the maximum arrival rate is 453.5 $^{req}/_{s}$. The second sub-figure, 6.7b, depicts the trace after the workload supported by the two *heavy* reserved VMs calculated. The new trace is already for the 38.94% of the time at zero, while the maximum arrival rate becomes 425.11 $^{req}/_{s}$, which is exactly 28.39 $^{req}/_{s}$ less than the previous one. This is another proof of the bursty nature of the input workload trace: nearly 40% of the time the trace is below 28.39 $^{req}/_{s}$, which is the sustained workload of the 2 *heavy* reserved VMs. So, 40% of the trace is below the 6.26% of the total dynamic range of the arrival rates.
The third sub-figure 6.4c is made after removing the workload supported by the two *medium* reserved VMs. The trace is for 73.71% of the time at zero and the maximum arrival rate is 395.99 $^{req}/_{s}$. With these 2 VMs another 29.12 $^{req}/_{s}$ sustained workload is eroded from the trace. The amazing result of this fact is that 4 VMs are sufficient for the 74% of the time!
Finally, part 6.4d of the Figure shows the trace after removing the two *light* reservation VMs calculated. This is the trace that will be written as *remaining workload* output and used as input in the on-demand plan synthesis. This trace contains zeros for the 90.31% of the time, while the maximum arrival rate is 366.01 $^{req}/_{s}$. Again, if there was any doubt, showing this inner part of the study, it is confirmed the burstiness nature of the input trace. Only in 10% of the total time the input trace is above 117.53 $^{req}/_{s}$, which is the supported workload by all the reserved VMs, but when it does it reaches peaks 4 times higher than this value. The definition of *bursty*, high accesses peaks for short periods of time, applies perfectly to the situation.

6.2.2 On-demand plan synthesis

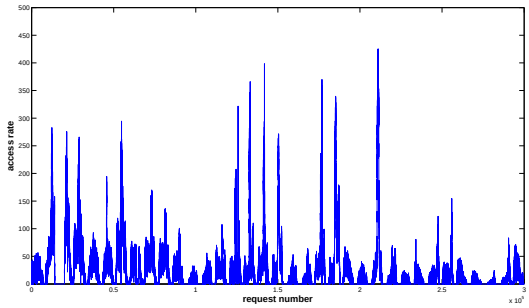
After generating the plan using exclusively one VM type at different reservations, the remaining workload output of that process in used as input for the plan synthesis created with exclusively on-demand VMs. Here there is no limitation in using only one type of VM. Different VMs with different service rates are used together to create buckets of VMs, where each bucket has the duty to support a certain workload. The size of each bucket is studied with the procedure explained in Section 5.5.

6.2.2.1 Configuration file

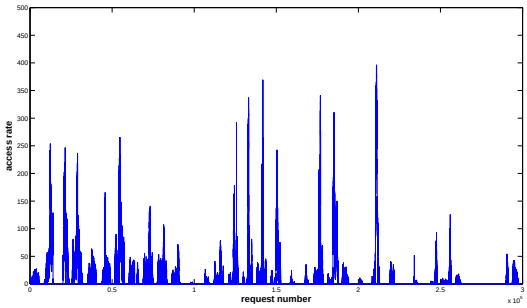
The configuration file for on-demand plan synthesis is showed in Figure 6.5.



