

Dynamic scaling of a web-based application in a Cloud Architecture

MD. IQBAL HOSSAIN
and
MD. IQBAL HOSSAIN



**KTH Information and
Communication Technology**

Degree project in
Communication Systems
Second level, 30.0 HEC
Stockholm, Sweden

Dynamic scaling of a web-based application in a Cloud Architecture

Md. Iqbal Hossain (Older)
mihossai@kth.se

And

Md. Iqbal Hossain (Younger)
mihiqbal@kth.se

2014-02-28

Master's thesis

Examiner and academic adviser
Professor Gerald Q. Maguire Jr.

School of Information and Communication Technology
KTH Royal Institute of Technology
Stockholm, Sweden

Abstract

With the constant growth of internet applications, such as social networks, online media, various online communities, and mobile applications, website user traffic has grown, is very dynamic, and is oftentimes unpredictable. These unpredictable natures of the traffic have led to many new and unique challenges which must be addressed by solution architects, application developers, and technology researchers. All of these actors must continually innovate to create new attractive application and new system architectures to support the users of these new applications. In addition, increased traffic increases the demands for resources, while users demand even faster response times, despite the ever-growing datasets underlying many of these new applications. Several concepts and best practices have been introduced to build highly scalable applications by exploiting cloud computing. As no one who expect to be or remain a leader in business today can afford to ignore cloud computing.

Cloud computing has emerged as a platform upon which innovation, flexibility, availability, and faster time-to-market can be supported by new small and medium sized enterprises. Cloud computing is enabling these businesses to create massively scalable applications, some of which handle tens of millions of active users daily. This thesis concerns the design, implementation, demonstration, and evaluation of a highly scalable cloud based architectures designed for high performance and rapid evolution for new businesses, such as Ifoodbag AB, in order to meet the requirement for their web-based application. This thesis examines how to scale resources both up and down dynamically, since there is no reason to allocate more or less resources than actually needed. Apart from implementing and testing the proposed design, this thesis presents several guidelines, best practices and recommendations for optimizing auto scaling process including cost analysis. Test results and analysis presented in this thesis, clearly shows the proposed architecture model is strongly capable of supporting high demand applications, provides greater flexibility and enables rapid market share growth for new businesses, without their need to investing in an expensive infrastructure.

Keywords: *cloud computing, internet, application scalability, internet traffic, performance.*

Sammanfattning

Med den ständiga tillväxten av Internet- applikationer, såsom sociala nätverk, online media, olika communities och mobila applikationer, har trafiken mot webbplatser ökat samt blivit mycket mer dynamisk och är ofta oförutsägbara. Denna oförutsägbara natur av trafiken har lett till många nya och unika utmaningar som måste lösas med hjälp av lösningsarkitekter, applikationsutvecklare och teknikforskare. Alla dessa aktörer måste ständigt förnya sig för att skapa nya attraktiva program och nya systemarkitekturer för att stödja användarna av dessa nya tillämpningar. Dessutom ökar den ökade trafikmängden krav på resurser, samtidigt som användarna kräver ännu snabbare svarstider, trots den ständigt växande datamängden som ligger som grund för många av dessa nya tillämpningar. Flera koncept och branchstandarder har införts för att bygga skalbara applikationer genom att utnyttja ”molnet” (”cloud computing”), eftersom att ingen som förväntar sig att bli eller förbli en ledare i näringslivet idag har råd att ignorera ”molnet”.

Cloud computing har vuxit fram som en plattform på vilken innovation, flexibilitet, tillgänglighet och snabbhet till marknaden kan uppnås av nya, små och medelstora företag. Cloud computing är möjligt för dessa företag att skapa mycket skalbara applikationer, vilka kan hantera tiotals miljoner aktiva användare varje dag. Detta examensarbete handlar om utformning, genomförande, demonstration och utvärdering av en mycket skalbar molnbaseradarkitektur som utformats för höga prestanda och snabb utveckling av nya företag, såsom Ifoodbag AB, för att uppfylla kravet på deras webb- baserad applikation. Detta examensarbete undersöker hur man både skalar upp och ner dynamiskt, eftersom det inte finns någon anledning att tillägna applikationer mer eller mindre resurser än vad som faktiskt behövs för stunden. Som en del av examensarbetet implementeras och testas den föreslagna utformningen, samt presenterar flera riktlinjer, branchstandarder och rekommendationer för att optimera automatisk skalning av processer. Testresultat och de analyser som presenteras i detta examensarbete, visar tydligt att den föreslagna arkitekturen/modellen kan stödja resurskrävande applikationer, ger större flexibilitet och möjliggör snabb tillväxt av marknadsandelar för nya företag, utan att deras behov av att investera i en dyr infrastruktur.

Nyckelord: Cloud computing, molntjänster, Internet, skalbarhet för applikationer, internettrafik, prestanda

Acknowledgements

Iqbal Hossain (Older):

First I would like to thank almighty Allah for giving me strength and the patience to accomplish this thesis project. I am thankful to my mom (Mrs. Nurjahan Begum) and my older brothers for their unconditional support and motivation, even from thousand miles away. I am also grateful to my beloved wife (Rahena Easmin Ratna) for her continuous inspiration and insisting to complete this thesis work. I would also like to thank my 4 years old charming boy (Farhan Iqbal Taseen) for not demanding too much attention from me during this thesis project. I am also thankful to my colleagues Tobias Östensson and Marked Jakob for helping us writing abstract in Swedish. I would also like to thank my friends and all family members across the globe for their encouragement during all this time. Last but not the least I would like to thank our supervisor and examiner (Professor Gerald Q. Maguire Jr.) who introduced us the idea of working with cloud architecture and his quick invaluable insights have always been very helpful throughout the project.

Iqbal Hossain (Younger):

Praise to almighty, the origin of knowledge, who enables me to undertake and accomplish this thesis work. My special gratitude goes to our supervisor and examiner Professor Gerald Q. Maguire Jr. whose precious guidance accompanied me during this research work. I would like to sincerely thank to my program coordinator May-Britt Eklund-Larsson for her help and kind cooperation during my studies. My deepest gratitude goes to my parents for their infinite support throughout my life. Finally, I would like to thank my brother, sisters and friends for encouraging me during all this time. The efforts of myself, inspirations of many, have led to a successful completed of my thesis project.

Table of Contents

Abstract	i
Sammanfattning.....	iii
Acknowledgements	v
Table of Contents	vi
List of Figures	ix
List of Tables	xi
List of Acronyms and Abbreviations.....	xiii
Contribution of the Authors	xv
1 Introduction	1
1.1 Overview	1
1.2 Problem Statement.....	1
1.3 Research methodology	4
1.4 Project goals.....	4
1.5 Project scope	4
1.6 Structure of this thesis.....	5
2 General Background	7
2.1 What is cloud computing?.....	7
2.1.1 On-demand self-service	8
2.1.2 Ubiquitous network access	9
2.1.3 Elasticity and scalability	9
2.1.4 Horizontal and Vertical Scalability.....	9
2.1.5 Resource pooling.....	10
2.1.6 Pay-per-use	11
2.1.7 Self-managed platform	11
2.1.8 Standardized interfaces.....	11
2.1.9 Quality of Service (QoS).....	11
2.1.10 Reliability.....	12
2.2 Cloud computing service models.....	12
2.2.1 Infrastructure as a Service (IaaS)	13
2.2.2 Platform as a Service (PaaS)	13
2.2.3 Software as a Service (SaaS).....	13
2.3 Cloud computing deployment models	14
2.3.1 Public cloud.....	14
2.3.2 Private cloud	14
2.3.3 Community cloud.....	15
2.3.4 Hybrid cloud.....	15
2.4 Virtualization	15
2.4.1 Server / hardware virtualization	16
2.4.2 Kernel based virtual machine (KVM)	17
2.5 Lamp stack – Overview	18
2.5.1 Linux.....	18
2.5.2 Apache	19
2.5.3 MySQL.....	19
2.5.4 PHP.....	19
2.6 Current cloud service providers	19

2.7	Related work	21
3	Scalable Cloud Architecture for a Web Application.....	23
3.1	Scalable Web Application Reference Architecture.....	23
3.2	Load Balancing Tier	23
3.3	Application Tier	24
3.4	Caching Tier	25
3.5	Database Tier	25
3.6	Management Node/Nodes.....	26
3.7	Security guidelines in the architecture	27
4	Detail Descriptions of the Component in the Proposed Architecture.....	29
4.1	DNS server.....	29
4.1.1	DNS infrastructure	30
4.1.2	DNS name resolving process.....	31
4.1.3	DNS security aspects.....	32
4.2	Load Balancer (Squid/HA Proxy)	32
4.2.1	How to calculate response time.....	34
4.2.2	Different types of load balancers	34
4.2.3	Squid	35
4.2.4	HA Proxy	35
4.3	Web server/Application server.....	36
4.4	Caching web data (memcached)	37
4.5	Database	39
4.6	Cloud Storage.....	40
4.7	Management node.....	41
5	Implementation	43
5.1	Experimental Setup	43
5.2	Motivation for choosing Cloudify	44
5.3	Introduction to Cloudify.....	45
5.4	Deploying Cloudify.....	45
5.4.1	Boot-strapping Cloudify on EC2.....	50
5.4.2	Deploying the Ifoodbag application recipe.....	51
5.5	Motivation for choosing AWS.....	52
5.5.1	The differences that distinguish AWS	52
5.5.2	Introduction to AWS	53
5.5.2.1	Amazon Elastic Compute Cloud (Amazon EC2)	53
5.5.2.2	Elastic Load Balancing.....	53
5.5.2.3	Amazon Virtual Private Cloud (Amazon VPC).....	53
5.5.2.4	Amazon ElastiCache.....	54
5.5.2.5	Amazon Route 53	54
5.5.2.6	Amazon Elastic Block Storage (EBS).....	54
5.5.2.7	Amazon Relational Database Service (Amazon RDS).....	54
5.5.3	Amazon EC2 instance types	54
5.5.4	Amazon EC2 pricing	56
5.5.5	EC2 cloud setup for Cloudify	56
5.6	Webserver load or performance measurement tool - Httppref.....	58

6	Results and analysis	59
6.1	Successful deployment of the Ifoodbag application on EC2 from the management machine.....	59
6.2	Scalability Guidelines and Analysis.....	60
6.2.1	Scalability Guidelines.....	60
6.2.2	Scalability Analysis.....	63
6.3	Cost analysis	65
6.3.1	Utility style pricing for cloud	65
6.3.2	Cost factors.....	66
6.3.3	Instance type selection	68
6.3.4	Total Cost of Ownership (TCO) of running a web application in a cloud.....	68
6.3.5	Cost Analysis Summary	72
6.4	Comparison with some other solutions and some recommendations.....	72
7	Conclusions and Future Work.....	75
7.1	Conclusions	75
7.2	Future Work	75
7.3	Reflections	76
7.3.1	Social aspects.....	76
7.3.2	Economic aspects.....	76
7.3.3	Sustainability aspects	77
7.3.4	Legal and ethical aspects	77
	References	79
	Appendix A: Installation of Cloudify.....	87
	Appendix B: Configuration of Cloud controllers and cloud drivers.....	89
	Appendix C: Writing Ifoodbag Application Recipe.....	93
	Appendix D: Implementing Auto-Scaling Policies.....	97
	Appendix E: Deploying Ifoodbag Application in EC2	99
	Appendix F: Amazon EC2 Management Console	101
	Appendix G: Cloudify Web Management Console.....	103
	Appendix H: Simulating Auto-Scaling Process	105

List of Figures

Figure 1-1:	Traditional Infrastructure Model	2
Figure 1-2:	Scalable Cloud Architecture Model	3
Figure 2-1:	Basic single N-tier Architecture (Adapted from Figure 1, page 9 of [21])	10
Figure 2-2:	Horizontally scaled load balancing and web-tier and vertically scaled database tier (Adapted from Figure 2, page 9 of [21])	10
Figure 2-3:	Server stack comparison between on-premise infrastructure, IaaS, PaaS, and SaaS (Adapted from Wely Lau's online article[22])	12
Figure 2-4:	Cloud computing stack (Adapted from Figure 1.3, page 14 of [31])	14
Figure 2-5:	Basic architecture of virtualization [17]	15
Figure 2-6:	Bare metal/native and hosted hypervisor [17]	16
Figure 2-7:	The hypervisor manages VMMs that host virtual machines [38]	17
Figure 2-8:	LAMP architecture (adapted from [35])	18
Figure 3-1:	Scalable reference architecture for Ifoodbag's web-application	23
Figure 3-2:	Database Tier for Ifoodbag Web-Application	26
Figure 3-3:	Architecture with security guidelines as recommended in [63]	27
Figure 4-1:	The normal DNS resolution process (adapted from [67]).	30
Figure 4-2:	Partial DNS Name Space Hierarchy (adapted from [66])	31
Figure 4-3:	DNS name resolving process (adapted from [69]).	32
Figure 4-4:	Load balancing for balancing load among multiple application servers (adapted from [73]).	33
Figure 4-5:	Master-slave replication of databases (adapted from [88])	39
Figure 5-1:	High level experimental setup using Cloudify and EC2 clouds	43
Figure 5-2:	Cloudify Shell	46
Figure 5-3:	Achieving the No Code Change objective	47
Figure 5-4:	Achieving the No Lock-in objective	48
Figure 5-5:	Achieving the Full control objective	48
Figure 5-6:	Cloudify Architecture	49
Figure 5-7:	Bootstrapping Cloudify on EC2	50
Figure 5-8:	Cloudify Web Management Console	50
Figure 5-9:	Deploying the sample Ifoodbag web application locally	51
Figure 5-10:	Ifoodbag web application launched in local-cloud	51
Figure 5-11:	Create new a key pair for Amazon EC2	57
Figure 5-12:	Added a new key pair named ifoodbag with a secret key	57
Figure 5-13:	Creating an Access Key ID in Amazon EC2	57
Figure 6-1:	Ifoodbag application on EC2 cloud	59
Figure 6-2:	Cloudify web-management console for Ifoodbag application	59

Figure 6-3: Defined metrics for Ifoodbag application 60

Figure 6-4: Assumed traffic pattern of a production version of the
iFoodbag application..... 61

Figure 6-5: Ping-Pong Effect 61

Figure 6-6: Scale Up Process 65

Figure 6-7: Scale Down Process 65

Figure 6-8: Monthly TCO of traditional infrastructure versus cloud..... 70

Figure 6-9: Yearly TCO of traditional infrastructure versus a cloud..... 72

List of Tables

Table 4-1:	Different types of Top-Level Domains (TLD).....	30
Table 4-2:	DNS name resolving process [68]	31
Table 4-3:	Different types of load balancers [76]	35
Table 4-4:	Different types of web servers [82, 83]	37
Table 4-5:	Different tasks perform by memcached [60].	38
Table 4-6:	Advantages of master-slave replication [88]	40
Table 5-1:	Experimental configuration.....	44
Table 5-2:	Amazon EC2 instance types.....	55
Table 5-3:	Amazon EC2 pricing for Linux OS and US East (N. Virginia) region	56
Table 6-1:	Amazon EC2 Scale up Time	62
Table 6-2:	Amazon EC2 Scale Down Time.....	62
Table 6-3:	Results of implementing the algorithm with $RPS_{Peak}=1300$, $RPS_{Min}=50$, $D=2$, $U=3$, $T_D=40$, $T_U=80$	64
Table 6-4:	Utility Style Pricing [120, 121]	66
Table 6-5:	Different types of cost factors [119, 121]	66
Table 6-6:	Types of instances according to costs saving.....	68
Table 6-7:	Saving of reserved instance types over on-demand instances	68
Table 6-8:	TCO of on-premises infrastructure vs. cloud infrastructure	69

List of Acronyms and Abbreviations

AMI	Amazon Machine Image
API	Application programming interface
ASG	Auto Scaling Group
AWS	Amazon Web Services
BSD	Berkeley Software Distribution
CBS	Cloud Block Storage
CPU	Central Processing Unit
CRM	Customer relationship management
CSS	Cascading Style Sheets
DNS	Domain Name System
DNSSEC	DNS Security
EBS	Elastic Block Storage
EC2	Elastic Compute Cloud
ELB	Elastic Load Balancing
ESXi	Elastic Sky X
FTP	File Transfer Protocol
GUI	Graphical User Interface
HA Proxy	High Availability Proxy
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
IaaS	Infrastructure as a Service
IIS	Internet Information Services
I/O	Input / Output
IP	Internet Protocol
ISP	Internet Service Provider
IT	Information Technology
JEE	Java Enterprise Edition
JS	JavaScript
KVM	Kernel Virtual Machine
LAMP	Linux, Apache, MySQL, PHP
LAN	Local Area Network
LB	Load Balancer
LRU	Least Recently Used
NIST	National Institute of Standards and Technology
OS	Operating System
PaaS	Platform as a Service
PHP	Hypertext Preprocessor
PUE	Power Usages Effectiveness
QEMU	Quick Emulator
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational state transfer

RDS	Relational Database Service
RPS	Request Per Second
RTT	Round Trip Time
SaaS	Software as a Service
SLA	Service Level Agreement
SOA	Service Oriented Architecture
SSD	Solid State Drive
SSL	Secure Socket Layer
TCO	Total Costs of Ownership
TCP	Transmission Control Protocol
TLD	Top Level Domain
USM	Universal Service Manager
VM	Virtual Machine
VMM	Virtual Machine Monitor
VPC	Virtual Private Cloud
VPN	Virtual Private Network
VT-x	Virtual Technology
W3C	World Wide Web Consortium
WAN	Wide Area Network

Contribution of the Authors

Chapter	Sections	Subject	Author (s)	Assist
		Abstract	Iqbal (Older)	
1		Introduction	Iqbal (Older)	
2		General Background	Iqbal (Younger)	Iqbal (Older) assist in selecting study materials and covering different topics
3		Scalable Cloud Architecture Design for Web Application	Iqbal (Older)	Iqbal (Younger) assist in designing the architecture
4		Detail Descriptions of the Component in the Proposed Architecture	Iqbal (Younger)	Iqbal (Older) assist in covering topics
5	5.1-5.4	Implementation	Iqbal (Older)	
5	5.5-5.6	Implementation	Iqbal (Younger)	
6	6.1-6.2, 6.4	Results and analysis	Iqbal (Older)	
6	6.3	Results and analysis	Iqbal (Younger)	
7	7.1-7.2	Conclusions and Future Work	Iqbal (Older)	
7	7.3	Conclusions and Future Work	Iqbal (Younger)	

1 Introduction

This chapter describes the main purpose and the problem statement that motivated and guided this thesis project. Following this the chapter describes the research methodology that was selected for this project. The following two sections present the goals of our thesis project and its scope. The chapter ends with a description of the structure of the entire thesis.

1.1 Overview

Cloud computing extends information technology (IT) computing resources across the Internet. Today clouds are made available by various cloud service providers. Usually, users are not concerned with the underlying technologies or challenges that must be overcome for the cloud service provider to support a scalability diverse infrastructure. These users are also unconcerned with the number of servers or details of the other resources that are necessary to support their currently desired computing/storage/networking requirements, these users simply want to pay for the computing capacity which they use and they expect the capacity to scale up or down to meet their current requirements in an on-demand basis.

The numbers of applications, which exploit the cloud-computing model, are increasing rapidly as connectivity costs fall and computing hardware becomes more efficient – especially when operated on a large scale. Cloud services have extended beyond web applications to include data storage, raw computing, and access to different specialized services, such as Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Cloud based computing is becoming the ideal environment for scalable applications because it allows for rapid resource allocation in times of high demand, as well as resource de-allocation as demand declines. With a suitably scalable architecture, the resources and infrastructure of the cloud can accommodate all of the different phases of an application’s lifecycle, thus providing a single consistent context in which to bring an application from conception to development, then from production to maintenance, and finally, to a gradual end of life. Additionally, the cloud has become a popular solution to the problem of horizontal scalability. As a result “cloud application scalability” is gaining a tremendous amount of attention by both practitioners and researcher, see for example [1,2,3,4,5,6].