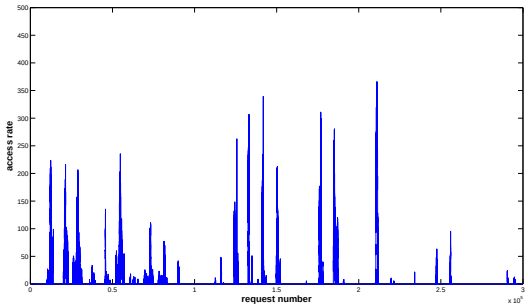
(a) Initial workload *log*



(b) Workload *log* after *heavy* VMs reservations



(c) Workload *log* after *medium* VMs reservations



(d) Workload *log* after *light* VMs reservations, becoming *remaining workload*

Figure 6.4: Evolution of *log* workload during the reserved plan synthesis

```
<?xml version="1.0" encoding="UTF-8" ?>
<config>
    <element name="secondsBetweenWorkloadValues" value="10" />
    <element name="verbosityLevel" value="2" />
    <element name="allowOverwrite" value="1" />
    <element name="writeToStdout" value="1" />
    <element name="costModelFileName" value="C:\Thesis\amazon.xml" />
    <element name="sla" value="[0 0; 0.1 1; 0 2; -0.1 inf]" />

    <on_demand>
        <element name="mu" value="7.07 14.45 18.98 20.34" />
        <element name="vmName" value="m1-small c1-medium m1-large c1-xlarge"/>
        <element name="regionsToAnalyze" value="eu-west-1" />
        <element name="os" value="linux" />
        <element name="bootingTime" value="120" />
        <element name="vmOnDemandOutputFileName" value="C:\Thesis\vmOnDemand" />
    </on_demand>
</config>
```

Figure 6.5: On-demand XML configuration file

In this case, all the parameters used for reserved plan synthesis are still in the configuration file. This has been designed as normally a complete plan synthesis involves firstly the reserved process and secondly the on demand process. Using the same configuration file for both, it allows the program to do in just one sweep both cases, in sequence, using the outputs of the first process as inputs of the second process. As many of the configuration parameters are in common between the two processes, they are specified only in one point, the reserved plan synthesis parameters. It is also possible to run just the on-demand plan synthesis, under the condition of writing all the required parameters in the right places.

The on-demand specific parameters are collected inside the tag *on_demand* of the XML configuration file. Let's review them.

- *mu*: specifies the service rate (μ) values for the VMs used in on-demand plan synthesis. There are 2 modes to write them. The first one is write all the μ values, in ascending order, separated by a single space. The number of μ values must be the same as the number of VMs written in the *vmName* tag. The second one is to write only the first μ value, the smallest one, and then let the program calculate the others with the *muScalingFactor* factor.

- *muScalingFactor*: optional parameter, it specifies the multiplying factor to obtain all the μvalues. It is used only if the parameter *mu* is composed of only one element and *vmName* contains multiple VMs. In this case, it is possible to specify a single scaling factor that is multiplied to *mu* in order to obtain the service rates values for the remaining VMs. It is also possible to specify an array of scaling factors, that must be of dimension $N - 1$, where N is the number of VMs written into the *vmName* tag. In that case, to obtain each μ value the corresponding scaling factor value is utilized.

- *vmName*: it specifies the names of the VMs used for on-demand plan synthesis. It can be a single VM or multiple VMs, in which case the different names are separated by a space. The order is important and should be from the least powerful to the most powerful, as in the *mu* tag.

- *regionsToAnalyze*: this parameter specifies the names of the regions that the program will use to generate the plan. In the case of multiple regions, their name are separated by a comma. If the will is to compare all the regions, it is sufficient to write "all" instead of all the region's names.

- *os*: this parameter specifies the OS of the VMs used in the plan synthesis. If set to

"all", all the OS are used.

- *workloadLogInputFileName*: it specifies the name of the file containing the remaining workload trace. In case of single run of both reserved and on-demand plan synthesis, this parameter is unknown as the output file has not been written yet. In that case, it can be left empty. Otherwise, it must contain the path to the remaining workload trace file, either absolute or relative to the program's base path.

- *bootingTime*: this parameter specifies, in seconds, the time that a VM takes to boot, i.e. from deployment to be fully operative.

- *vmOnDemandOutputFileName*: it specifies location and the initial file name of the output XML files generate by the on-demand plan synthesis. After the initial file name are appended the region and the OS that generated that plan. These files will contain the generated buckets. A file name example is *vmOnDemand_eu-west-1_linux_.xml*

Apart from these on-demand specific parameters, a subset of parameters from the reserved plan synthesis configuration is also used. These parameters must be present, even if there is no reserved plan synthesis. They are:

- *verbosityLevel*

- *allowOverwrite*

- *writeToStdout*

- *costModelFileName*

- *secondsBetweenWorkloadValues*

- *sla*

In case there is both reserved and on-demand plan synthesis, these parameters are shared between the two different procedures.

6.2.2.2 Program execution

After setting the parameters in the configuration file, called *config.xml*, it has been run the program again. The configuration used is showed in Figure 6.5.

```
<?xml version="1.0" encoding="UTF-8" ?>
<root>
<workload_log_file filename="C:\Thesis\workloadLogOutput_us-west-1_linux_c1-xlarge.txt" />
<virtual_machines><single_virtual_machine>
    <name>m1-small</name>
    <mu>7.07</mu>
    <region>us-west-1</region>
    <provider>Amazon</provider>
    <reservation_period>not_set</reservation_period>
    <os>linux</os>
    <payments>
        <vm_payment periodic_payment="0.0" running_payment="0.065" reservation_type="not_set" />
    </payments>
</single_virtual_machine>
<single_virtual_machine>
<single_virtual_machine>
<single_virtual_machine>
</virtual_machines>
<thresholds>
    <threshold id="1" workload="15.64" workload_step="15.64">
        <on_demand num="0" vm_name="m1-small" />
        <on_demand num="0" vm_name="c1-medium" />
        <on_demand num="0" vm_name="m1-large" />
        <on_demand num="1" vm_name="c1-xlarge" />
    </threshold>
    <threshold id="2" workload="35.98" workload_step="20.34">
        <on_demand num="2" vm_name="m1-small" />
        <on_demand num="0" vm_name="c1-medium" />
        <on_demand num="0" vm_name="m1-large" />
        <on_demand num="1" vm_name="c1-xlarge" />
    </threshold>
```

Figure 6.6: On-demand plan synthesis XML output file

To generate the on-demand plan for Amazon, the command issued was still

```
java -jar planGenerator.jar config.xml
```

The program, together with a sample configuration file, can be downloaded from [27].

This time, the process takes around 1.5 seconds. The output of the program is one XML file, *vmOnDemand_us-west-1_linux_.xml.xml*, written in the position specified in the configuration file, containing the results of the on-demand plan synthesis. An extract of the file is shown in Figure 6.6. This XML file contains 3 main tags inside the *root* tag: *workload_log_file*, *virtual_machines* and *thresholds*. Some tags are compressed in the figure to condense in one image all the main parts of the file.

The first tag, *workload_log_file*, contains in the attribute *name* the full path to the workload file used in the process, a.k.a. *workload remaining*.

The second tag, *virtual_machines*, contains M *single_virtual_machine* tags, where M is the number of VMs used in the process. Each *single_virtual_machine* element contains all the information about one of the VMs used in the process. The VMs are sorted in ascending order by their μ value. Naturally, having this process only on-demand VMs, their payments consists of only one element.

The third tag, *thresholds*, holds the information about the calculated buckets.

It contains a series of *threshold* tags, each one representing a bucket. Each bucket is identified by a unique sequential *id* attribute, that serve also as position identifier. The attribute *workload_step* indicates the amount of workload that the current bucket alone can sustain; i.e., is the bucket supported workload calculated in Section 5.5.6. The last attribute, *workload*, holds the maximum arrival rate value that *id* buckets can sustain; i.e., is the sum of all the *workload_step* values from the first bucket to the current one, included.
Inside the *threshold* tag there are M tags, one per each VM, called *on_demand*. The *on_demand* element has two attributes: *vm_name* and *num*. The first one specifies the type of VM, while the second one carry the number of VMs of the specified type that compose the current bucket. The *on_demand* elements appear in the same order as the *single_virtual_machine* tags.

6.2.2.3 Results

The program computed that given as input the *remaining workload* and using the VMs specified in the configuration file it is appropriate to use 18 buckets.