1.2 Problem Statement

Application scalability may take many forms, but in principle an application and its underlying infrastructure should adapt to the dynamically changing conditions (demands and available resources at various costs) to promote the availability and reliability of a service, while minimizing the cost for the application service provider. With the increase in numbers and size of on-line communities there has been an increasing effort to exploit cross-functionalities across these communities. However, application service providers have encountered problems due to the unpredictable demand for their application(s), especially when external events can lead to unprecedented traffic levels to and from their application* [7]. This dynamic nature of demand and traffic drives the need for a massively scalable solution to enable the availability (and reliability) of web-based applications.

In the earlier traditional infrastructure model, two approaches were taken in order to address the unpredictability of site traffic and system load, each of which is illustrated in Figure 1-1. One approach was to overprovision resources to handle spikes that may occur in traffic. Although this enables an application to increase its availability in high-traffic situations, it does not make effective use of resources - because a portion (and perhaps the

* For example, flash crowds or denial of services attacks can both lead to very high levels of traffic to/from an application.

majority) of these resources are idle during off-peak periods. This inefficiency is illustrated in Figure 1-1 by the gap between the blue line representing infrastructure’s capacity (which can be generalized to represent the number of servers in use) and the green line that is an indication of actual user demand for the service provided by the application. The gray vertical arrow illustrates the disparity between the two. This approach is obviously a costly solution due to the presence of unutilized capacity; therefore, this is generally not a recommended approach. The second approach is based on dimensioning the system for the typical usage (pattern) of the application, while suffering the consequences of lost traffic when peak demands are encountered. Although this has a lower cost in times of normal usage, it is costly during traffic spikes because the lost traffic typically leads to lost revenue opportunities. This scenario is illustrated in the Figure 1-1 by the shaded region under the demand curve between the green line (demand) and the blue line (infrastructure capacity). In this situation when the demand exceeds capacity traffic is lost and/or the application service may even become unavailable.

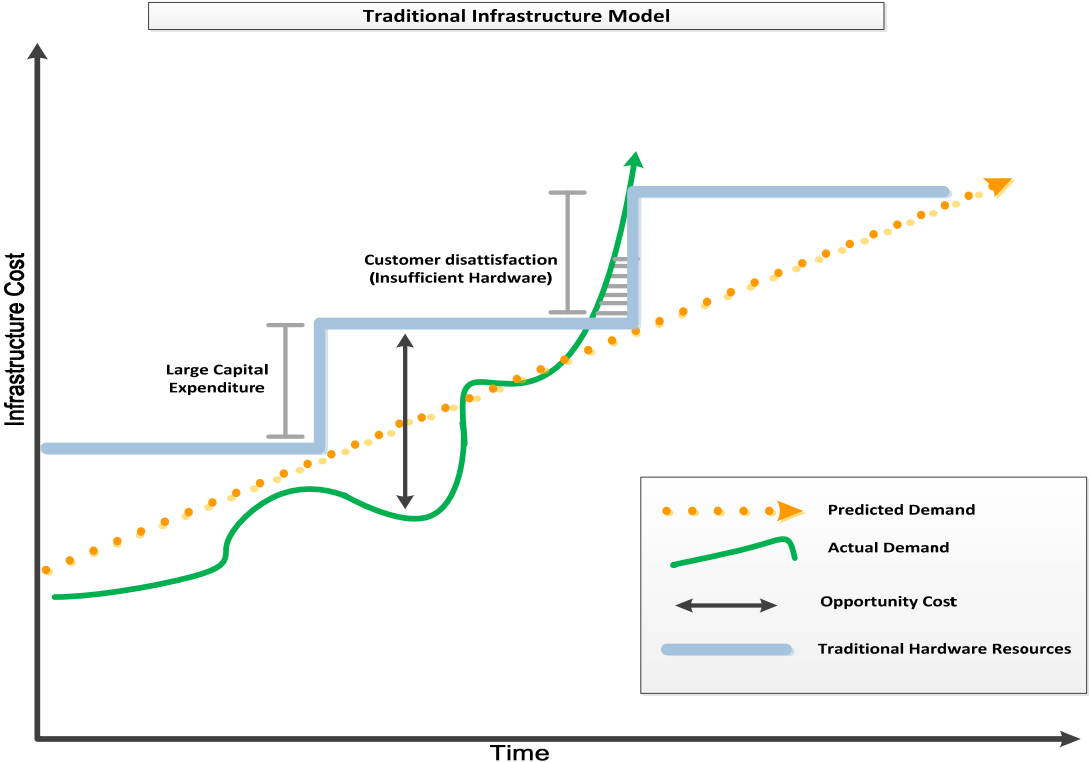


Figure 1-1: Traditional Infrastructure Model

For a dynamic and unpredictable environment neither of the above approaches with a traditional infrastructure model is desirable. This is why a scalable cloud architecture model offers an excellent fit for such dynamic and unpredictable loads. In a scalable cloud architecture model, it is possible to dynamically provision additional resources only when they are needed and then decommission them when they are no longer required. The result is a true utility computing paradigm where customers incur charges only for the time period during which they use the resources. Figure 1-2 illustrates this scalable cloud architecture model for dynamically providing application resources.

In Figure 1-2, the demand curve is identical to that of Figure 1-1, but due to the dynamic provisioning of infrastructure resources, no infrastructure resources sit idle when there is no demand for this application, nor is there insufficient capacity when it is necessary to accommodate an increased demand for the application.

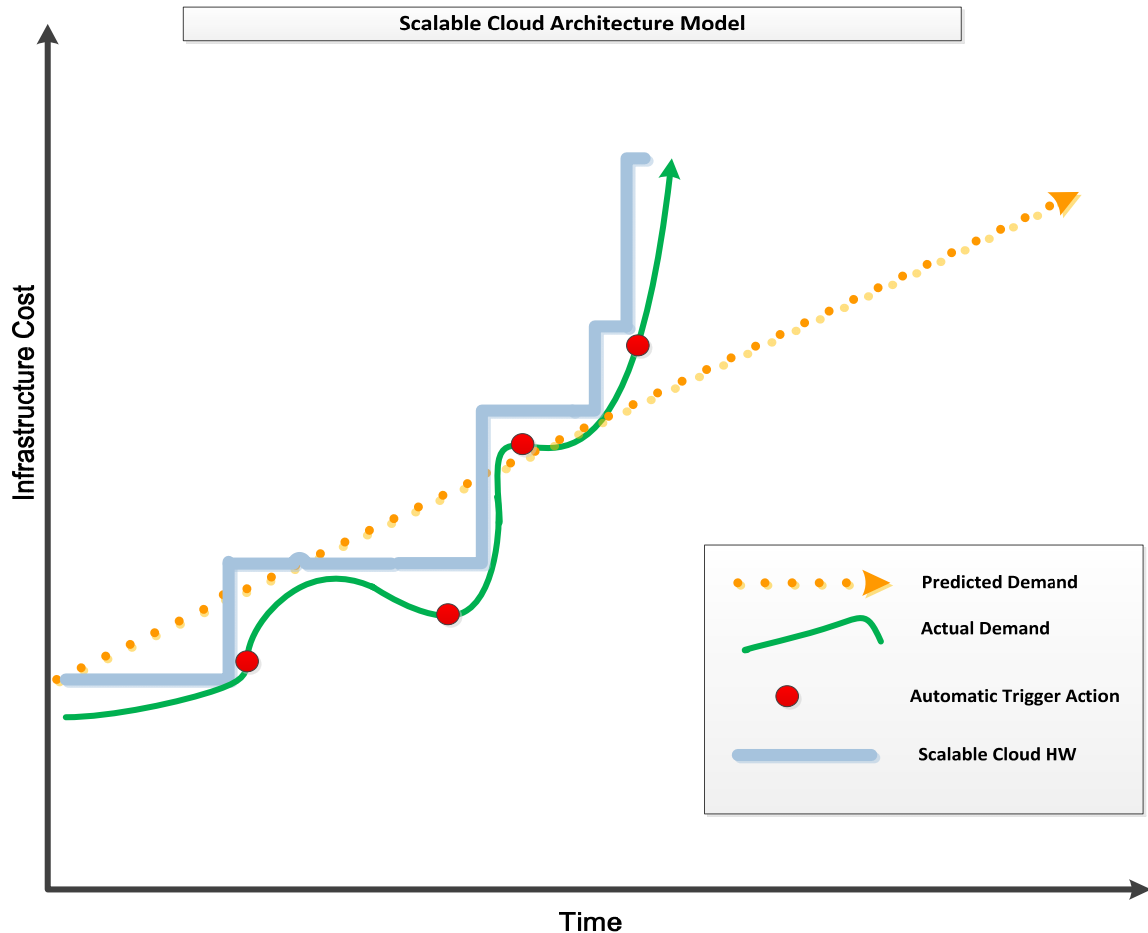


Figure 1-2: Scalable Cloud Architecture Model

In this thesis we will design, implement, demonstrate, and evaluate our proposal for a highly scalable cloud based architecture which is designed to meet the performance and rapidly evolution for a new business, such as Ifoodbag’s web-based application*. In Chapter 3, we will describe our scalable cloud architecture’s design and present our choice of preferred methods and techniques for best implementing the proposed scalable cloud architectural model at all levels of an application’s multi-tiered architecture. This thesis will clarify how to scale both up and down, since for a cloud based application which is used by people in a local area there is going to be a fluctuation of users throughout the day and there is no reason to have more or less resources than actually needed. Furthermore, we will examine how fast we can scale up or down, and what happens if we do not scale up and down rapidly enough. This will be described in terms of a control loop that determines the correct combination of virtual machines (VMs) needed to meet the expected demands for an application.

In the conclusions of the thesis, we will summarize our with respect to the gains that we could achieve though our performance analysis of our pilot setup of a scalable cloud architecture design. We identify additional mechanisms that could enable the deployment and maintenance of a scalable application in the cloud. We also suggest some future work that might build upon the results reported in this thesis.

* Ifoodbag is a Stockholm based startup offering weekly home delivery of food with personalized recipes. Further details can be found at <http://www.Ifoodbag.se/>.

1.3 Research methodology

For this thesis project we have adopted positivist philosophic assumptions and followed a design science research methodology including pragmatic approach [8], which is also known as ‘mixed methods’, as this approaches grant researchers the freedom to use any of the methods, procedures and techniques typically associated with quantitative or qualitative research methodology. This method allowed us using different data sources, multiple perspectives to interpret the results and multiple methods to study a research problem. We have followed five steps design problems as defined by Seyyed Khandani to solve design problems according to the *Engineering Design Process* [9]. The five steps are:

- I. Define the problem
- II. Gather pertinent information
- III. Generate multiple solutions
- IV. Analyze and select a solution
- V. Test and implement the solution

In the beginning we have followed quantitative or qualitative only research methods, however we have rejected this individual methods as of both are focused on very individual specific requirements and procedures, for example the objective of quantitative research is to develop and employ mathematical models, theories or hypothesis pertaining to phenomena [10]. But that was not appropriate to drive our research goals. Hence we have followed the pragmatic approach where we have had the freedom of utilizing any of these approaches whenever it was needed.

1.4 Project goals

The main goal of this thesis is to design, implementation, demonstration, and evaluation of a highly scalable cloud based architectures designed for high performance and rapid evolution for new businesses, such as Ifoodbag AB, in order to meet the requirement for their web-based application. The goal is also to examine how to scale both up and down, since for a cloud based application which is used by people in a local area there is going to be a fluctuation of users throughout the day and there is no reason to allocate more or less resources than actually needed. Additionally, this thesis examines the limitations on the rate at which this scaling may occur when using information from the running instances of the service.

1.5 Project scope

This thesis focuses on designing scalable cloud architecture model and defining scaling policies and implementing a management node to monitor and scale the application. Physical security, legal compliance, disaster recovery strategy, risk management and overall security of the architecture are **out** of the scope in this thesis project. We do **not** consider what activities the application servers (mainly what type of the services and application security itself) are supposed to perform, thus actual application implementation and its security is **out** of this thesis project. This means that we will focus on the *interaction* between these servers, virtual machines, and client web browsers via the network. As our proposed solution is implemented and proposed for cloud service provider either in private or public or hybrid cloud architecture, thus underlying infrastructure nodes (e.g. routers, switches, firewalls, servers, etc.) and defining their security is **not** focused in our thesis project.

1.6 Structure of this thesis

Chapter 2 provided the reader with the necessary background to read the rest of this thesis. Chapter 3 describes the fundamental parts of scalable cloud architectures. Based upon these parts Chapter 4 describes the details of the design that we have selected for each of these parts. Chapter 5 describes the implementation of each of these parts and our experimental setup that will be used to evaluate our implementation. The experimental results and their analysis are given in Chapter 6. The thesis concludes in Chapter 7 with some conclusions, suggestions for future work, and some reflections on the social, economic, legal, and ethical considerations of this work. Further details are given in the appendices for those who might want to build upon the work described in this thesis.

2 General Background

Cloud computing has emerged as one of the hottest topics in IT. The concept of cloud computing comes from various computing research areas, such as high performance computing, virtualization, utility computing, and grid computing. Due to the introduction of cloud computing it has never been cheaper, faster, and easier to set up a scalable, on-demand, geographically optimized web application environment. Cloud computing brings all of these features together. Cloud computing comes into focus when IT professionals think about what IT always needs: a way to increase capacity or add capabilities on the fly *without* investing in new infrastructure, training new personnel, or licensing new software. Cloud computing encompasses pay-per-use service via the Internet that extends an organization's existing capacity and capabilities. Cloud computing has its own conceptual, technical, economic, and user experience characteristics. Clear insights into cloud computing will help the development and adoption of this evolving technology by both academic and industrial users. Additional details about cloud computing and its characteristics will be given in section 2.1. The cloud model is composed of three service models and four deployment models. More details about service and deployment models will be given following sections 2.2 and 2.3.

One of the major component of cloud computing is virtualization. While virtualization technologies share a common bond by maximizing computing resources, there are differences between the virtualization technologies and cloud computing. Virtualization is the process of simulating “virtual” versions of infrastructure resources, such as computing environments, operating systems, storage devices, or network components. Cloud computing is the delivery of shared computing resources, software, or data as a service via the Internet. More details about virtualization will be given in section 2.4. The acronym “LAMP”^{*} refers to a solution stack of software, usually free and open source software, used to run dynamic web sites or servers. Details about LAMP will be discussed in section 2.5. Cloud providers offer different cloud services based on service level of abstraction. Section 2.6 gives more detail about a number of the current major cloud providers. Section 2.7 reviews related work.

2.1 What is cloud computing?

Traditionally business applications have been very complicated and expensive. The amount and variety of resources (both software and hardware) needed to run these applications caused companies to require a whole team of experts to install, configure, test, run, secure, and update these systems. Cloud computing eliminates these headaches because resources are not managed locally; but rather an experienced vendor is responsible for managing the resources[11]. According to Amazon (one of the earliest cloud service providers), the term “cloud computing” refers to the on-demand delivery of IT resources via the Internet with pay-as-you-go pricing[12].

In the last few years, the cloud-computing model has become an important concept and has been widely adopted by many companies. Different companies have their own definition of the cloud and cloud computing, but most of these definitions focus on several important attributes; such as requested resources are provided rapidly on demand, the service is scalable, and the consumer pays only for what he or she uses. These resources might be computational power, storages, networks, or applications[2]. Here we quote a few definitions of cloud computing:

^{*} Typically LAMP is realized by the combination: Linux, Apache, MySQL, PHP; however, other combinations of software can also be used to realize LAMP as will be described in section 2.5.

“Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.” - The National Institute of Standards and Technology (NIST), USA [13].

“A Cloud is a type of parallel and distributed system consisting of a collection of inter connected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreements established through negotiation between the service provider and consumers.” - R. Buyya, C.S Yeo, and S.Venugopal [14].

“A cloud is a powerful combination of cloud computing, networking, storage, management solutions, and business applications that facilitate a new generation of IT and consumer services. These services are available on demand and are delivered economically without compromising security or functionality”. – Cisco Systems, Inc. [15].

“Cloud computing is the next stage in the Internet's evolution, providing the means through which everything from computing power to computing infrastructure, applications, business processes to personal collaboration can be delivered to you as a service wherever and whenever you need”. – J. Hurwitz, R. Bloor, m. Kaufman and F. halper [16].

From the above definitions, it should be clear that cloud computing is an Internet based computing service that shares resources and provides information to the consumer on demand, much like electricity grid provides electricity on demand. The concept of cloud can be traced to grid computing and has been extend to address QoS (quality of service) and reliability issues. If there is a single point of failure in the grid, then there is a risk of failure; this is a big disadvantage of grid computing. In contrast, cloud computing avoids having a single point of failure by virtualizing grid computing in a shared environment within a common cloud. Note that the cloud can utilize resources from multiple administrative domains.

The most important cloud computing paradigm is virtualization. IT resources can be utilized more effectively by virtualizing the major resource(s); this reduces complexity for consumers - while allowing IT organizations to perform their own optimizations. Cloud computing builds upon a virtualized infrastructure consisting of computational resources, storage, and network devices[17]. The details of this virtualization will be discussed later in this chapter.

Basically, the cloud is a set of virtualized resources that are managed. There are many key characteristics, but today three different service models and four deployment models are well defined with respect to the cloud-computing mode. These will each be discussed in following sections.

2.1.1 On-demand self-service

A consumer* can provision computing resources based on their current (or near future) needs. As the consumer's needs may change with time it is important to adapt the reservation of resources to those that are appropriate. To perform tasks such as building, deploying, managing, and scheduling, a cloud computing environment should allow the user to interact with the cloud in such a way as to be able to explicit reserve and return resources. The user

* The consumer that we are referring to here is the customer of the cloud service provider, rather than an end customer.

should be able to access all the resources they needed without any interaction in advance with the cloud service provider[18]. Furthermore, the consumer should not be limited to a specific set of servers. The cloud service provider is responsible for providing sufficient resources to satisfy the consumer's needs. The user controls the reservation of resources and returning of resources, thus the consumer is responsible to avoid wasting resources (which is in their own interest as they are paying for these resources – whether they are effectively using them or not). The better the decision made regarding current and future needs, the better the service that the consumer can provide and the more cost effective this service can be.

Provisioning computing resources on demand for a large number of enterprises is one of the most desired capabilities of a cloud, because this eliminates the need for planning for future growth and avoids the loss of customers when short term traffic demands are greater than expected. Pay-per-use reduces the unnecessary upfront costs that otherwise an enterprise would have to make to purchase and install resources which would need to meet or exceed the expected demand. Unlike the traditional model, cloud computing helps the consumer avoid the costs of underused resources[19].

2.1.2 Ubiquitous network access

Accessing the computing and storage capacity of a cloud should enable access through standard Internet enabled devices. Cloud computing is device independent, because the computing resources can be accessed by heterogeneous thin or thick client platforms, in fact any authorized platform that has an internet connection and a web browser (or a specific application). It really does not matter that what kind of devices are used to access resources, be they smartphones, tablets, laptops, or workstations.

2.1.3 Elasticity and scalability

The computing resource allocations can increase or decrease according to the consumer's demand. This change in resources is called *elasticity*. Elasticity enables scalability; hence a cloud should be able to scale resources (by increasing or decreasing) as necessary. Scalability also implies that an application can be scaled up due to additional users or when the application's requirements change[20]. If on a particular day the demand varies over time, the system should be scaled up or down in resources to meet the actual demand.