The composition of the buckets is the following:

- Bucket number 1. Maximum supported access rate: 15.64 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 15.64 $^{req}/_{s}$
  - 0 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*
- Bucket number 2. Maximum supported access rate: 20.34 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 35.98 $^{req}/_{s}$
  - 2 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*

  - 1 VMs *xlarge*

- Bucket number 3. Maximum supported access rate: 24.75 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 60.73 $^{req}/_{s}$

  - 0 VMs *small*
  - 1 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 4. Maximum supported access rate: 24.75 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 85.48 $^{req}/_{s}$

  - 0 VMs *small*
  - 1 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 5. Maximum supported access rate: 24.75 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 110.23 $^{req}/_{s}$

  - 0 VMs *small*
  - 1 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 6. Maximum supported access rate: 24.75 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 134.98 $^{req}/_{s}$

  - 0 VMs *small*
  - 1 VMs *medium*

  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 7. Maximum supported access rate: 29.98 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 164.96 $^{req}/_{s}$
  - 0 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 2 VMs *xlarge*

- Bucket number 8. Maximum supported access rate: 24.75 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 189.71 $^{req}/_{s}$
  - 0 VMs *small*
  - 1 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 9. Maximum supported access rate: 24.75 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 214.46 $^{req}/_{s}$
  - 0 VMs *small*
  - 1 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 10. Maximum supported access rate: 15.64 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 230.1 $^{req}/_{s}$
  - 0 VMs *small*

  - 0 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 11. Maximum supported access rate: 18.27 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 248.37 $^{req}/_{s}$
  - 1 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 1 VMs *xlarge*

- Bucket number 12. Maximum supported access rate: 11.74 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 260.11 $^{req}/_{s}$
  - 1 VMs *small*
  - 1 VMs *medium*
  - 0 VMs *large*
  - 0 VMs *xlarge*

- Bucket number 13. Maximum supported access rate: 6.7 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 266.81 $^{req}/_{s}$
  - 3 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 0 VMs *xlarge*

- Bucket number 14. Maximum supported access rate: 6.7 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 273.51 $^{req}/_{s}$

  - 3 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 0 VMs *xlarge*

- Bucket number 15. Maximum supported access rate: 6.7 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 280.21 $^{req}/_{s}$
  - 3 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 0 VMs *xlarge*

- Bucket number 16. Maximum supported access rate: 8.68 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 288.89 $^{req}/_{s}$
  - 4 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 0 VMs *xlarge*

- Bucket number 17. Maximum supported access rate: 13.59 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 302.48 $^{req}/_{s}$
  - 0 VMs *small*
  - 0 VMs *medium*
  - 1 VMs *large*
  - 0 VMs *xlarge*

- Bucket number 18. Maximum supported access rate: 6.7 $^{req}/_{s}$.
  Cumulative maximum sustained access rate: 309.18 $^{req}/_{s}$
  - 3 VMs *small*
  - 0 VMs *medium*
  - 0 VMs *large*
  - 0 VMs *xlarge*

The evolution of *remaining workload* trace is shown in Figure 6.7. Sub-figure 6.7a depicts the input trace, the *remaining workload* generated by the reserved plan synthesis. It is composed at the 90.3113% of zeros and the maximum value is 366.01 $^{req}/_{s}$.

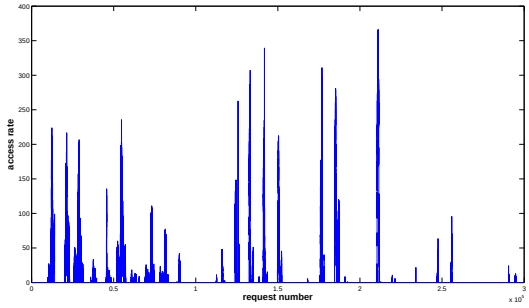
The on-demand plan synthesis starts and after calculating 4 buckets the trace remaining is shown in sub-figure 6.7b. The 4 buckets, combined, sustain a workload of 85.48 $^{req}/_{s}$ that has been removed from the initial trace. The trace, at this point, is composed of zeros at the 97.4667% and the maximum value is 280.53 $^{req}/_{s}$.

The process continues, and after reserving 4 more buckets the situation is the one depicted in sub-figure 6.7c. Eight buckets support 189.71 $^{req}/_{s}$ access requests, bringing up the number of trace elements to zero at 99.323%. The maximum value becomes 176.3 $^{req}/_{s}$.

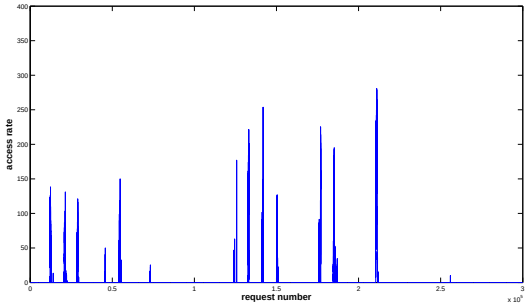
Sub-figure 6.7d shows the remaining trace after the calculation of the workload supported by 12 buckets, equal to 260.11 $^{req}/_{s}$. The number of trace elements at zero is the 99.82%, while the maximum arrival rate is 105.9 $^{req}/_{s}$.

In sub-figure 6.7e the buckets calculated are 16, supporting an arrival rate of 288.89 $^{req}/_{s}$. There are very few positive trace elements left, as the number of *workload remaining* set to zero is the 99.9153%. The maximum arrival rate is 77.12 $^{req}/_{s}$.

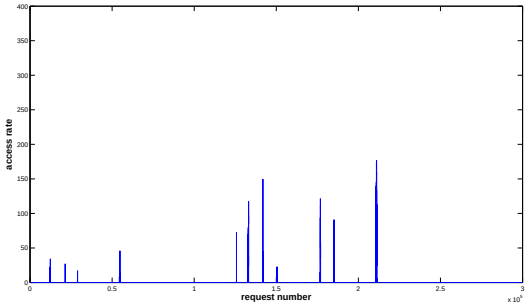
The last sub-figure 6.7f shows the trace left at the end of the on-demand plan synthesis process. In total, 18 buckets with a overall support of 309.18 $^{req}/_{s}$ have been calculated. The remaining trace contains a total of 114 elements with a value greater than zero over a total of 300.000 trace elements, the 0.038% (or, equivalently, the 99.962% of the trace is at zero). The maximum value of these 114 elements is 56.83 $^{req}/_{s}$. The bucket creation process stopped because this spikes lasts for so little time that it is not appropriate to switch on new VMs; i.e., when the VMs finished their booting and were ready to serve requests, the spikes in the workload will have finished.



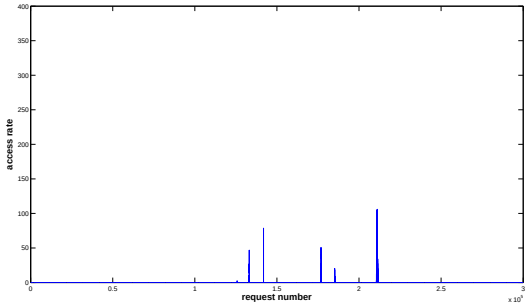
(a) Initial trace, *workload remaining*



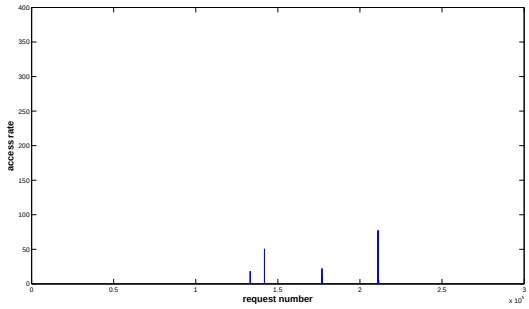
(b) Trace after 4 buckets



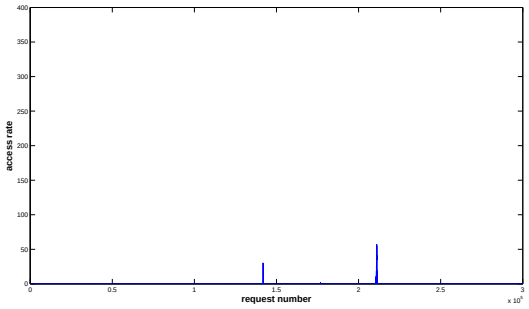
(c) Trace after 8 buckets



(d) Trace after 12 buckets



(e) Trace after 16 buckets



(f) Final trace, after 18 buckets

Figure 6.7: Evolution of *remaining workload* during the on-demand process synthesis

Chapter 7

Conclusions

"Beam me up, Scotty"

Captain Kirk

It is time to draw the conclusions of the work done in this thesis.