For example, imagine a cloud based website that averages 1,000 hits per day. Suddenly, on one particular date the website launches a special offer. In this case there is a higher probability that a larger number of users will access the site at nearly the same time. For example, due to this special offer the number of access to this website might rise to 10,000 on a particular day. In this scenario, we assume that during a normal day the cloud would assign one server, but during the peak hours on this particular day the service might be instantiated on five different servers and later return to running on a single server during non-peak hours. If we are hosting this service ourselves, we would need to purchase five servers in order to prepare for the load during the peak hours, but outside of these peak hours four of our servers will be idle (hence wasting resources).

2.1.4 Horizontal and Vertical Scalability

A consumer can scale the set of resources which they reserve either horizontally (also called scaling out) or vertically (also called scaling up) in order to match the application's performance to meet increasing or decreasing demands upon the consumer's application. Horizontal scaling (scaling out) requires adding or removing cloud servers, specifically VMs or devices to handle an increased or decreased application work load. Vertical scaling (scaling up) requires replacing a single cloud server by a more powerful server (where this power is quantified in terms of virtual CPU performance, available RAM, available disk capacity, etc.)

in order to handle increased or decreased demand. This is represented in the transition from Figure 2-1 to Figure 2-2.



Figure 2-1: Basic single N-tier Architecture (Adapted from Figure 1, page 9 of [21])



Figure 2-2: Horizontally scaled load balancing and web-tier and vertically scaled database tier (Adapted from Figure 2, page 9 of [21])

In vertical scaling there are additional CPU cycles available, so if the task simply requires additional processing power then vertical scaling may suffice. Additionally, in some cases scaling up may also increase I/O bandwidth. When businesses experience gradual increases in traffic, scaling up provides adding extra resources to support additional demand until the load exceeds the newly provisioned resources[21]. Conversely, horizontal scaling can handle sustained increases in demand as horizontal scaling scales CPU power, memory, and I/O (both disk and network bandwidth). However, horizontal scaling requires load balancing to spread the load over the separate instances. Additionally, to avoid idle resources it is very useful to utilize resources from a pool of resources, so that unneeded resources can be returned to the pool and used by others (this is the scaling advantage that large cloud service providers offer). Further details of resource pooling are described in the next subsection.

2.1.5 Resource pooling

Cloud providers typically allocate their resources in order to serve multiple consumers using a multi-tenant model. This means that different physical and virtual resources are pooled then assigned to specific consumers for their use based upon dynamic assignments according to their customers’ demands. These resources are generally location independent, thus the consumer generally does not know the location of resources, however, it is possible in many cases for the consumer to specify the location at a higher level of abstraction (e.g., country, state, or data center)[13]. Providers dynamically allocate their resources to different consumers and these allocations change over time based on their consumers’ demands. These changes in allocation should be transparent to the consumer, as it is the cloud provider’s responsibility to ensure that one consumer cannot access the data of other customers. The cloud provider must also address other security issues. For example, the cloud provider does not permanently assign a particular resources to a specific individual customer, but rather dynamically assigns resources based upon their consumers’ demands[22]. An additional

advantage of resource pooling is that it allows consumers for the adding and removing resources. Another advantage is that resource pooling can facilitate increased reliability. In fact, Damon Wischik, Mark Handley, and Marcelo Bagnulo Braun in their article “The Resource Pooling Principle” [23] make two observations:

- 1 “Resource pooling is often the only practical way to achieve resilience at acceptable cost.”
- 2 “Resource pooling is also a cost-effective way to achieve flexibility and high utilization.”

2.1.6 Pay-per-use

Without making an upfront investment, the consumer pays the cloud provider as with other utility based subscriptions, such as paying for electricity. Consumers are charged fees based on the amount of resources they actually use. The pay-per-use model helps the user to keep track their usage and ultimately helps them to reduce their costs. Cloud providers keep track of their customers’ usage information enabling them to charge their customers, generate reports, and invoice their customers[24]. The information gathered should be readily available to the customer. This information is necessary to enable the customer to realize the cost benefits that cloud computing brings. This pay-per-use underlies the concept of cloud computing and is closely related to utility computing.

2.1.7 Self-managed platform

In order to provide an efficient cloud service, the cloud provider must have a technology platform that is self-managed. Software automation can be used to make a cloud self-managing. By leveraging some capabilities of this software the cloud provider can realize a best-of-breed cloud. The cloud platform is able to deploy services and tearing them down to recovering resources through a provisioning engine. This provisioning engine platform has a mechanism for scheduling and reserving resources. The platform may also have capabilities for configuring, managing, and reporting to ensure that resources can be allocated and reallocated to different consumers as the consumers’ demands change. There tools control access to resources and enforce policies concerning how resources can be used or what specific operations can be performed by each party[24].

All of these abilities enable business agility and also reduce necessary administration. A self-managed platform minimizes the amount of IT administrative effort and reduces the cloud provider’s operating expenses.

2.1.8 Standardized interfaces

An essential issue is how applications and data sources communicate with each other. In the case of cloud services standardized application programming interfaces (APIs) can be used to solve this problem. A standardized interface also enables a consumer to integrate different cloud services together[20]. Today there are a number of the APIs, for details the reader should refer to [25, 26].

2.1.9 Quality of Service (QoS)

Providing support for Quality of service (QoS) requires the ability to provide different levels of service to different applications, users, or data flows. When we speak of QoS other than best effort, we generally refer to a guarantee of a certain level of performance, availability, security, and dependability being made by some provider[27]. QoS has been an issue in many distributed computing paradigms, such as grid computing and high performance computing. Cloud computing must also assure the desired service level for users. The cloud provider should ensure that their guarantees on round-the-clock availability, adequate

resources, performance, and bandwidth are met as agreed to in the service-level agreement of their service (to which they and their customers agree). Any compromise in these guarantees could prove fatal for the cloud provider’s customers[18].

2.1.10 Reliability

Cloud provider should have able to provide their customers with reliable service, i.e., with a committing uptime for their service. In today’s public clouds, reliability is specified as a fixed service parameter, e.g., Amazon published that its EC2 users can expect 99.95% uptime in terms of reliability, which corresponds to a once-a-week failure rate[28].

2.2 Cloud computing service models

Cloud providers offer cloud services, which give their users more or less control over the resourced provided by the provider’s cloud depending upon the type of cloud service. When customers choose a cloud provider, they should compare their needs to the cloud services available from each provider. The cloud service type and optimal choice of cloud provider will vary with the type of customer (e.g., personal home use, business). Customer should keep in mind that their cloud provider will be charging them on a pay as you go basis, which means that the customer can rent new resources or release existing resources according to their needs at any point of time[29]. However, the customer may be charged a minimum cost for changes in resource allocation and these changes do not occur instantaneously.

There are three types of service models that are widely used in cloud computing, the user can choose or subscribe to: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), or Software as a Service (SaaS). These three different types of service models are provide by many cloud providers. These different service models differ in the amount of control which the user has versus the control which the cloud provider has. Figure 2-3 shows these differences in the number of parts of the complete service stack that a customer of an IaaS, PaaS, and SaaS cloud has. This should be compared to the control of a private on-premises server. The first stack on the left is an on-premises environment where user must take care of everything from the networking all the way up to applications. This is the traditional IT infrastructure business model that many businesses use today. The following subsection will described the other types of service models.

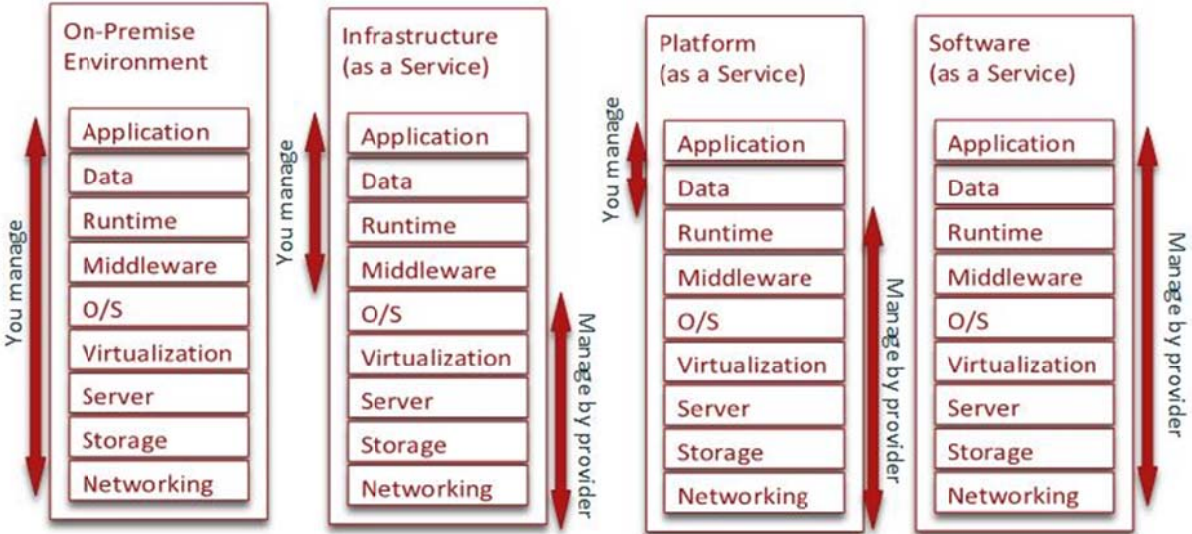


Figure 2-3: Server stack comparison between on-premise infrastructure, IaaS, PaaS, and SaaS (Adapted from Wely Lau’s online article[22])

2.2.1 Infrastructure as a Service (IaaS)

In the case of IaaS a cloud supplier provides an online infrastructure on which their customers can store data and develop and run whatever applications they want. IaaS offers virtualized resources (e.g., computation, storage, and communication) on demand [30]. IaaS helps the user by taking care of some of the components, starting from networking to provisioning the OS (as shown in Figure 2-3). However, users are responsible for middleware, runtime, data, and applications levels. Users basically rent a virtual machine (VM) with their preferred OS installed. The provider generally does not care what users do with this VM[22].

A fundamental building block of a cloud computing infrastructure is a server. Cloud computing servers are used to deploy VMs on which applications can be run. A cloud provider also provides various forms of data storage. Users are given privileges to perform certain activities on the server, such as: starting and stopping a VM, configuring access permission, etc. [31]. Examples of IaaS providers include Amazon, Go Grid, and Eucalyptus.

2.2.2 Platform as a Service (PaaS)

PaaS provides a toolkit and a number of supported programming languages to enable the cloud provider's customers to build their own application and deploy this application in the provider's cloud infrastructure. The users of PaaS are typically developers who develop their applications on the platform and provide their applications to their own end users[32]. PaaS is one level up in abstraction from IaaS, as the cloud provider manages the platform-level components (such as middleware and runtime), as shown in Figure 2-3. The cloud customer does not manage or control the underlying cloud infrastructure, but has control over the deployed application and possibly can choose their preferred configuration settings for the application-hosting environment. Some examples of PaaS providers are Google App Engine and Microsoft Windows Azure Platform.

2.2.3 Software as a Service (SaaS)

In SaaS computer applications are accessed over the Internet, rather than being installed on a local computing device or in a local data center. SaaS is the most common cloud service that end users may have used. The cloud provider takes responsibility for the entire stack from the network and server to the application level, as shown in Figure 2-3. The cloud customers are not allowed to access the underlying infrastructure or platform; rather they can only change the application's user settings. These applications are normally accessible through a thin client interface, such as web browser. Today end users are rapidly shifting from locally installed programs to online software services that offer same functionality[22].

SaaS can provide the general cloud computing advantages of dynamic scalability. Additionally, SaaS is generally end user device independent[32]. A great advantage of SaaS for an application provider is that there is frequently no upfront hardware cost in deploying an application via SaaS. This means that SaaS applications can be up and running quickly at a low cost. Many SaaS applications are also collaborative, in that they allow multiple users to share documents and even to work on these shared documents at the same time. The most common examples of SaaS applications are Gmail, Office 365, and Google Docs.

Figure 2-4 shows a variety of access methods and management tools which a user will use to access and configure their services. The figure also shows the type of content that a particular service offers.

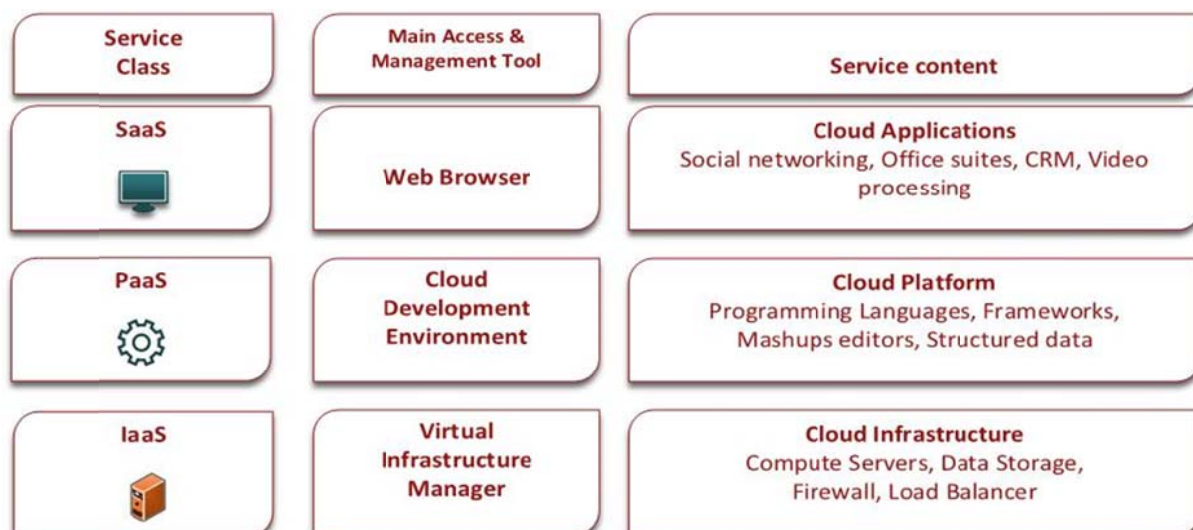


Figure 2-4: Cloud computing stack (Adapted from Figure 1.3, page 14 of [31])

2.3 Cloud computing deployment models

Although cloud computing has emerged mainly due to the appearance of public computing utilities, the different infrastructure deployment models are distinguishing by their architecture, variations in physical location, and location of the data center (and their geographical distribution). Regardless of cloud service type, a cloud can be classified as public, private, community, or hybrid based on its deployment model. These different deployment models will be describe in following subsections.

2.3.1 Public cloud

Any subscriber can access a public cloud via the internet. The cloud provider owns the physical infrastructure of a public cloud. Armbrust et al. [33] define a public cloud as a “cloud made available in a pay-as-you-go manner to the general public”. Some customers are reluctant to choose a public cloud due to privacy, policy, and security concerns when their application operates on sensitive data. Examples of public cloud services include: Microsoft’s Windows Azure Platform, Amazon’s AWS, and Google’s AppEngine.

2.3.2 Private cloud

Armbrust et al. [34] define a private cloud as an “internal data center of a business or other organization, not made available to the general public.” A private cloud is built for a specific group or organization and access is limited to that group or organization. One specific customer owns and fully controls the private cloud. Although a private cloud might be owned by the customer, these private clouds are frequently built, installed, and managed by a third party rather than the customer. A private cloud is less cost effective than a public cloud, but may be more suitable when an application must process sensitive data.

Another way to build a private cloud is to create a virtual private cloud. To build a virtual private cloud, a cloud provider allocates particular resources within their public cloud infrastructure to this particular virtual private cloud. Due to the allocation of specific resources within the cloud the customer can be assured that their data is stored on and processing only on dedicated servers and that these servers are not shared with any other customer of the cloud provider.

2.3.3 Community cloud

A community cloud is shared among two or more organizations that have similar cloud requirements (e.g., mission, security requirements, and policy and compliance considerations). Customers might agree to share configuration and cloud management. The management of this community cloud might be done by themselves or by a third party.

2.3.4 Hybrid cloud

A hybrid cloud is a composition of two or more distinct cloud infrastructures (public, private, and community) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds)[35]. A hybrid cloud can also be defined as multiple cloud systems that are connected in such a way that programs and data can be moved easily from one cloud to another cloud. Customers might utilize this approach because they want to exploit the scalability and cost-competitiveness capabilities that a public cloud provider offers, but they also want to keep their sensitive data on their own premises or in a private cloud. The hybrid cloud model combines the benefits from both deployment models, hence this solution has become increasingly popular[22].

2.4 Virtualization

Virtualization is not a new concept in the computing industry; it is actually an old practice, as it has been used since 1960. The original idea was to help maximize the power and potential of mainframe computers. In cloud computing, virtualization involves the creation of virtual resources on top of a set of underlying physical resources. There are different definitions for virtualization. One such definition is:

“Virtualization is the abstraction of IT resources that masks the physical nature and boundaries of those resources from resource users. An IT resource can be a server, a client, storage, networks, applications or OSs. Essentially, any IT building block can potentially be abstracted from resource users.”-Gartner, Inc. [36].

We can describe virtualization as the process of instantiating virtual versions of infrastructure resources. For example, a physical server or host consists of some resources (OS, memory, storage, etc.). All or part of these resources can be allocated to virtual machines (VMs) that run in a container provided by that host. Virtualization enables multiple instances of infrastructure resources to run on the same hardware that is controlled and managed by a hypervisor[17]. Figure 2-5 shows the basic architecture of this approach to virtualization.

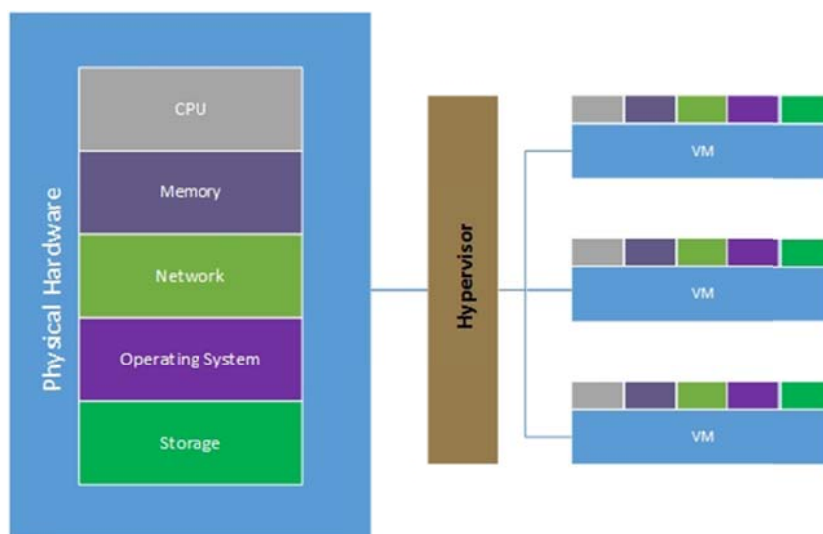


Figure 2-5: Basic architecture of virtualization [17]

A hypervisor, also called a virtual machine monitor (VMM), is a software platform that allows multiple operating systems to run on a physical host. It controls access to the host's physical hardware and creates and runs the VMs. Hypervisors can be categorized into one of two primary categories: bare metal (also called native) and hosted. A bare metal or native hypervisor runs directly on the host's hardware. VMware ESXi, Microsoft Hyper-V, KVM, and Citrix XenServer are examples of bare metal/native hypervisors. In contrast, a hosted hypervisor runs on top of the host's operating system. VMware Workstation, VMware Fusion, Microsoft's Virtual PC, and Oracle's VirtualBox are examples of hosted hypervisors. Figure 2-6 shows the architectural design of these two different categories of hypervisors.

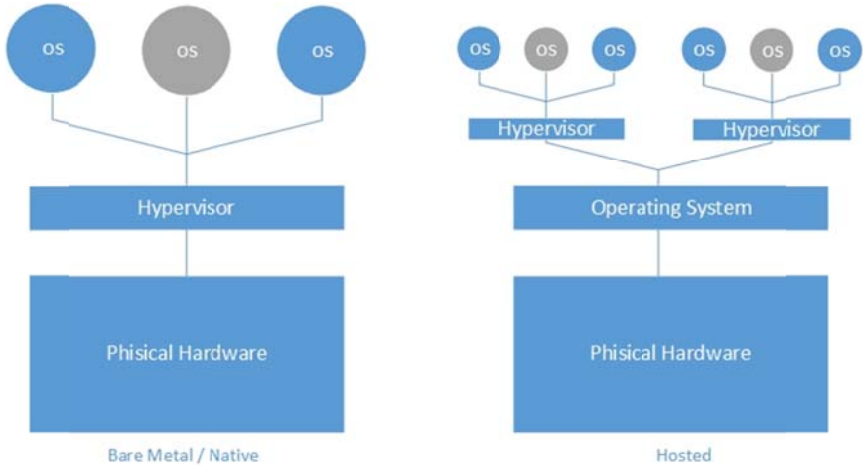


Figure 2-6: Bare metal/native and hosted hypervisor [17]

People are often confused about how virtualization and cloud computing differ. Virtualization is the ability to run multiple operating systems on a single physical system and share the underlying hardware resources. Cloud computing is the provisioning of services in an on-demand manner, to allow scaling up and down of resources for a customer. Virtualization is an *enabler* for cloud computing, because virtualization provides a straightforward means for a cloud provider to increase capacity or add capabilities for a customer *without* the customer needing to invest in new infrastructure. Cloud computing removes the dependence of a customer on specific hardware and software through virtualization[30]. Today the term virtualization is widely applied to a number of concepts including server/hardware virtualization, operating system virtualization, application virtualization, network virtualization, and storage virtualization. In this thesis, we will use server/hardware virtualization and kernel-based virtual machines (KVMs). Each of these will be described in a following subsection.

2.4.1 Server / hardware virtualization

Virtualizing the hardware/server is perhaps the most common type of virtualization used for hosting customers. One physical machine is divided into many VMs. Virtualization is based upon abstraction, hence hardware virtualization is accomplished by abstracting the physical hardware layer by use of a hypervisor[37]. The hypervisor shares the physical resources of the hardware between the different guest operating systems (OSs) running in VMs on the underlying host. Figure 2-7 illustrates the case where the hypervisor is running directly on the underlying hardware. Each VM running on the hypervisor runs as a VM and exploits the hardware abstraction to run a guest OS. The functionality of the hypervisor varies with architecture and implementation[38].

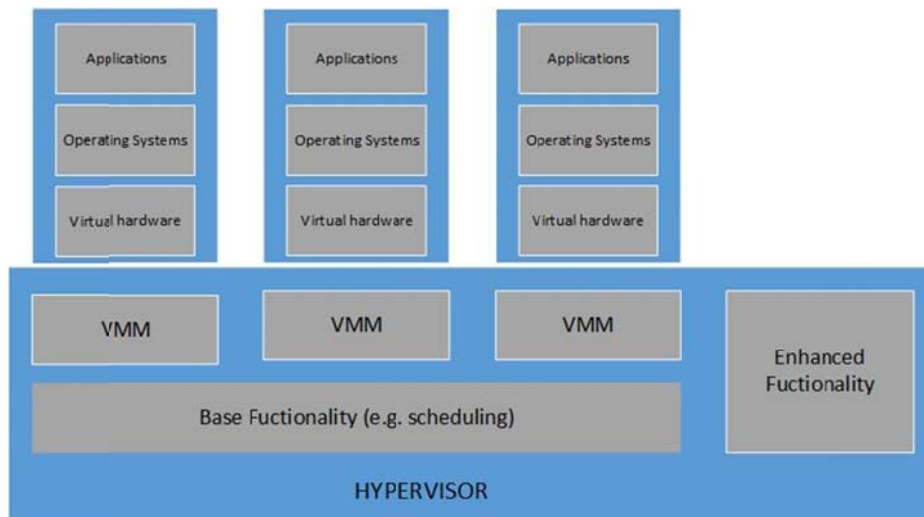


Figure 2-7: The hypervisor manages VMMs that host virtual machines [38].

Based on different levels of abstraction, there are different levels of hardware or server virtualization. These different levels of virtualization are:

Full The guest OS does not have any knowledge that it is running on a VM. The hypervisor handles all the operations between the guest OS and the underlying hardware. Note that the hypervisor may cache results of physical operations on the hardware for future use.

Hardware assisted Hardware assisted virtualization is an alternative approach. In recent years, hardware vendors have added virtualization support to their processors in order to enhance the performance and functionality of this processor for running VMs. For this reason, specific CPU instructions can be sent to the CPU *without* being translated by the hypervisor. This reduces the load on the hypervisor and increases the performance. Intel's VT-x and AMD's AMD-v are the examples of processors with virtualization support [19].

Para-virtualized In para-virtualization the guest OS is recompiled prior to installation inside a virtual machine. The modified version of the OS needs to know that it is virtualized in order to take advantage of the functions provided by the hypervisor. OSs require extensions to make appropriate calls to the hypervisor.

2.4.2 Kernel based virtual machine (KVM)

Kernel-based Virtual Machine (KVM) is a full virtualization solution for Linux on x86 hardware which implements virtualization extensions (specifically by exploiting Intel VT or AMD-V)[39]. KVM is an example of such a hypervisor and it has been developed as part of the Linux kernel. KVM is included with a variety of Linux based OSs. The main requirement for KVM hypervisors is QEMU (a generic and open source machine emulator and virtualizer). No matter what Linux distribution you want to use, you can run the KVM hypervisor. The KVM hypervisor delivers a secure, robust virtualization platform with unmatched performance and scalability for guests running Linux or Microsoft's Windows OSs [40]. In addition, the following hardware requirements need to be met [41]:

- Within a single cluster, the hosts must be running the same distribution (version) of KVM.
- All hosts within a cluster must be homogenous, this means that the CPUs must be of the same type, count, and have the same feature flags.

- The processors must all be either Intel-VT or AMD-V enabled.
- A 64-bit CPU and x86 processor is recommended, but not required.

2.5 Lamp stack – Overview

The acronym LAMP refers to a stack of software that is widely used to build general purpose web servers. This software is generally free and open source software which combines some of the principle components (OS, web server, database server, and scripting language). LAMP is an acronym which original stood for:

Linux	an operating system
Apache HTTP server	a web server
MySQL	a database management system or database server (PostgreSQL can also be used as database server)
PHP	a scripting language (other scripting languages such as Python, Perl, and Ruby can be used)

The combination of these technologies is widely used to realize a web server infrastructure. Today many different stacks including LAMP are designed to augment a basic HTTP web server. Some of the most popular available web server stacks are:

- LAMP stack,
- Tomcat Java-based stack,
- Full Java Enterprise Edition (JEE) stack,
- the WISA stack: Windows (operating system), Internet Information Services (web server), Microsoft SQL Server (database), and ASP (scripting language), and
- Full .NET stack.

Although LAMP has not had the same amount of commercial promotion that J2EE and .NET have had, the LAMP stack is used by more than two-thirds of the scripting languages, databases, and servers on the web today. The main attraction of the LAMP stack for developers around the world is that it is free, easily configured, easily deployed, fast, highly scalable, and very robust. LAMP allows developers to achieve high performance *without* requiring that the developer spending a disproportionate amount of time on administrative details.

Figure 2-8 illustrates the very straightforward architecture of the LAMP stack. Linux forwards HTTP connections to the Apache HTTP server, which serves static content directly from the Linux kernel. Apache forwards dynamic page requests to PHP and is responsible for executing the PHP code. Database queries are sent to MySQL through PHP[42].

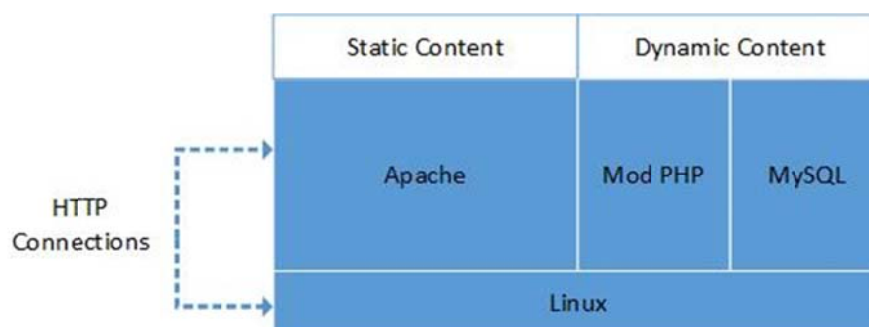


Figure 2-8: LAMP architecture (adapted from [35])

2.5.1 Linux

Linux is the most important component of the LAMP stack installed on servers. Linux provides a robust operating system, which provides the underlying security and platform for the web server. Dozens of different Linux distribution are available today, but choosing the

right distribution is a bit complicated. Linux is the OS running more than 60% of web servers on the internet because Linux based servers provide excellent performance, security, scalability, availability, and there is an audited industry performance benchmark[43].

2.5.2 Apache

The second component of the LAMP stack is an Apache HTTP server. This HTTP server played a significant role in the initial growth of the World Wide Web. Apache is an open source tool created in the early 1990s. Apache is used by more than 60% of the web servers worldwide. The web server accepts requests for content from browsers, interprets and executes the request, and returns a result to the browser. When a browser requests a static page, the web server simply retrieves that HTML file and returns the results. In response to dynamic page requests from a browser, the web browser transfers control to a program or module running at the HTTP server that interprets the request and returns a results[43].

2.5.3 MySQL

The third component of the LAMP stack is a MySQL database. MySQL is an open source tool which can be used to store content and configuration information for web applications. MySQL is a general purpose database. MySQL in particular and databases in general, have made it possible to build and present fully dynamic websites [43]. MySQL is a relational database management system. MySQL is frequently chosen by developers because it provides speedy website loading, reliability, and ease of use. The MySQL database architecture is capable of effectively scaling out by adding multiple replicated database servers. This can be done at low cost and as needed. Today many of the largest and fastest growing websites in the world employ MySQL, including Facebook, Google, Yahoo, Flickr, etc.[43].

2.5.4 PHP

PHP was originally an acronym for “Personal Home Page”. It was introduced in 1994 as a set of Common Gateway Interface binaries programs written in the C programming language. Today, PHP is a widely used general purpose scripting language that is especially well suited to web development and can be easily embedded into HTML[43]. A web server takes PHP code as input, then executes it and creates a webpage as output. PHP is another integral component of the LAMP stack and can be found in a wide range of applications ranging from personal homepages to content management systems, such as Joomla[44].

Since PHP has a relatively simple syntax and is available with open source licensing, developers around the world are migrated from more difficult scripting language such as Perl. Full object oriented syntax support is included in the latest version of PHP, along with a command line capability for quick testing. PHP’s speed and adaptability play a key role in its increased use by enterprises.

2.6 Current cloud service providers

The service level of abstraction differs between the different cloud providers. The management level of resources also varies by cloud provider. This section presents a few of the most common provider details and describes the services that they offer to their customers.

<i>Amazon Web Services (AWS)</i>	AWS is a bundled remote computing service that provides a cloud computing infrastructure over the Internet. Amazon Inc. launched AWS in 2006 [45]. Amazon packages AWS with scalable and virtually unlimited computing, storage, and bandwidth resources. AWS uses a subscription-pricing model of pay-as-you-go or pay-for-what-you-use. The customer can avoid up-front capital infrastructure expenses by substituting low variable costs that scale as their needs change. AWS provides a flexible, cost-effective, scalable, and easy-to-use cloud computing platform that is suitable for research, educational use, individual use, and for organizations of all sizes[46]. Amazon's EC2 and Amazon S3 are two core IaaS services. These two services have been used by cloud application solution developers worldwide.
<i>Eucalyptus</i>	Since the Eucalyptus infrastructure is compatible with AWS (in either a private or hybrid cloud), the allocated resources can be dynamically scaled up or down as application workloads change. Eucalyptus Systems has announced compatibility with AWS Elastic Load Balancing (ELB), Auto Scaling, and CloudWatch in their release 3.3.
<i>Salesforce</i>	Salesforce is one of the pioneering cloud computing providers. Their Customer Relationship Management (CRM) web service is their first and main product. Enterprise customers build their own application(s) on top of Salesforce's CRM. Initially Salesforce only offered a SaaS class product. One of the traditional issues with SaaS products is the limited ability to customize the application. However, Salesforce.com is offering force.com as a PaaS product. The force.com platform allows developers to develop applications that will execute natively on their Salesforce platform or they can be integrated with third party services. Force.com development is performed using nonstandard, purpose-built tools and a proprietary development language called Apex[47]. Scaling the platform up and down as needed and making all the physical resources transparent is the responsibility of Salesforce.
<i>OpenNebula</i>	OpenNebula is the most feature-rich, innovative, customizable and mature open alternative to proprietary cloud solutions when building virtualized enterprise data centers and cloud infrastructures on top of Xen, KVM, and VMware deployments[48]. OpenNebula is a fully open source toolkit to build IaaS private, public, and hybrid clouds. OpenNebula can be installed and run on the majority of the Linux distributions and it is also Amazon EC2 compatible. OpenNebula is primarily used as a virtualization tool to manage a virtualized infrastructure in a data center or cluster (typically within a private cloud). OpenNebula also supports a hybrid cloud to combine a local infrastructure with a public cloud-based infrastructure, enabling highly scalable hosting environments. OpenNebula also supports public clouds by providing cloud interfaces to expose its functionality for VMs, storage, and network management. OpenNebula also can work as a data center virtualization manager within an OpenStack or Eucalyptus cloud.

2.7 Related work

Red Hat Inc. published “Scaling the LAMP stack in a Red Hat enterprise virtualization environment”[43] during 2009. This work presents performance and scaling of the industry standard LAMP web application stack running on Red Hat enterprise Linux 5.4 guests on a Red Hat Linux 5.4 host with a KVM hypervisor.

Kaur, Kaur, and Singh published “Evaluating performance of web services in cloud computing environment with high availability” [49] in 2012. This paper presents a methodology for attaining high availability to meet the demands of web clients. In order to improve the response time of web services during a peak hour, dynamic allocation of host nodes was used. Web users can be very demanding, as they expect web services to be quickly accessible from anywhere in the world at all times.

Vaquero, Rodero-Merino, and Buyya. published “Dynamically Scaling Applications in the Cloud” in January 2011. This work presents the most notable initiatives towards whole application scalability in cloud environments[1].

Joyent Inc. published a whitepaper entitled “Performance and Scale in Cloud Computing” [21]. Joyent’s Smart Technologies address many issues of scalability and performance in cloud computing, including dynamic vertical scalability, more efficient allocation of virtual resources, and efficient I/O load balancing.

Aleksandar Draganov published a master’s thesis entitled “Exploiting Private and Hybrid Clouds for Compute Intensive Web Applications” [2] in 2011. This work investigates the use of an open source cloud management platform (OpenNebula) to create a private cloud and using OpenNebula for hosting compute intensive web application by managing a farm of virtual web servers to meet the application’s demands.

Chieu, Mahindra, Karve and Segal published “Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment” [50] in 2009. This paper presents a novel architecture for the dynamic scaling of web applications based on thresholds in a virtualized Cloud Computing environment.

Hung, Hu and Ching Li published “Auto Scaling Model for Cloud Computing System” [51] in 2012. This paper presents an auto-scaling algorithm for automated provisioning and balancing of virtual machine resources based on active application sessions as well as the energy cost is considered in the proposed algorithm.

Wolke and Meixner published “TwoSpot: A Cloud Platform for Scaling Out Web Applications Dynamically” [52] in 2010. This paper presents a methodology for combining existing and open technologies to build new software platform, which runs on virtual machines typically offered by IaaS provider.

Zsolt Siklosi published a master thesis entitled “Dynamically Scalable Applications in Cloud Environment” [53] in 2013. This thesis work focused on automatically scaled infrastructure and also ensures that the amount of reserved resources is always sufficient to keep up a certain service level while optimizing costs by avoiding over-provisioning.

3 Scalable Cloud Architecture for a Web Application

This chapter presents the main architecture of a web-based application designed for running in clouds. Each of the sections describes the different tiers and the management nodes of the architecture in detail.

3.1 Scalable Web Application Reference Architecture

For Ifoodbag[54], we have designed a scalable cloud architecture for use with their web application. In this section we will describe this reference architecture and outline the distinct tiers of this model, as well as demonstrate the optimized functionalities provided by each of these tiers.

Figure 3-1 illustrates the reference architecture model for Ifoodbag’s web-application. This architecture looks much like the classic three-tier web application architecture with the addition of a caching tier between the application servers and the database. In addition, each tier incorporates various enhancements to provide high performance [55, 56]. For example, the Squid web cache daemon will enhance the performance of the web service and APC (a PHP caching tool) will enhance the application server performance.

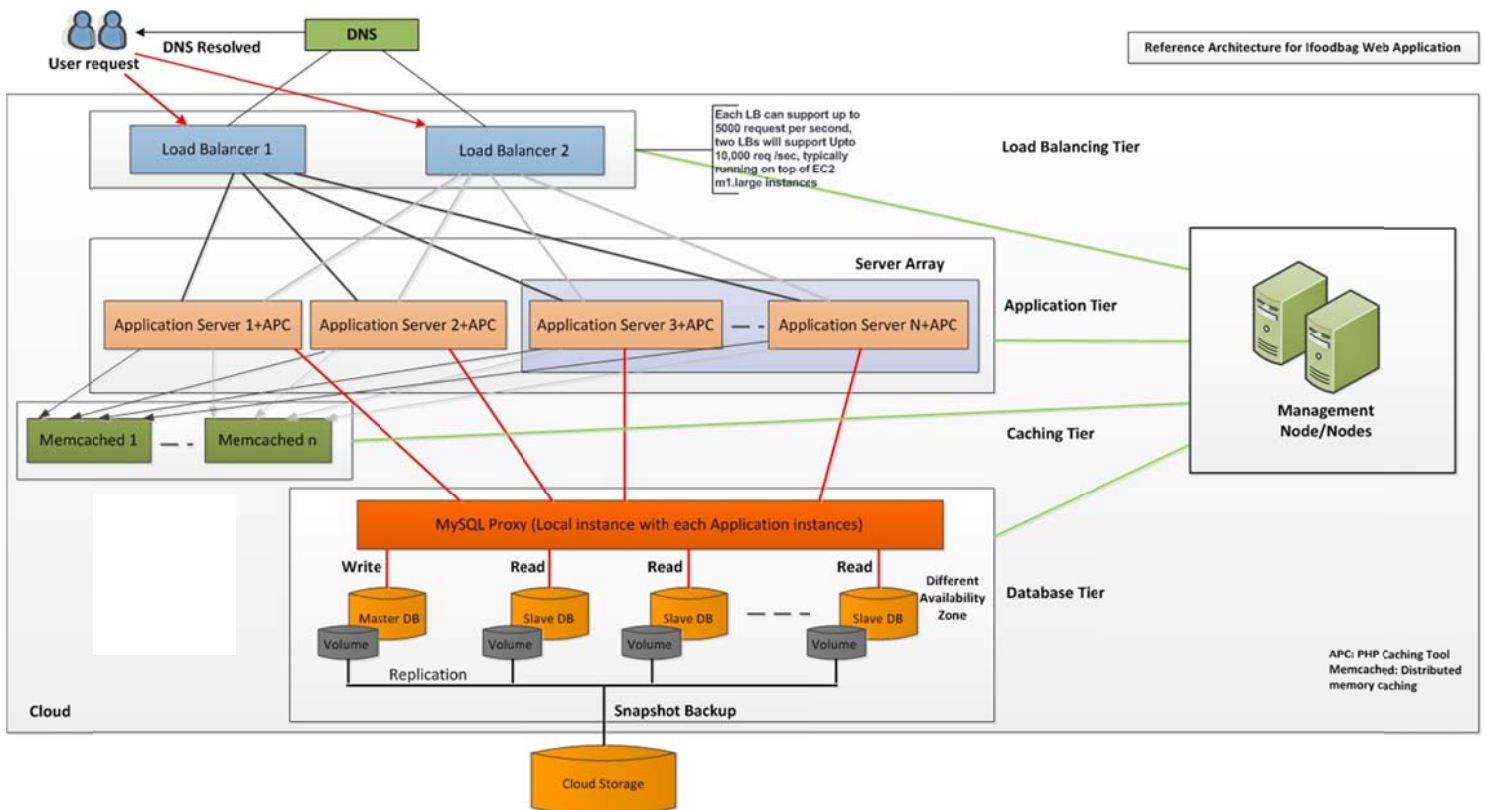


Figure 3-1: Scalable reference architecture for Ifoodbag’s web-application

3.2 Load Balancing Tier

In the reference architecture the first (load balancing) tier (as shown in the Figure 3-1) we utilize two load balancers, for example running HA proxy [57]. These load balancers (LBs) are usually run on top of Amazon’s Web Services (AWS) Elastic Compute Cloud (EC2)[58] m1.large instance types, which provide 2 virtual cores, 7.5GB of memory, and a 64-bit platform[59]. Each LB has the capacity to handle approximately 5,000 requests per second, thus two LBs support a total of about 10,000 requests per second. It is possible to estimate a new website’s highest expected traffic rate in terms of requests per second, then divide this rate

by 5,000 to estimate the number of load balancers that will be required to handle this traffic load.