We have shown how a relevant part of Internet services experiences request arrivals with a bursty nature, meaning that a high number of requests, even an order of magnitude higher than the normal arrival rate, are concentrated in a short period of time. Dealing with this problem with traditional methods, i.e. physical servers, is costly and not effective, as the dimensioning of the servers is done a priori and so there is a upper arrival rate limit that the servers can sustain. If the arrival rate is above their dimensioned capacity, the service provided with the servers becomes slow or can even crash, with obvious monetary repercussions for the service owner and unsatisfactory users experience.

Cloud Computing is a new and important paradigm that rise in popularity and adoption among companies and end users. It consists of renting online virtual machines just for the time needed to perform some work or to offer a service. Everything below the OS is a commodity offered by the cloud provider, so the client should never bother about electricity, servers' hardware, maintenance, Internet connection, etc. because it is all taken care of. This fact, coupled with the intrinsic property of Cloud Computing called *elasticity*, is the main reason that makes convenient the usage of such a platform instead of a private server.

This thesis has proposed a method to synthesize adaptation plans that manage cloud elasticity in terms of number of active virtual machines. To be able to generalize

the core method of the adaptation plan synthesis we have proposed a cost metamodel, which enables the representation of the costs of VMs of different cloud providers, acting as an abstraction layer between the provider's costs implementations and the developed process of plan synthesis. We have also developed a tool that automatically generates the cost model for Amazon, Rackspace and HP cloud providers. This tool is easily extendible to other online providers and, being written in Java, is multiplatform.

During the development of this thesis I have been in contact with some companies that offer cloud services, which showed interest for the work done in this thesis and were very interested in the activation/deactivation of VMs just-in-time. This which is a very plausible future work using the work of this thesis as basis.

It is worth mentioning that the study performed during this thesis about VMs cost and usage thresholds led us to discover an incoherence in the instance prices of one of the most important cloud providers in the world. We notified them the incoherence we found in their prices, and the result was that they acknowledge our notification and updated worldwide the incoherent price. These inconsistencies are not easily seen at a first sight in the website, but some study about prices, as the one I performed, was required in order to realize the incoherence.

Most of cloud providers offer the possibility to reserve VMs, i.e. they allow paying an amount of money upfront in order to decrease the hourly cost of the VMs. This has been taken into account by our approach. The first part of the adaptation plan, in fact, is synthesized using reserved VMs. The expected workload of the application we are studying is required for this process, together with the SLA and the characteristics of the reserved VM used. For synthesizing the plan, we have cared of requiring only information that is feasible to believe that can be provided by the cloud client as the expected workload, SLA and application's performance in a VM. The excepted workload is composed by the log trace of the application under study, where each value consists of the number of requests received by the application during a sequential interval I of time. The SLA, instead, models the service's performance and it consists of a series of stepwise rewards obtained if the service responds within a certain amount of time. This modeling of the performance of the service allows a reward when the application responds quickly to the requests and a penalty when the response times exceeds a configurable amount of time. Finally, the VM characteristics are all the information about a VM, including provider, OS, billing prices, ect. and its performance, i.e. the service rate μ. This value should be obtained by the user either by online public benchmarks of the VM used or directly by measuring the service's performance on a real instance of the VM.

It has been shown how the reserved VM plan synthesis take care of the most

stable part of the workload, which is usually 90% of the workload trace analyzed. The VM calculated are therefore effective for the 90% of the total time the service is used. The remaining 10% of the workload trace is the one that contains the bursty behavior, with peaks higher than the normal access rate for even 5 times, if not more. It is analyzed by the on-demand plan synthesis, which as the name states uses only on-demand VMs in the process. As the bursts of requests are unpredictable and have a rapid growth, it has been demonstrated how deploying one VM at a time to deal with them is ineffective. So, the on-demand plan synthesis creates *buckets* of VMs that, deployed sequentially, deal with the bursts of requests in a more effective way, maximizing the service's profit.