Regardless of the estimated load, we have designed to LB tier to have at least two LBs in order to provide redundancy in the early phase of the deployment of *Ifoodbag's* application. For high reliability and availability we recommend placing each LB in a different availability zone (thus decoupling their probability of failure). As the number of users is expected to rapidly increase in the future, it will be possible to introduce more LBs in order to handle the required capacity. Initially it is possible to run each load balancers on an AWS m1.small instances (1 virtual core, 1.7GB memory, 32-bit platform). This is a cost-cutting measure, but allows the LB to subsequently be migrated to a larger instance as demand increases. In the early phases of an application's lifecycle it is also possible to combine the front-end LBs and the application servers in the same instance in order to achieve additional cost savings. However, this is not a recommended approach for a production high traffic site!

3.3 Application Tier

Second tier of the reference architecture shown in the Figure 3-1 is called the application tier. This tier consists of the application servers and their associated scalable server array. In this tier we recommend a minimum of two application servers (to be placed in different availability zones) for the initial configuration. These instances must implement alert mechanisms to allow automatic scaling (both up and down) of the array based on instance specific metrics. The most common metrics that can be used for auto-scaling are CPU-idle fraction, amount of free memory, and system load. It is also possible to include application specific metrics for controlling this auto-scaling. If any of the thresholds specified by these metrics are met, then an alert associated with this metric is triggered, and this should result in scaling up by adding additional application servers in the case of increased demand or decommission of active servers if the load decreases.

For guaranteed service operation, we recommend a conservative approach for scaling up and down the application server arrays. Therefore additional instances should launch *before* they are needed when an upward trend in demand is detected. It is important to determine the amount of time required for a server to become operational after it is launched as we must factor this into our scaling metrics. (Measurements of this will be reported later in Chapter 6.) Similarly, we should only decommission instances when they have been lightly utilized for a predetermined period of time. Scaling up liberally (that is, as soon as lower thresholds are met) helps to ensure that resources are continually available to serve application requests, while scaling down conservatively prevents terminating application server instances prematurely, thus avoiding undesirable user experiences. The only disadvantage of this is that if too liberal an approach is used when scaling up; the business will be charged for additional server time that was actually unnecessary. In utility computing if a server is launched unnecessarily, the business would experience increased charges for a *maximum* of one hour (the smallest billing granularity) because the scale-down metrics would terminate this server before the next billable hour began.

It is possible to configure arrays of servers in order to bound both the minimum and maximum number of instances. In our design a minimum of two standalone application servers are recommended for high availability and reliability of the application, thus the array minimum should be two. However, this minimum value should be increased if the minimum amount of application traffic increases and the two array instances are insufficient to handle this load. On the other hand, the maximum array size provides an upper bound on the total number of running instances. This upper bound can be used to place a limit on infrastructure costs. The optimal instance size for an application server in a scalable array can be determined

via load testing and performance benchmarking. This optimal instance size will be investigated in Chapter 6.

3.4 Caching Tier

The caching tier is located in the reference architecture between the application and database tiers. This caching is typically implemented with memcached[60]. This additional caching tier is not appropriate for all application architectures, because not all applications are compatible with a data caching solution. Fortunately, the majority of scalable applications will realize improved performance by using a distributed cache. For a read-intensive application, caching can provide a huge performance improvement due to reduced application processing time and avoiding database accesses. However, for write-intensive applications typically there is not a great benefit to caching, but with some modifications to the classic caching paradigm it may be possible to achieve a considerable performance improvement.

The memcached solution is fairly lightweight in terms of CPU utilization, but heavy (as heavy as the developer will allow) in terms of memory usage, so we advise that Ifoodbag use larger VM instance sizes (in terms of memory) for servers in this tier. Although in the early phases of an application's lifecycle, the total size of all of the cached objects will tend to be small and sometimes a single instance of the cache may be sufficient to provide a cache for the entire application server tier, but we do not advise this for production applications – especially if the traffic increases. Additionally, a single caching server is a potential single point of failure for the application's cache. A loss of this single cache instance can have a major negative performance impact on the application and its database. As a result we recommended that Ifoodbag use multiple instances of caching servers (distributed across multiple availability zones within the selected region/cloud) when implementing the caching tier.

3.5 Database Tier

The final tier in the reference architecture shown in Figure 3-1 is called the database tier. As for any web-based application this tier is quite critical and challenging to design correctly because there is no “one-size-fits-all” solution when it comes to data storage and management. Fortunately, there are a number of typical categories and types of applications that have an associated set of architectural components and best practices.

Among the numerous potential database applications, we have selected MySQL for *Ifoodbag* as it is one of the most common and widespread open source database packages. The architecture of the database tier is shown in Figure 3-2. This database architecture illustrates a scalable and recommended best practice for MySQL when used in the cloud. Although cloud-based resources enable application flexibility, maintaining physical accessibility of these resources requires additional planning and consideration. Although hardware failures are uncommon in the cloud, they do occur and need to be planned for. Hence we recommend that Ifoodbag use one or more database slaves that can take over if the current master database fails. If the master fails a slave can be quickly promoted to become the new master using pre-configured scripts. If the financial budget allows, we recommend placing additional slaves in different availability zones in order to increase the availability and reliability of the data store. While the ultimate goal of database design should be to allow automated horizontal scaling of the database tier, practical implementations of such a solution remain an indefinable goal. However, there are some design concepts that different applications can incorporate to allow database scaling to varying degrees. As previously mentioned, it is highly recommended that one or more slave databases be implemented in addition to the master database, regardless of the phase of the application's lifecycle – as loss of the stored data may lead to a business failure. Multiple slave databases will increase the overall reliability and availability of the

application. Additionally, these slaves enable horizontal scaling of the database using proxy mechanism for database reads, such as provided by MySQL Proxy (shown in Figure 3-2).

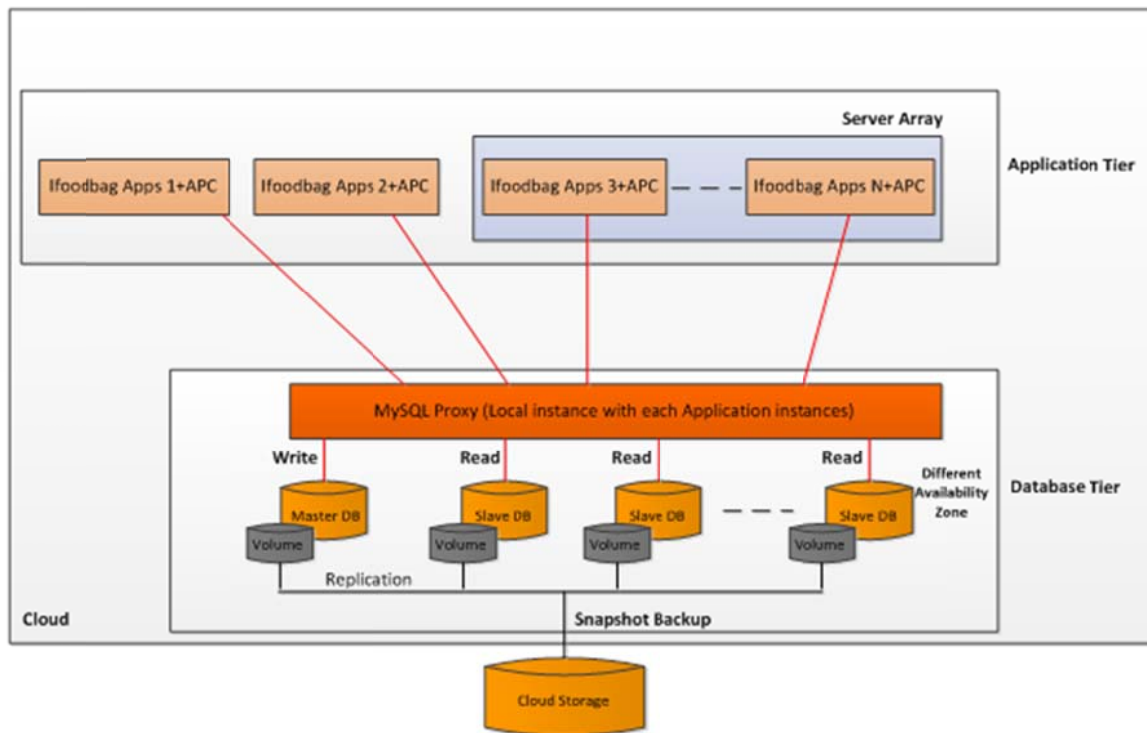


Figure 3-2: Database Tier for Ifoodbag Web-Application

In a MySQL Proxy configuration, no changes are needed in the application tier. Even though the application servers use their normal database connector, they point to the MySQL Proxy server instead of the master MySQL server. If an application performs a write operation to the database, the proxy server passes this request directly to the master database server, while if a read request is performed by the application, then the MySQL Proxy will send this request to one of the slave database servers, thus distributing the overall data read load of the application over the available slave database servers. This load distribution is quite important, but there is a risk that the replication delay to the slave databases may result in outdated data being returned in response to a read request if the read occurs soon after the data is written to the master database. As a result, for write intensive applications a proxy solution may not be the most effective method of database scaling. However, based upon our initial analysis the *Ifoodbag* application is dominated by read requests in comparison to write operations. We expect that this ratio will remain throughout the application's life cycle. Furthermore, given the nature of the database writes and the business needs it may be possible to delay all of (or at least most of) the write operations until an off peak time. As a result horizontal database scaling with MySQL proxy is appropriate for *Ifoodbag's* application.

3.6 Management Node/Nodes

Last but not least, a final component of this reference architecture is the management node/nodes. This/these node/nodes are actually the center of this reference architecture. Different methods will employ in order to provide scalability and high performance for the proposed cloud architecture. Several cloud management tools have being deployed in recent years, both in commercially and open source deployments [4, 5, 48, 61, 62]. We expect to utilize open source tools and customize them based upon our own design considerations and methods. These management node/nodes will employ all the scaling policies and monitoring

resources in the cloud architecture. Further details of these tools and the customization of them will be covered in Chapter 5 and Chapter 6.

3.7 Security guidelines in the architecture

Another Master's thesis project by Sabrina Ali Tandra and Sarwarul Islam Rizvi has proposed a set of security guidelines for our architecture[63]. Their security guidelines provide detailed security measures that should be applied to the DNS query to the DNS server and the recommend that HTTP's security be improved by using HTTPS, rather using HTTP. Also they propose that all of the nodes should use VPN tunneling instead of normal TCP connections for their communication with the management node(s).

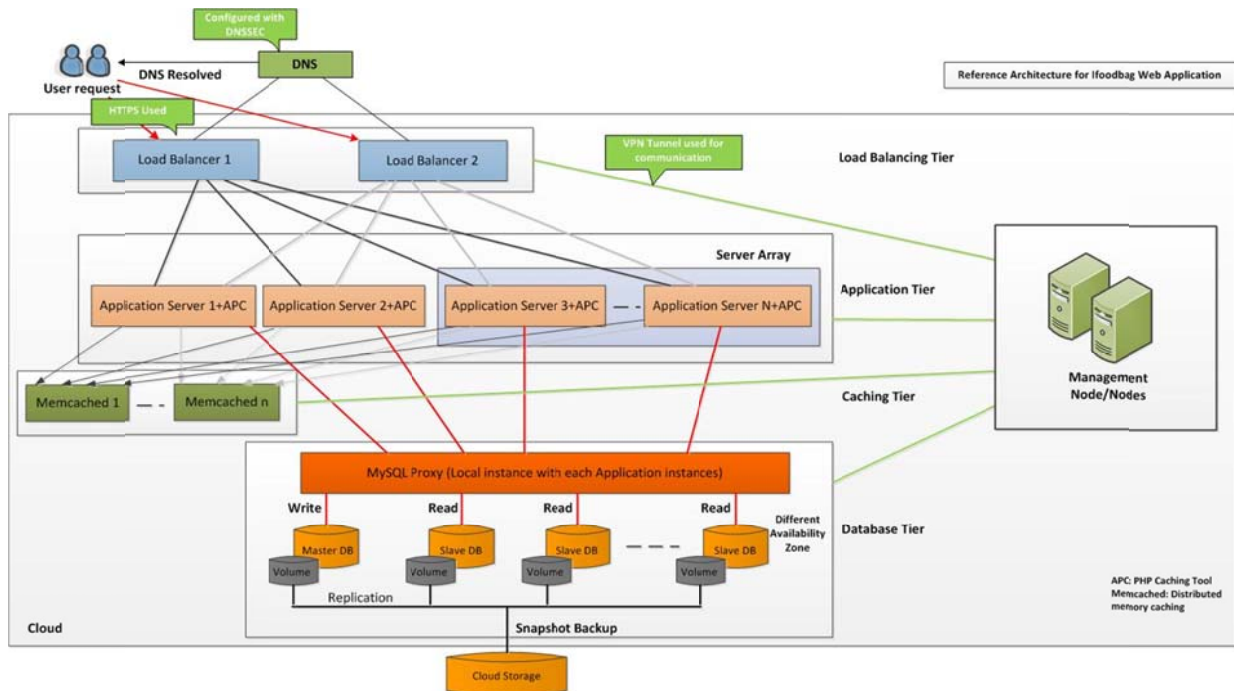


Figure 3-3: Architecture with security guidelines as recommended in [63].

4 Detail Descriptions of the Component in the Proposed Architecture

Using the reference scalable cloud architecture model described in the previous chapter, in this chapter we described each and every component in the various tiers of our proposed architecture.

4.1 DNS server

The Internet has billions of users and there are many resources distributed over this large network. From the user's perspective, each node on this network is identified by a unique name called the domain name. For example, a web server provides access to a website with a given domain name. A mail server is used for deliver email messages to a user within a given (e-mail) domain. From the network's perspective routers route Internet Protocol (IP) packets through the Internet based upon a unique network address (composed of a network and host interface portion). To access Internet resources using user –friendly domain names rather than IP addresses, the user need a means to translate the domain name to IP addresses and back. The Domain Name System (DNS) is an Internet service that translates domain names into IP addresses (and the reverse)[64]. Since domain names are alphanumeric, they are easier for most people to remember than IP addresses. This is especially true for IPv6 addresses as they are 128 bits long. Additionally, the name is likely to be a more stable identifier than and address as the structure of the network and the nodes attached to it may change, hence changing the mapping between a host interface name and an IP address. For these reasons domain names are widely used to identify Internet resources.

Because maintaining a central list of domain name to IP address correspondences would not be practical, knowledge about the mapping between IP addresses and domain names are distributed throughout the Internet in a hierarchy of authority [65]. When a user requests the IP address associated with a particular domain name, they probably query a DNS server in close network & geographic proximity to their access network provider. This DNS server either knows the mapping or forwards the query to another DNS server and so on, until a DNS server knows the IP address corresponding to the domain name in the query. After resolving the domain name to an IP address, the resulting IP address is returned to the user who made the query. Additionally, DNS servers along the way may or may not cache the mapping in anticipation of another query for this same mapping.

When users access Internet resources (e.g., a web server) through their web browser to retrieve the appropriate web page, the browser needs an IP address to contact this web server. Using DNS, the Web browser gets the information it needs to retrieve the requested web page. The process of using DNS to map domain names to IP addresses is called name resolution [66]. The DNS protocol is used to perform this action. For example, when a user want to access a web page for a web page with the domain name 'www.Ifoodbag.se' then the user's host queries a DNS server to learn the IP address of 'www.Ifoodbag.se'. When the DNS server returns the IP address (for example, '46.30.212.191') of that website then the user's browser can initiate a TCP connection to this particular IP address on TCP port 80 to access the first page at this web site. Figure 4-1 shows, how users can find the IP address of a specific web application via a DNS server.

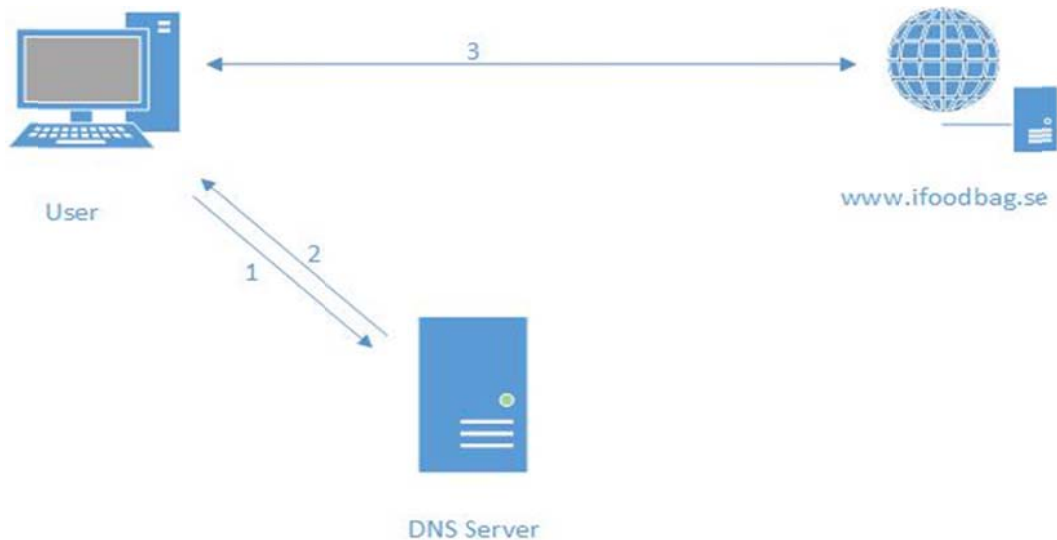


Figure 4-1: The normal DNS resolution process (adapted from [67]).

4.1.1 DNS infrastructure

The DNS server infrastructure is geographically distributed throughout the world. The domain name space is organized in the form of hierarchy with the top most level called the root domain, which is represent as a dot (“.”). The next level in the hierarchy is called a top-level domain (TLD), also called a child domain of the root domain. The next level domain is called a second level or enterprise level domains, and so on. There are more than 250 TLDs available under only a single root domain. Top Level Domains (TLDs) are categorized into the following three types [66]:

Table 4-1: Different types of Top-Level Domains (TLD).

Country-code TLDs (ccTLDs)	TLDs	These types of top-level domains are associated with countries and territories. There are more than 240 ccTLDs. Examples include .uk, .se, .in, and .jp.
Sponsored TLDs (gTLDs)	generic TLDs	These specialized domains have a sponsor representing a community of interest. These TLDs include .edu, .gov, .int, .mil, .aero, .coop, and .museum.
Unsponsored TLDs (gTLDs)	generic TLDs	These domains lack a sponsoring organization. The list of unsponsored gTLDs includes .com, .net, .org, .biz, .info, .name, and .pro.

A partial DNS name space hierarchy is shown in Figure 4-2. The DNS infrastructure consists of many name servers and each name server contains information about a portion of the domain name. Name servers are generally concern about the top three levels of the domain name space. If there is any further level of domain name space available, the DNS servers are either run directly by the organization or outsourced to an Internet Service Provider (ISP) or other service provider. For example, mail.Ifoodbag.se is a forth level domain, in that case the name servers outside of Ifoodbag.se know about only “Ifoodbag.se” and mail.Ifoodbag.se is run directly by the company and the DNS resolution of this name is done by a company operated DNS server associated with the domain name Ifoodbag.se.

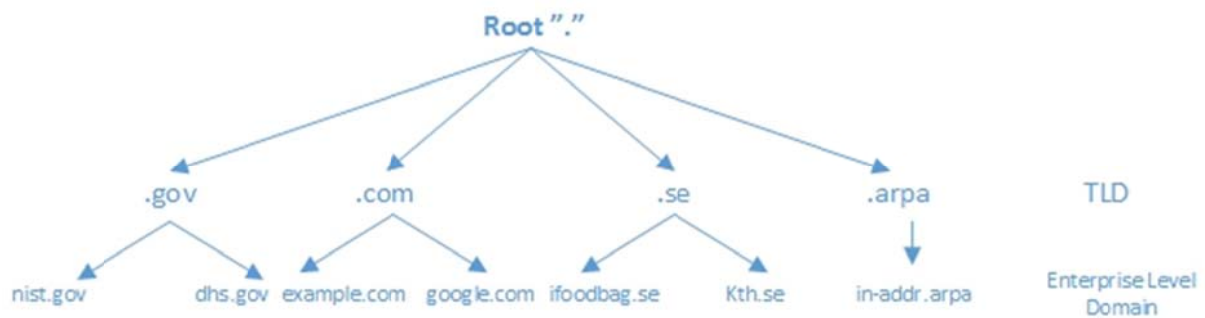


Figure 4-2: Partial DNS Name Space Hierarchy (adapted from [66])

4.1.2 DNS name resolving process

A resolver is a program that resolves hostnames to IP addresses (or the reverse) by communicating with appropriate name servers. Figure 4-3 presents a DNS name resolving process that consists of recursive and iterative queries when a visitor wants to browser the ‘Ifoodbag’ website. Usually, hosts perform recursive queries and a DNS server communicates with other DNS servers in order to resolve the query using iterative queries. However, each DNS query specifies whether an iterative or recursive lookup should be performed. The steps performed in this process are given in Table 4-2.

Table 4-2: DNS name resolving process [68]

- Step 1** A DNS client, also called a resolver, sends recursive query to a name server asking to resolve the IP address for www.ifoodbag.se (here we assume ‘ifoodbag.se’ as an example).
- Step 2** The DNS server immediate replies with requested information, if it found that information in cache. If it does not found anything then it forwards the request to a root name server.
- Step 3** The root DNS server recognizes the top-level domain name from that request and it knows the authoritative name server for top-level domains (TLDs). These top-level DNS servers know an authoritative name server for the .se domain. The server replies to the recursive name server with an IP address of the .se domain’s authoritative name server.
- Step 4** Now the recursive name server contacts the TLD (.se) name server for the Ifoodbag.se domain, which knows the mappings for hostnames in this domain.
- Step 5** TLD name server knows the authoritative name server for the Ifoodbag.se domain. So it replies to the recursive name server with this authoritative name server’s IP address.
- Step 6** Now the recursive name server contacts the authoritative name server for the Ifoodbag.se domain, which knows the mappings for hostnames in this domain.
- Step 7** The authoritative name server for the Ifoodbag.se domain replies with the IP address of the web server.
- Step 8** Finally, the recursive name resolving process has finished. Now the name server replies to the visitor’s host with IP address of ‘Ifoodbag’ web server. The resolver now stores this information in cache for communicating faster with webserver or to avoid resolving process. The DNS response has time stamp that indicates how long this response valid to communicate after that the information should be removed from the cache.

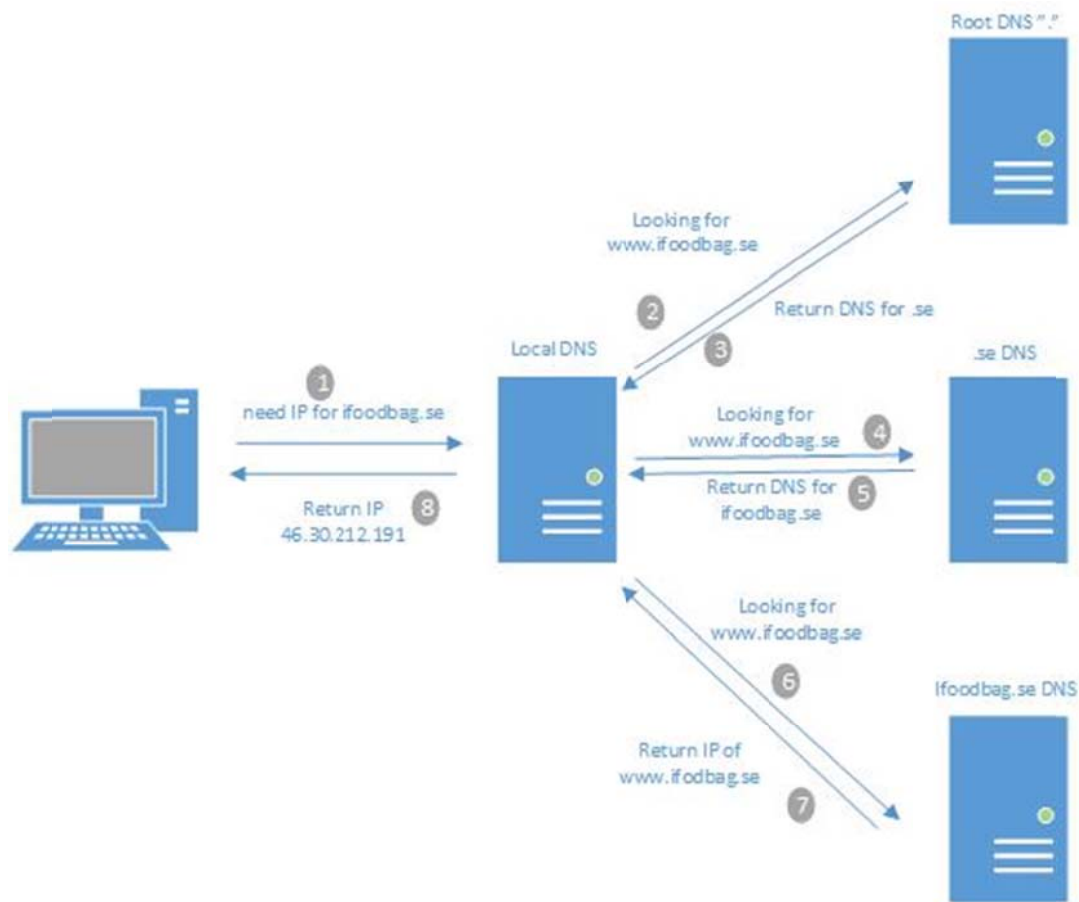


Figure 4-3: DNS name resolving process (adapted from [69]).

4.1.3 DNS security aspects

DNS security aspects and solutions based on our proposed architecture are described in the Master's thesis written by Sabrina Ali Tandra and Sarwarul Islam Rizvi and [63]. They propose that DNS Security (DNSSEC) be used to solve DNS security issues (such as DNS server attacks and DNS protocol attacks). Further details can be found in their thesis.

4.2 Load Balancer (Squid/HA Proxy)

Load balancing is a core networking solution that is responsible for distributing incoming traffic over a set of servers hosting the same application content. The main purpose of using a load balancer in a scalable architecture is to distribute application requests across *multiple* servers, thus avoiding problems that would occur with the failure of a single application server. Using a load balancer improves both application availability and responsiveness[70]. To scaling out an application server infrastructure, load balancing is the most straightforward method. New servers can be easily added to or removed from the resource pool as demand increases/decreases. Sending traffic immediately to these new servers is the main responsibility of a load balancer. To balance server load, the load balancer distributes requests to different nodes within the server cluster (a group of servers simultaneously running a given web application), with the goal of optimizing the overall system's performance.

Using a load balancer results in higher availability and increases the scalability of an enterprise web application. In this approach high availability is provided through redundancy. Thus, if any single server fails, another server takes its place as soon as possible to process the requests resulting in a highly available system [71].

As an example consider Yahoo, millions of users throughout the world access Yahoo’s portal every day. A given end user may visit Yahoo’s web application repeatedly and each of these users expects the same or better performance each time they visit this website. If the user does not get satisfactory performance, then this portal might risk losing its user base to its competitors. Each user request causes a certain amount of load on the web servers associated with this site. With millions of requests, the load on these servers can increase rapidly. Because each server has only a finite amount of computing power, to handle that site’s aggregate load requires a collection of servers (perhaps organized as server clusters). However, each server executes its copy of the web application separately so as the load increases the requests need to be distributed across the set of servers in order to maintain the same level of performance as seen by the end users. A load balancer distributes the incoming requests across the servers [72]. Figure 4-4 depicts a network diagram of how a load balancer can be used to balance the offered load among multiple application servers.

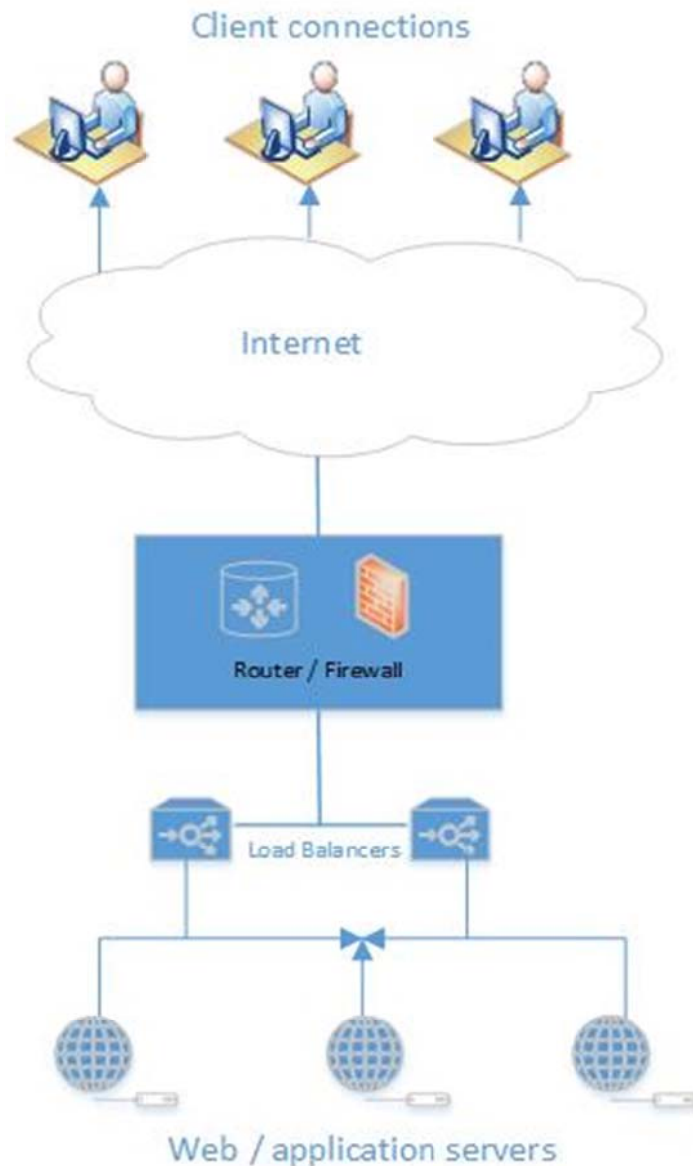


Figure 4-4: Load balancing for balancing load among multiple application servers (adapted from [73]).

4.2.1 How to calculate response time

In September 2006, Peter Sevcik and Rebecca Wetzel of NetForecast published a paper called “Field Guide to Application Delivery Systems”[74]. The paper focused on improving wide area network (WAN) application performance and included the equation shown here as Equation 4-1.

$$R \approx \frac{\text{Payload}}{\text{Bandwidth}} + \text{AppTurns}(\text{RTT}) + C_s + C_c$$

Equation 4-1: The original WAN performance equation [74].

Equation 4-1 showed WAN performance, but with a few minor changes it can be used to measure web application performance. The modified equation is shown in Equation 4-2.

$$R \approx \frac{\text{Payload}}{\text{Bandwidth}} + \text{RTT} + \frac{\text{AppTurns}(\text{RTT})}{\text{Concurrent Requests}} + C_s + C_c$$

Equation 4-2: The web version of performance equation [75]

Website performance depends on response time. A lower response time will improve the user’s satisfaction when visiting the website, while a higher response time degrades the user’s satisfaction. The terms used in these two equations are described below:

R	The response time is the total time from when the user requests a page to when the pages are displayed by the user’s web browser. Typically the response time is measured in seconds (or milliseconds).
Payload	The payload term represents the total number of bytes sent by the web server to the browser, including markup and all resources (such as CSS, JS, and image files).
Bandwidth	Rate of transfer to and from the browser. If the requested web page is generated from multiple resources then the bandwidth might asymmetrical. Usually, bandwidth is expressed in bytes per second by averaging all of the separate bandwidths.
Round Trip Time (RTT)	Amount of time required for a data packet to traverse from the user’s browser to the server and back.
AppTurns	The number of resource files a requested page needs. These resources files could be images, JS, CSS, or any other files retrieved by the browser in the process of rendering the page.
Concurrent Requests	Number of simultaneous requests a browser will make for resource files. For example, Internet Explorer performs a maximum of two concurrent requests by default.
C_s	Compute time on the server.
C_c	Compute time on the client.

4.2.2 Different types of load balancers

Load balancing comprises several different types of utilizing server clusters, network devices, CPUs, disk storages and other resources. It is the optimal way to use resources that distribute tasks into different available resources. Some load balancing strategies are described in Table 4-3.

Table 4-3: Different types of load balancers [76]

<i>Software-based load balancers</i>	Software based load balancers are traffic splitters that help reduce workloads on compute servers by distributing traffics to different servers. Citrix (netScaler), F5 (BIG-IP), Squid, HA proxy are examples of software based load balancers.
<i>Hardware-based load balancers</i>	Hardware based load balancers are greater deployment capabilities. Such a load balancer is more flexible to work with as they frequently are designed to support any TCP port or application service. Using this method is more complicated than using a software-based load balancer, while providing a competitive edge over software based load balancers. Hardware based load balancers typically have fewer less flaws as compared to strictly software based balancers.
<i>Round-robin DNS</i>	The service host can be selected by DNS using round-robin load balancing techniques, over several servers with identical services. This method is ideal for geographically distributed and internationally used web servers as traffic can be preferentially distributed locally to reduce communication delay or distributed more widely to further distribute computational load.

4.2.3 Squid

Squid is a proxy and web-caching server (available under the GNU Public License) that many organizations use to speed up client access to commonly accessed data. It is a caching proxy for the web supporting protocols, such as HTTP, HTTPS, FTP, etc. It reduces bandwidth demands upon the web server and improves response times by caching and reusing frequently requested web pages [77]. Squid reduces the load on the application servers by reducing the number of user requests that must actually be served by the web server.

When a HTTP request is made, the web server needs to serve some amount of content (such as scripts, HTML, CSS, and images). If a single server needs to serve all this content for hundreds of requests, performance will degrade as the number of requests increase. With a single server responding to hundreds of HTTP requests per second, the system's performance depends on factors such as: the number of scripts, amount of HTML/CSS, number & size of images, etc. that need to be served. If the load is sufficiently high, the end users many experience poor performance. Squid can be configured in such a way that it is able to round robin load balance according to the request. For example, if an 'Ifoodbag' user requests a page containing images, HTML, CSS, or a script connected to the database, then webserver-1 could be ask to serve the images while webserver-2 will serve CSS or database content. The distribution of the load will depend on how many web servers are available in the server cluster at a given time.

Squid allow web service administrator to distributed more requests to servers with better hardware and fewer requests to servers with poorer hardware using weighted round robin queuing. Squid can also notify the web service administrator that all the requests from a given user id failed for some reason. Squid proxy servers can establish hierarchical relationships through which cache data can be shared and requests can be passed to the proxy server in standard proxy mode [78].

4.2.4 HA Proxy

The HA (High Availability) proxy provides a high availability, load balancing, and proxy solution for TCP and HTTP based applications. It is open source, free, very fast, and quite reliable in comparison to other solutions. This solution is particularly well suited for websites with high loads. Because of its mode of operation, it is extremely easy and riskless to integrate with an existing architecture, while it also offers the possibility to avoid damage to web servers.

Today, processing tens of thousands of requests per second from users is possible for hardware, but these sorts of loads are rarely handled by multi-process or multi-threaded models because of memory limits, system scheduler limits, and lock contention. To support these high loads the HA proxy utilizes an event-driven, single process model that allows it to support a very large number of connection while operating at very high speed. The event driven model does not have the limitations of multi-process or multi-threaded models because it allows all of the task to run in user space with fine grained resource and time management [57].

4.3 Web server/Application server

A webserver is a program that utilizes software and networking to deliver web pages via the Internet or an Intranet. Two leading web server are Apache (the most widely used webserver) and Microsoft's Internet Information Server (IIS). In this project, Apache has been used as a webserver. Powerful features (such as openness, extensibility, portability, and flexibility) of the Apache webserver provide benefits to website administrators which lead to higher efficiency and greater utility [79]. Ifoodbag's web application was built with the open source programming language PHP and uses MySQL as its database.

In this project we have used a group of servers, organized as a server cluster or server array, working closely together to improve performance and/or availability over that provided by a single server. Our goal is that if any system failure occurs then other webserver is used to provide services to make system (i.e., by exploiting redundancy to increase reliability). All web servers within the cluster are built by installing the same application and they are all connected to same database.

When a failure occurs on one server in a cluster, another server takes over and the workload is redistributed to another server within the cluster. The benefits of using a server cluster are that it ensures users have constant access to important server-based resources. This solution is also well suited for applications that have long-running in-memory state or frequently updated data. In our architecture two servers will always be active in the cluster, the number of additional servers will scale up or down according to demand. These additional servers also provide a higher level of availability, reliability, and scalability compared to using a single computer. Some of the possible reasons for creating a server cluster are [80, 81]:

- Avoiding application and service failures, which could affect web and essential services.
- Avoiding system failures and reducing the impact of hardware failures, which affect different resources (such as CPUs, drives, memory, network adapters, and power supplies).
- Minimizing the impact of site failures in multisite organizations, which can be caused by natural disasters, power outages, or connectivity outages.