To store the output of the approach, I have chosen a easily understandable format: XML. Output of the plan synthesis process are two XML files, one for reserved VMs and one for on-demand VMs, containing the information about the VMs used in the process and their calculated quantity to sustain the workload generated by the service modeled.

With this information the clout client can use cloud providers price calculators to estimate in a more accurate manner the cost of deploying such a service on the chosen provider and decide whether it is more convenient the private servers solution or the cloud computing one. In case the second option is the best one, the user already have all the activation thresholds for the VMs.

As future work, I plan to use the work in this thesis to study deeply the just-in-time VM activation/deactivation, challenge in which the cloud companies I contacted were interested. This future work will include the management of the short-time variability of the workload, i.e, the slight variations that happen in intervals of few seconds. This variability can lead to adjustments that can be called as *false positives*. In this moment, I know that there are several research groups and many research works that are dealing with this challenge, i.e. [29, 14]. We have preferred not to implement any of them ad-hoc in the thesis, but leave the work in the thesis as a reliable background for a subsequent research on the dealing with false positives.

Bibliography

[1] Json (javascript object notation). http://www.json.org/. RFC 4627.

[2] Amazon. Amazon cloud computing definition. http://aws.amazon.com/what-is-cloud-computing/, November 2013.

[3] Amazon. Amazon ec2 pricing. http://aws.amazon.com/ec2/pricing/, November 2013.

[4] Amazon. Amazon prices calculator. http://calculator.s3.amazonaws.com/calc5.html, November 2013.

[5] T. Andrews. Computation time comparison between matlab and c++ using launch windows. http://digitalcommons.calpoly.edu/cgi/viewcontent.cgi?article=1080&context=aerosp.

[6] Aruba. Aruba Cloud. http://www.cloud.it/, November 2013.

[7] B. H. Cheng, R. Lemos, H. Giese, P. Inverardi, J. Magee, J. Andersson, B. Becker, N. Bencomo, Y. Brun, B. Cukic, G. Marzo Serugendo, S. Dustdar, A. Finkelstein, C. Gacek, K. Geihs, V. Grassi, G. Karsai, H. M. Kienle, J. Kramer, M. Litoiu, S. Malek, R. Mirandola, H. A. Müller, S. Park, M. Shaw, M. Tichy, M. Tivoli, D. Weyns, and J. Whittle. Software engineering for self-adaptive systems. pages 1–26, 2009.

[8] B. H. C. Cheng, R. de Lemos, H. Giese, P. Inverardi, and J. Magee, editors. *Software Engineering for Self-Adaptive Systems [outcome of a Dagstuhl Seminar]*, volume 5525 of *Lecture Notes in Computer Science*. Springer, 2009.

[9] CloudHarmony. Performance benchmarks of online virtual machines. http://cloudharmony.com/benchmarks, 2013.

[10] W. W. W. Consortium. Xml. http://www.w3.org/standards/xml/.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking

cloud serving systems with ycsb. In *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[12] D. Crockford. Json-lib. http://json-lib.sourceforge.net/, 2010.

[13] R. de Lemos, H. Giese, H. A. Müller, M. Shaw, J. Andersson, L. Baresi, B. Becker, N. Bencomo, Y. Brun, B. Cukic, R. Desmarais, S. Dustdar, G. Engels, K. Geihs, K. M. Goeschka, A. Gorla, V. Grassi, P. Inverardi, G. Karsai, J. Kramer, M. Litoiu, A. Lopes, J. Magee, S. Malek, S. Mankovskii, R. Mirandola, J. Mylopoulos, O. Nierstrasz, M. Pezzè, C. Prehofer, W. Schäfer, R. Schlichting, B. Schmerl, D. B. Smith, J. P. Sousa, G. Tamura, L. Tahvildari, N. M. Villegas, T. Vogel, D. Weyns, K. Wong, and J. Wuttke. Software engineering for self-adaptive systems: A second research roadmap. In R. de Lemos, H. Giese, H. A. Müller, and M. Shaw, editors, *Software Engineering for Self-Adaptive Systems II*, volume 7475, pages 1–32. Springer-Verlag, 2013. DOI: 10.1007/978-3-642-35813-5_1.