There are a number of different web servers available to host your applications. A few of them are free, while others are available on a pay to use basis. There are four leading web browsers: Apache, IIS, lighttpd, and Jigsaw. In addition to these web servers, there are additional commercial web servers available in the market, but they are very expensive. Major commercial web servers are Netscape's iPlanet, Bea's Web Logic, and IBM's Websphere. Table 4-4 briefly describes a few of these web servers.

Table 4-4: Different types of web servers [82, 83]

<i>Apache HTTP Server</i>	The Apache HTTP server is the most popular webserver in the world. It was developed by the Apache Software Foundation. The Apache web server is open source software and offers cross platform support. More than 60% of web servers world-wide run the Apache Web Server [84].
<i>Internet Information Services</i>	Microsoft's Internet Information Server (IIS) is a high performance Web Server. Because IIS is tightly integrated with the OS it is relatively easy to administer. IIS offer increased choice and control, without giving up reliability or security [85].
<i>Lighttpd</i>	Lighttpd is a free web server designed for speed. It provides all the essential functions of a web server. Jan Kneschke, a German software developer, developed Lighttpd. It is designed to have low memory consumption, be fast & secure, and offers more effective management of CPU load compared to other web servers. Lighttpd is frequently a solution for servers that are suffering load problems. It is open source software licensed under the revised BSD license.
<i>Sun Java System Web Server</i>	This web server from Sun Microsystems (now Oracle) is a secure, easy to use web server well suited for medium and large web sites. Although this web server is free, it is not open source. It is available for most major OSs, specifically it runs on Windows, Linux, and Unix platforms.. It offers built-in HTTP reverse-proxy capabilities to provide a highly scalable HTTP front-end to applications. The Sun Java System web server supports various languages, scripts, and technologies such as PHP, Ruby on Rails, Perl, Python, and more.
<i>Jigsaw Server</i>	Jigsaw server is a java-based webserver deployed by the World Wide Web Consortium (W3C). It is open source and free and can run on various platforms (such as Linux, Unix, Windows, Mac OSX, and Free BSD). The Jigsaw server is an experimental platform for W3C and the Internet community with a modular architecture and full HTTP/1.1 support.
<i>Apache Tomcat</i>	Apache Tomcat or simply Tomcat is an open source webserver and servlet container developed by the Apache Software Foundation. It implements Java Servlet and Java Server Pages technologies. Apache Tomcat is one of the most popular options for lightweight development scenarios. Even though it is a web server, it can also meet the requirement for an application server in many cases.

4.4 Caching web data (memcached)

Cache is a high-speed access storage area that can be a reserved portion of either main memory or a storage device. Caching is the process of storing data in a cache. Today's cloud supports a number of caching engines. For example, AWS ElastiCache supports two open source cache engines:

Memcached Memcached was developed by Brad Fitzpatrick for LiveJournal in 2003 [86]. Today, top worldwide websites and portals such as Facebook, Wikipedia, twitter, and others use memcached. Memcached is an open source & free, high performance, distributed memory object caching system, intended to speed up dynamic web applications by alleviating database load [87]. Memcached aims to decrease high database loads by adding a scalable object-caching layer to an application[60]. Many large companies use mamcached in their system (such as: LiveJournal, Wikipedia, Slashdot, and Digg). Memcached is designed to be simple in order to promote rapid deployment, ease of development, and to solve many problems facing large data caches.

Redis Redis is a popular open-source in-memory key-value store that supports data structures such as sorted sets and lists. Redis supports cross machine redundancy using master/slave replication.

Memcached was developed with some specific underlying assumptions, such as fast networks, cheap memory, and that memory storage should be spread out across multiple machines rather than a single server. A global hash table is responsible for a cache that can access multiple web processes to learn of changes made by others and to respond appropriately. Table 4-5 describes the major tasks perform by memcached to speed up the response process.

Table 4-5: Different tasks perform by memcached [60].

Server Instances	Generally, a number of memcached server instances are running throughout the network wherever free memory is available. Memcached instances listen on a specific port and IP address. A specific amount of memory is assigned to memcached on each machine. The memcached software will use all the spare memory dedicated to it over the entire network. Multiple server instances are easy to handle by configuring them to listen on different ports.
Client Read Process	When an application determines what object is needed, it uses a key (such as object id) as input to a hashing algorithm to check whether the object is available or not. If the object is available, the object is returned in response to the request; otherwise memcached fetches the object from the database and places a copy of it in its cache for later use.
Client Write Process	When an object fetched from the database or cache is updated then the updated object is saved in both the database and the cache. This maintains the integrity of the data, but also involves an extra update to the database.
Hashing and Keys	In a client server relationship, the server instances store the data of different web servers and provide this data to the server at some time in the future. The web application maintains a hash table to determine which server instance stores information about what information in stored by which memcached instance(s). So that, requested objects are sent to appropriate server in the distributed cache <i>before</i> accessing database to respond to requests. A set of keys is used to look up results via a hash. Eventually, the keys (and the information associated with these keys) are spread out across the multiple nodes running memcached.
Independence	Each memcached server instances is operated independently. If a server fails within a memcached cluster, then the remaining active servers run as normal. However, clients can be configured to route requests to other machines. All data contained within theinstance that fails will be lost when it fails.
Expiration	Memcached follows the least recently used (LRU) caching principles, hence it discards the least recently used data first from it memory. That means, the most frequently used data will remain in the cache, while data that are not used frequently will be phased out as new data enters the cache.

4.5 Database

A database is a collection of information that is organized so that it can easily be accessed, managed, and updated. Typically a database contains aggregations of data records or files, such as user information, product details, transactions, and inventories. A database administrator provides users with the appropriate capabilities to control read/write access, generate reports, analyze usage, and so on. In the ‘Ifoodbag’ infrastructure we use a distributed database in order to place data in different geographic locations.

A distributed database is a database that is under the control of a central Database Management System in which storage device are not (all) attached to a common CPU. Data may be stored in multiple instances and these instances can be located either in the same geographic location or different geographic locations spread though out the network. To ensure that the distributed databases are up to date, replication and duplication processes ensures that each of the databases contains up to date data over time. Replication uses specialized software that looks for changes in the distributed database. When changes are identified, the replication process distributes all of the databases changes to ensure that queries will return the current data. In our project we used a master-slave replication process to update our database when any change occurs.

Master-slave replication enables data from one database server (the master) to be replicated to one or more other database servers called slaves. When the master logs the updates, they ripple through to the slaves. Each slave sends a message stating that it has successfully received each update. Replication in a master-slave combination can be either synchronous or asynchronous. The difference between these is simply the time when the changes propagate. If the changes are made at the same time in both the master-slave databases, then the update is synchronous. If changes are enqueued and written later, then it is asynchronous [88]. Figure 4-5 depicts master-slave replication of databases.

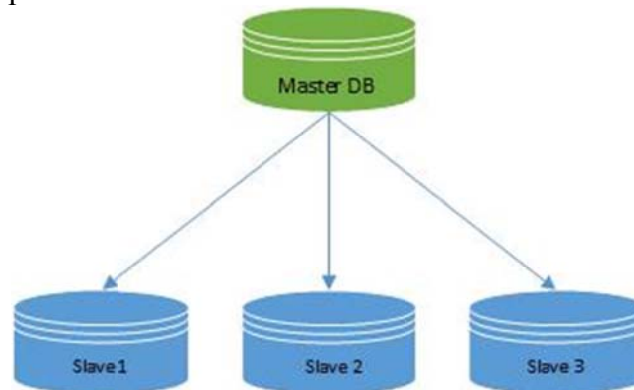


Figure 4-5: Master-slave replication of databases (adapted from [88])

Master and slave replication of databases is designed to improve performance in responding to user requests. Table 4-6 describes the advantages of using master-slave replication in the system.

Table 4-6: Advantages of master-slave replication [88]

Scale-out solutions	To improve performance we can split the load of database queries across multiples database servers. Replication distributes the update of one master to one or more slaves. If any application requires more reads than writes or updates, then a master-slave solution is well suited for this environment. In such a scenario, all writes and update must take place on the master server, while reads can utilize any one of the slave servers.
Analytics	Live data can be updated or created in the master, while analysis of the data can take place without negatively affecting the performance of master.
Long-distance data distribution	Long distance data distribution is so easy. For example, when a local office needs to work with a subset or all of the data; then you can easily create a local copy of the data for their use without giving them permission to permanently access the master server.
Backups	As data is replicated to the slave, slaves are able to pause the replication process or make a backup without corrupting the corresponding master data.
Increasing the performance	Master-slave replication can improve the performance of writes (since the master is dedicated to updates), while dramatically increasing read speed across an increasing number of slaves.
Failover alleviating	In a master-slave environment, any slave can become the master in the event of a failure of the master.
Spreading the load	Load can be spread across different slave databases, as needed. For example, different sales data could be distributed to different departments – so that each department has the data relevant to it.

4.6 Cloud Storage

Cloud storage is a model of networked enterprise storage where data is stored in virtualized pools of storage. The cloud provider operates large data centers. Then organizations that require their data to be hosted buy or leased storage capacity from the cloud provider. Physically, the resources might be located in different geographical locations, thus the safety of data depends upon the cloud provider and on the application that leverages the cloud storage. Cloud storage provides a user with the ability to back up data stored on a server, typically this server is hosted by a cloud service provider [89]. Most service providers also offer redundant storage. When a data center is hit by a natural disaster or power outage, the data can still be safe and available to the user through an identical copy of the data stored in a separate data center. Amazon Glacier is an example of cloud storage, which offer an extremely low-cost storage service that provides secure and durable storage for data archiving and backup. Using a cloud Storage Gateway, you can back up to a given point-in-time snapshot of your on-premises application data to cloud storage for future recovery. There are three important characteristics of cloud storage [90, 91]:

- First, consider a storage service over a network. Purists will insist that this network must be the Internet and it must use a web services API and REST protocol.
- The second characteristic is that the solution is easy to scale. Scaling is more than just increasing capacity. Scaling should address how to reduce effort and costs. Effort is reduced by the removal of detailed provisioning tasks intrinsic to traditional storage.
- The last characteristic is that the solution is easy to manage. Every storage vendor claims their solution is easy to manage. A single administrator can manage a petabyte across hundreds of servers. Cloud storage should be easy to manage, as it is a single storage appliance. Today most cloud storage management is truly easy, regardless of the claims by different storage vendors.

A database snapshot is a read-only, static view of a database (the source database). A snapshot is a copy of your disk volume at a specific moment in time. It contains the full directory structure of the volume. A snapshot can also be used for incremental backup of volumes; such as, a restore point of your database, long-term storage, or the starting point of new Cloud Block Storage (CBS) volumes [92]. A snapshot persists until the database owner explicitly drops it. Snapshots and replication in a conventional storage system can serve the same function as a traditional backup strategy.

4.7 Management node

Cloud management strategies typically involve dealing with important tasks, including performance monitoring (response times, latency, uptime, etc.), security, compliance auditing and management, and contingency plans. Ideally, you can perform this entire set of tasks from a management node. There are a lot of management tool available. With a management tool you can manage the cloud infrastructure, including provisioning management and automation of enterprise class applications across private, public, and hybrid cloud platforms. A management tool allows you to automate updates and manage physical, virtual, and cloud based systems from a single interface. The management node can also accelerate delivery of innovative services and simplify control of virtualized environments. In this project, we used ‘Cloudify’ as the management tool to manage our cloud platform. Details can be found in Chapter 5. The main reason to use management node are [93, 94]:

- Reduce infrastructure costs and complexities with an integrated management platform,
- Increase speed of deployment IT operations, and
- Enable dynamic cloud service delivery with reusable workload patterns.

5 Implementation

Based on the proposed architecture (described in Chapter 3), in this chapter we consider the motivation for the proposed solution and describe an experimental setup to measure the performance of this proposed solution.

5.1 Experimental Setup

To implement a scalable realization of Ifoodbag’s web application in a cloud environment the proposed design utilizes nodes in different tiers which are connected to each other. Management nodes are connecting to each of these different tiers in order to monitor the complete cloud’s health and to implement a policy for scaling the number of applications instances up and down either manually or automatically based on the traffic or user load.

Figure 5-1 shows the experiment setup that we used to measure the performance of the proposed solution. In this experimental setup we mainly used the Cloudify* manager, Amazon Elastic Compute Cloud (Amazon EC2)†, and a simple static Ifoodbag web application including Apache Tomcat service under the license agreement [95] for testing with Cloudify. The Cloudify control machine is installed in a personal laptop on a private LAN and the EC2 cloud instances resides in AWS.

Table 5-1 shows the details of each component of the experiment setup. To simulate the proposed solution and to perform basic testing of the Ifoodbag application, we emulate Cloudify in the local cloud [96]. In the following sections we described Cloudify and AWS EC2 clouds, including our main motivations for selecting them, how to install them, and how to deploy them in our experimental setup. Finally in section 5.6 we described a mechanism for generating a traffic load for the Ifoodbag application server in order to experiment with our scaling rules as defined in the Ifoodbag application recipe.

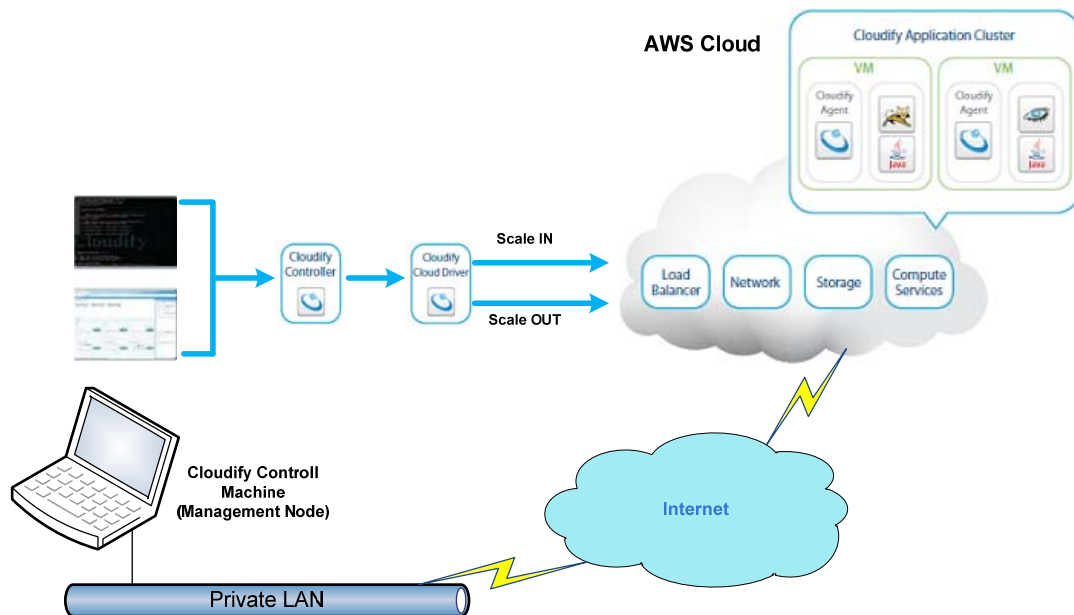


Figure 5-1: High level experimental setup using Cloudify and EC2 clouds

* Cloudify is an enterprise-class open source PaaS stack that sits between your application and your chosen cloud. Details can be found at: <http://www.cloudifysource.org/>

† Amazon Elastic Compute Cloud (Amazon EC2), available at: <http://aws.amazon.com/ec2/>

Table 5-1: Experimental configuration

Network	Cloudify machine is sitting on the gigabit per second private LAN. There is a fixed link with 250 Mbps of bandwidth from this LAN to the Internet. This fixed link was used to connect to the public network. Additionally, we tried to connect to the public network via 3G internet connectivity.
Web client	As a web client we could use any web browser (e.g. Internet Explorer, Google Chrome, or Mozilla Firefox).
Client machine	Operating System: OS X Mavericks 10.9.1 RAM: 8.00 GB Processor: Intel®Core™ i5 @ 2.6GHz, x64-based processor Hard disk: 250GB
Cloud Instances	Instance Type: m1.small (See section 5.5 for details on Amazon Instance types and pricing) Configuration of the instance: RAM: 1.7 GB, OS: Amazon Linux 3.4 AMI 2013.09.2, 1, Processor: 32 bit 1 vCPU of Intel Xeon Processor
Tools or Software Used	We used the latest version of Java JDK as this was a prerequisite for installing the Cloudify shell [97]. We used TextWrangler tools [98] for configuring (changing/configuring .groovy files, as cloudify is mainly written in the groovy language[99]).
Plots	To create plots we used Microsoft's Office Excel 2010.

5.2 Motivation for choosing Cloudify

Some of the drivers for moving to clouds include on-demand availability and scalability (enabling us to scale the application resources to efficiently consume what is needed now); rapid deployment and agility (as this reduces the time to deploy the application by utilizing an agile process for provisioning computing resources); and cost-reduction (thus enabling us to realize cost benefits by procuring cloud based computing resources *without* the overhead of system administrators or need to set up the underlying infrastructures).

However, using clouds commonly requires us to compromise on one or more of the following objectives:

No Code Change One of the primary objectives when moving to a cloud solution is to deploy the application in the cloud *without* rewriting any of the code. This can be challenging if the application is not ready for an elastic cloud-based deployment. It can be significantly more difficult if the cloud's prepackaged images do not provide the versions of services on which the application relies or because these images simply do not provide the specific environment that the application requires.

No Lock-in An important objective is facilitating moving to a different cloud provider when necessary. Maintaining this flexibility requires that we avoid customizing the application to work in a specific cloud, as such customization can make it challenging to change to another of the many cloud providers due to the complexities of migrating to a new architecture.

Full Control It is desirable to have full control of the environment in which the applications runs in order to fine-tune, monitor, upgrade, and configure resources in the cloud according to the application's needs. This means we must avoid the limiting environments that some clouds provide.

After considering all of the above objectives and our organization’s individual business requirements, Cloudify was selected as our enterprise cloud management tool. This tool fits well into our proposed cloud architecture for Ifoodbag. An introduction to this tool will be given in the next section.

5.3 Introduction to Cloudify

Cloudify is an enterprise-class open source PaaS stack that sits between the application and the chosen cloud [100]. It enables the application to operate smoothly, while Cloudify monitors the applicaiton and ensures that the resources the application needs are available *regardless of the cloud and stack used*. Cloudify offers the following features:

Any App, Any Stack Any application can be moved to the cloud without changing any code, regardless of the application stack (i.e. Java/Spring, Java EE, Ruby on Rails, etc.), database (e.g., relational databases, such as MySQL, or non-relational databases, such as Apache Cassandra, etc.), or any middleware components that the application uses. This enables us to achieve the objective of “no code changes”.

Any Cloud Any application can be moved to any cloud environment, from any platform, at any time. Cloudify supports or has been tested on almost all public/private clouds, including Amazon EC2, Windows Azure, Rackspace, and private clouds (such as OpenStack, CloudStack, VMWare vCloud, Citrix XenServer, etc.) Additionally, enterprises can deploy the same application in multiple environments (say, for cloud bursting [101]). Cloudify hides the APIs and configuration of a cloud from the application, thus the application can more easily be moved from cloud to cloud. This enables us to achieve the objective of “no lock-in”.

Full Control The application can have the full control of its environment. In many clouds, there is less control because the underlying infrastructure does not exposed suitable interfaces to the public, and hence a management tool cannot monitor and fine-tune the cloud for the application as it would with traditional data centers and applications. However, Cloudify does have access to the infrastructure via its cloud driver and controller, hence it can provide a much greater level of control, if the organization wants [100].