[14] A. Gambi, A. Filieri, and S. Dustdar. Iterative test suites refinement for elastic computing systems. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 635–638, New York, NY, USA, 2013. ACM.

[15] P. D. H. Giese. Self-adaptive website. http://www.self-adaptive.org/, November 2013.

[16] Google. Google json library. https://code.google.com/p/google-gson/, 2013.

[17] P. M. Harchol-Balter. *Performance Modeling and Design of Computer Systems Queueing Theory in Action*. Cambridge University Press, February 2013.

[18] HP. HP Cloud website. http://www.hpcloud.com/, November 2013.

[19] C. Hubbard. Flexjson. http://flexjson.sourceforge.net/, 2010.

[20] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1984.

[21] Lunacloud. Luna Cloud. http://www.lunacloud.com/, November 2013.

[22] Mathworks. Matlab. http://www.mathworks.com/products/matlab/.

[23] P. Mell and T. Grance. The NIST definition of cloud computing. Technical report, July 2009.

[24] Microsoft. Azure. http://www.windowsazure.com/en-us/pricing/details/virtual-machines/, 2013.

[25] Microsoft. Azure prices calculation. http://www.windowsazure.com/en-us/pricing/calculator/, November 2013.

[26] Oracle. Java. http://www.oracle.com/technetwork/java/index.html.

[27] F. Pegorao. Thesis website. https://sites.google.com/site/costeffectiveadaptationplan/home/, November 2013.

[28] F. Pegoraro. Amazon discussion thread. https://forums.aws.amazon.com/thread.jspa?threadID=128095.

[29] D. Perez-Palacin, R. Mirandola, and J. Merseguer. Qos and energy management with petri nets: A self-adaptive framework. *Journal of Systems and Software*, 85(12):2796 – 2811, 2012. <ce:title>Self-Adaptive Systems</ce:title>.

[30] Rackspace. Rackspace prices calculator. http://www.rackspace.com/calculator/, November 2013.

[31] Rackspace. Rackspace public cloud pricing. http://www.rackspace.com/cloud/public-pricing/, 2013.

[32] Wikipedia. Hypervisor types comparison. http://en.wikipedia.org/wiki/File:Hyperviseur.png, November 2013.

APPENDICES

Appendix A

List of online providers

This is a partial list of online cloud providers, sorted in alphabetically order.

- Amazon[3]
- Aruba[6]
- Azure[24]
- HP Cloud[18]
- LunaCloud[21]
- Rackspace[31]

Appendix B

Algorithms

Algorithmus B.1 Calculate VMs to reserve for each type of reservation

Require: sla, workload log , demand e_i, usage thresholds $usage$

Ensure: VMs to reserve for each type of reservation $reservation[]$, adaptation rules, remaining workload

1: set $n = N$ N is the number of entries in $usage$: usage.length()
2: set $reservation_i = 0 \; \forall i \in [1..N]$
3: **while** $n > 1$ **do**
4: set $workloadThresholds[] = EmptyArray$
5: set $j = 1; limitArrivalRate = 0$
6: **while** $limitArrivalRate \leq max(log)$ **do**
7: $limitArrivalRate \leftarrow calculateThreshold(usage_n.cost, e_i, sla, j)$
8: $workloadThresholds[j] \leftarrow limitArrivalRate$
9: $j \leftarrow j + 1$
10: **end while**
11: $resourceNeedsLog \leftarrow createResourceNeeds(log, workloadThresholds[])$
12: $countOfResourceUsage \leftarrow count(resourceNeedsLog)$
13: $numberOfVMsToReserve \leftarrow selectNumberOfVMs(countOfResourceUsage, usage_{n-1})$
14: $reservation_n \leftarrow (usage_n.cost, numberOfVMsToReserve)$
15: $supportedWorkload \leftarrow calculateEquivalentProfit(usage_n.cost,$ $numberOfVMsToReserve, usage_{n-1}.cost, 1, sla)$
16: $log_i \leftarrow max(log_i - supportedWorkload, 0)$
17:
18: $n \leftarrow n - 1$
19: **end while**
20: **return** Reservation, Rules, log