5.4 Deploying Cloudify

We deployed Cloudify version 2.6 in “.nix” (MAC OS X Mavericks) machine, but it also works for machine running Microsoft’s Windows OSs [102]. There are some prerequisites [103] to compile the Cloudify distributions (details are available at [104]). In order to run the Cloudify shell, after downloading the distribution you simply unzip it, then browse the bin directory of the distribution and run the “./cloudify.sh (for .nix)” or “cloudify.bat (for Windows)” file (detailed step by step installation and configuration are available in Appendix A). Figure 5-2 shows the Cloudify shell prompt after running the cloudify.sh file. In the following paragraphs we described how Cloudify works and what the Cloudify architecture looks like.

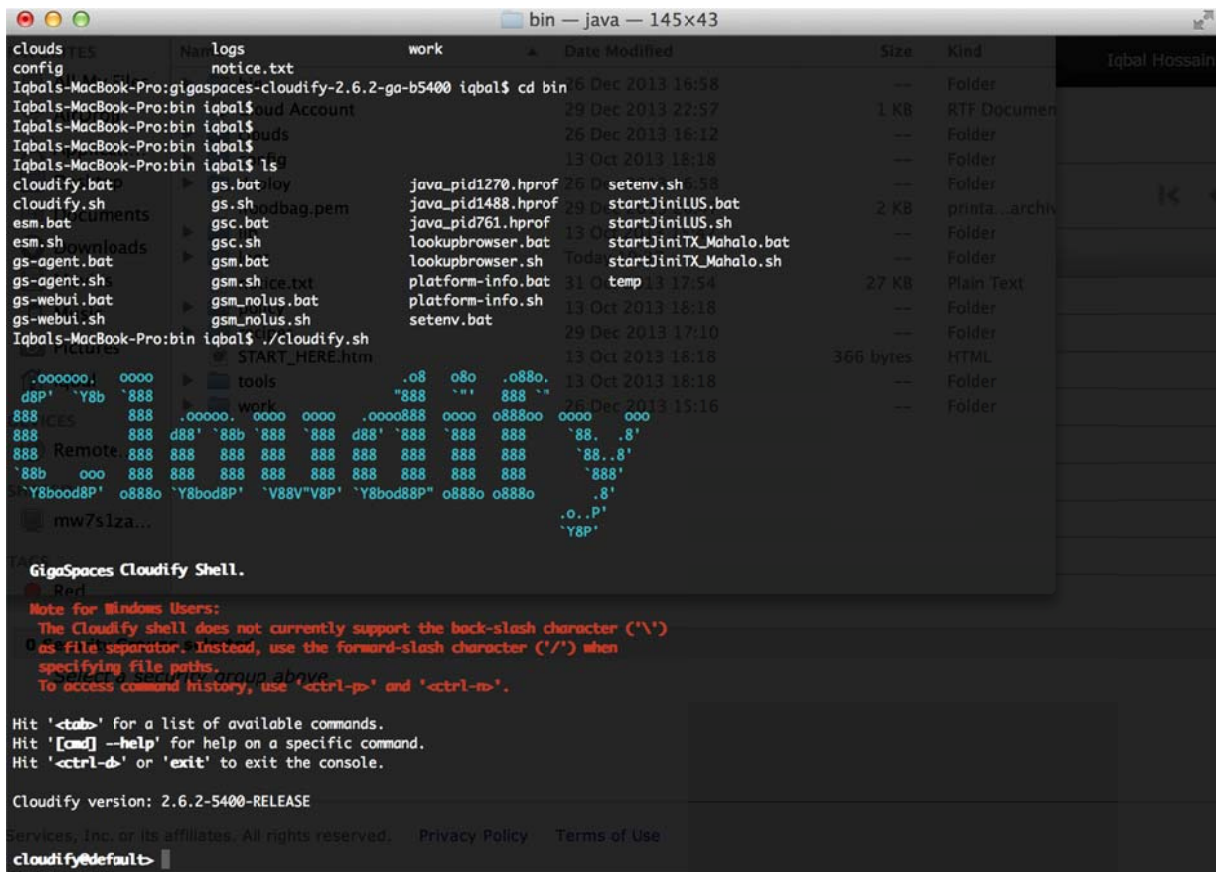


Figure 5-2: Cloudify Shell

Cloudify uses sets of instructions and methods called *recipes* to describe an application and its services & their interdependencies, in order to monitor, self-heal, and scale in/out its services & their resources. Deploying and managing the application with Cloudify becomes a simple process, as follows:

1. Deployment preparations

- Cloud setup (for our implementation we used EC2 clouds) and configured machine details (hardware, image ID, location, key file, etc.) in the cloud driver. For our experiment we used Linux image with m1.small instance and location as Europe zone. For detail about EC2 cloud setup, instances, hardware, and locations - see section 5.3 of this chapter.
- Prepared the binaries required for the services and application. For our experiment we used simple Apache Tomcat* and run a simple jsp file to launch the Ifoodbag application in the cloud.
- Finally describe the application's lifecycle and its services using recipes.

2. Deploy the application and its services

- Provision required machines in the cloud by configuring cloud drivers (in our case EC2 cloud drivers).
- Download, install, and configure the required services of the application (in our experiment this is Apache Tomcat).

* Apache Tomcat is an open source software implementation of the Java Servlet and JavaServer Pages technologies. Details available at: <http://tomcat.apache.org/>

- Install the application (for our experiment we used a simple Ifoodbag application).
 - Configure the metrics used to monitor and scale the application's features.
3. Finally monitor and manage the deployment using Cloudify web management graphical user interface (GUI) or the Cloudify shell.

The above processes are made simple and possible due to the Cloudify architecture. This architecture enables us to achieve the application's objectives, as follows:

No Code Change: Just to configure cloud drivers (in our case us ec2-cloud.properties and ec2-cloud.groovy files) and Ifoodbag application recipes. Figure 5-3 shows the procedure to achieve the No Code Change objective.

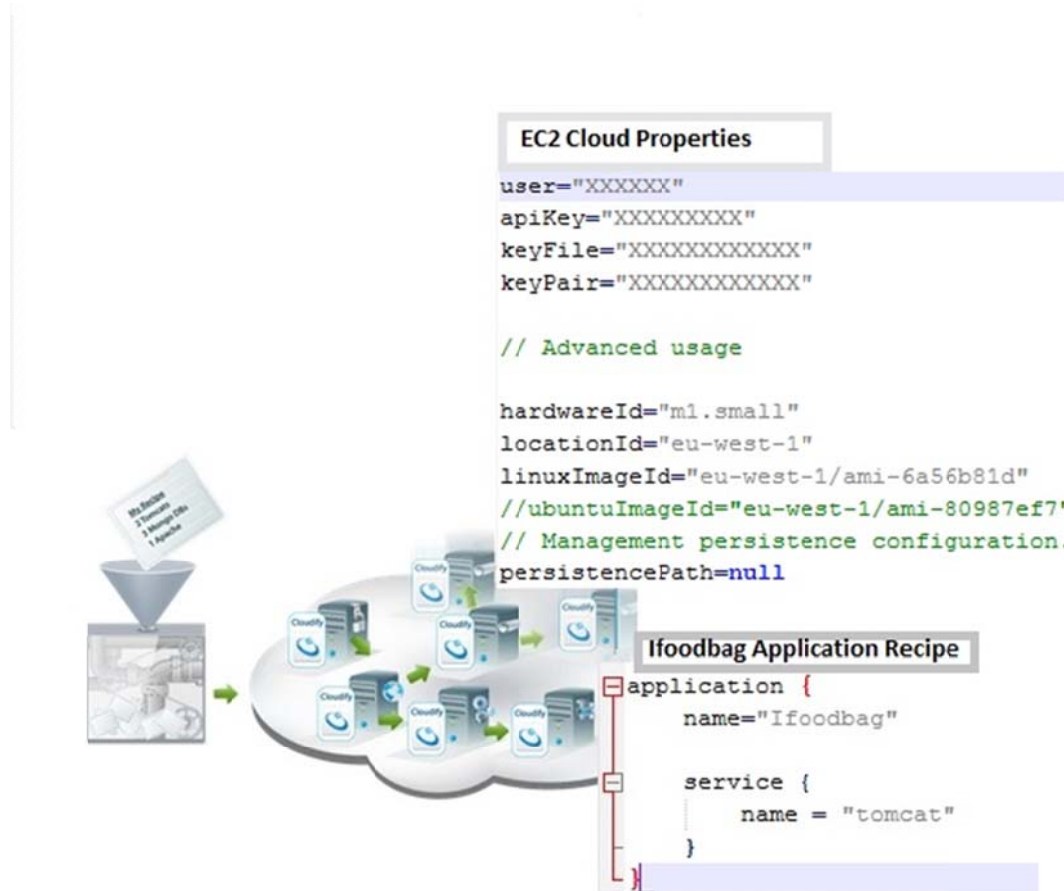


Figure 5-3: Achieving the No Code Change objective

No Lock-in: This objective is achieved by configuring the compute template sections of the cloud drivers. For our experiment we used the "SMALL_LINUX" template. This template exists in any open cloud platform. Hence there is No Lock-in if in future if Ifoodbag wants to move from EC2 to any other cloud platform. Figure 5-4 shows this objective.

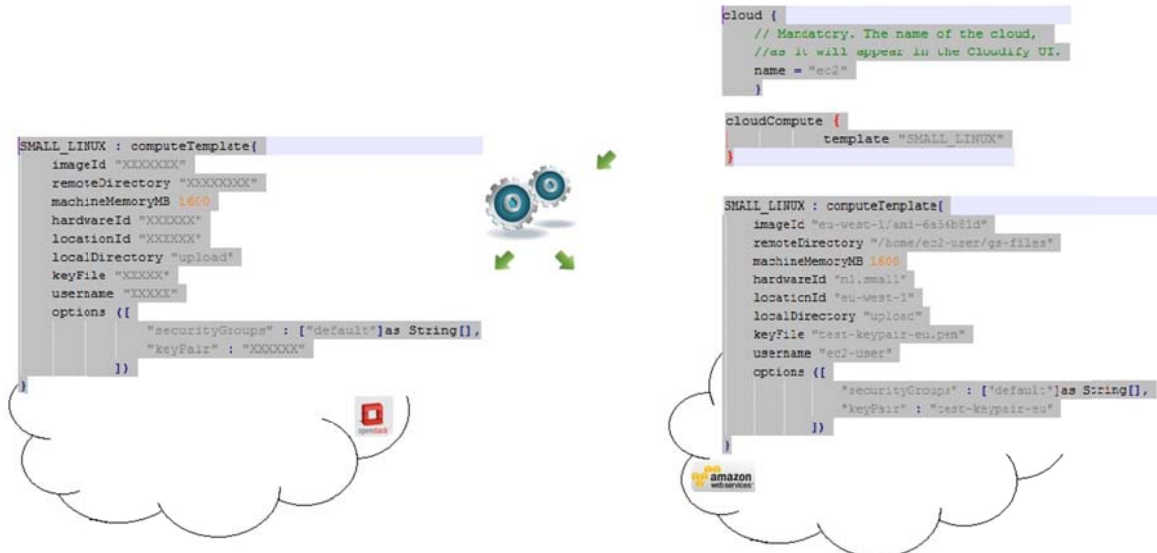


Figure 5-4: Achieving the No Lock-in objective

Full Control: This is achieved using the Cloudfy web management console or the Cloudfy shell by configuring application/service recipes and defining custom monitoring metrics and scaling rules. For our experiment we created several monitoring or scaling metrics. Figure 5-5 shows these monitoring/scaling metrics and shows meters associated with two of them: **Total Request Count** and **(Number of) Active Sessions**. This figure shows how we have achieved the full control objective.



Figure 5-5: Achieving the Full control objective

By configuring and optimizing the Cloudfy controllers or application recipes we can easily achieve a number of desirable features, such as: *automatic self-healing* (If any machine crashes at run time Cloudfy will automatically replace this machine with a new machine, by following the instructions in a recipe) and *Auto Scale- Your way* (By configuring scaling rules in the application recipes we can easily scale in/out our application services based on out-of-the-box or custom metrics). Other features include *any app, any stack and any cloud, automation of*

the entire application lifecycle, and *fully testable on your personal laptop* without hassle of provisioning any VMs.

Cloudify uses a layered architecture that hides (most of) the implementation details and enables a simple deployment process. The layers are called Universal Service Manager (USM), Cloud Controllers, and Cloud Driver. This layering is made possible by Cloudify's Universal Service Adapters that are deployed in every Cloudify provisioned machine. These adapters realize the key features that allow the application to achieve its objectives with no code change, no lock-in, and full control by simply translating recipes into actions with the installation, service initialization, scaling mechanism, and service monitoring. Figure 5-6 shows the architecture of these three layers:

- Universal Service Manager** The USM allows deployment and management of any middleware services in any tier by using an extensible recipe to describe the operations that are to be performed.
- Cloud Controllers** Cloud controllers are the brains of the system. A cloud controller orchestrates the deployment of the application, continuously monitors the application, and triggers alerts and scaling rules based on the values of real-time metrics and loads.
- Cloud Driver** Using the cloud driver it is possible to host provisioning in any virtual environment and abstract the provisioning details from the application.

Using this architecture, it is very easy to write recipes describing all the components required to run any specific application including how to install, start, orchestrate, and monitor the application stack. Recipes use cloud driver configuration files that describes specific machines and images for chosen cloud.

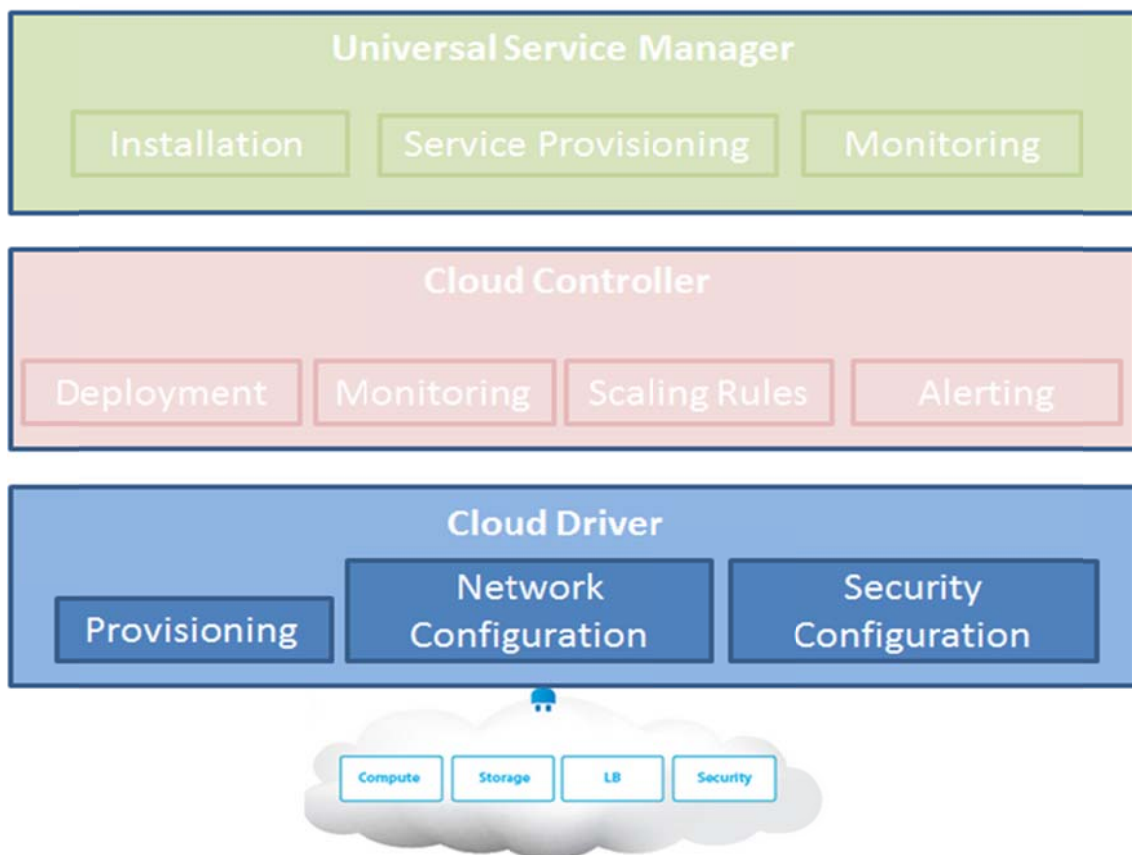


Figure 5-6: Cloudify Architecture

5.4.1 Bootstrapping Cloudfify on EC2

After completing all the EC2 cloud configuration files and compiled Cloudfify as described in the previous section, we now bootstrap Cloudfify by issuing the command below in the Cloudfify shell:

```
# cloudfiy@default>bootstrap-cloud ec2
```

Figure 5-7 shows the bootstrapping process of Cloudfify on EC2. For simulating and troubleshooting purpose we also bootstrapped Cloudfify on the local-cloud of our personal computer.

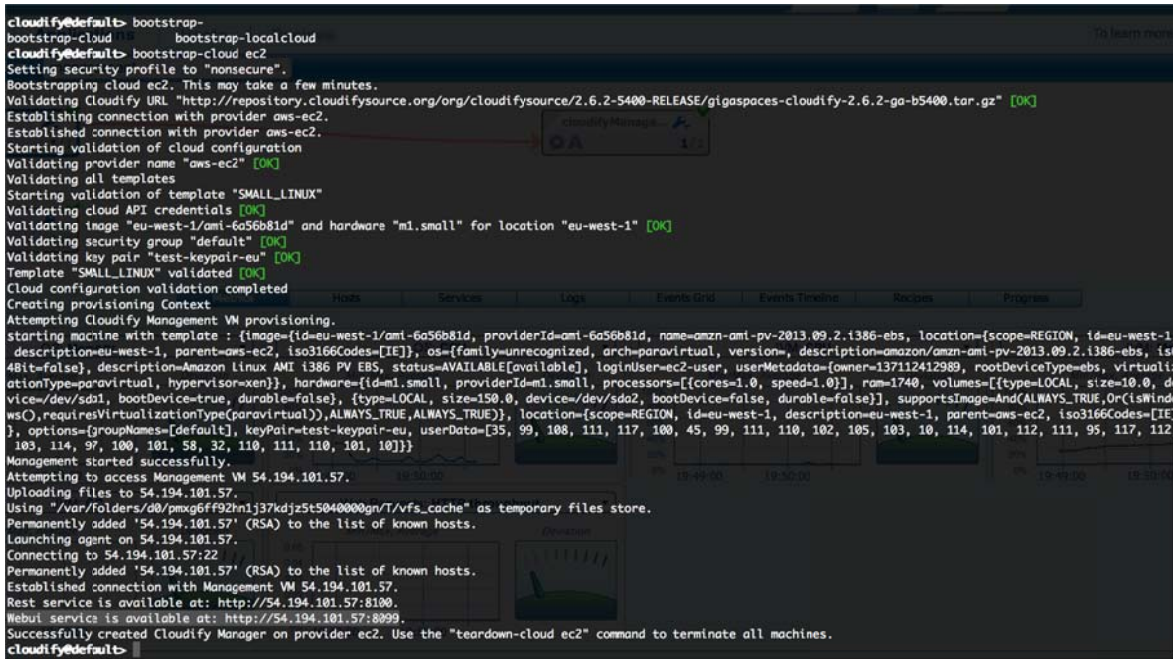


Figure 5-7: Bootstrapping Cloudfify on EC2

Figure 5-8 shows the Cloudfify web management console after we have bootstrapped it on the local cloud.

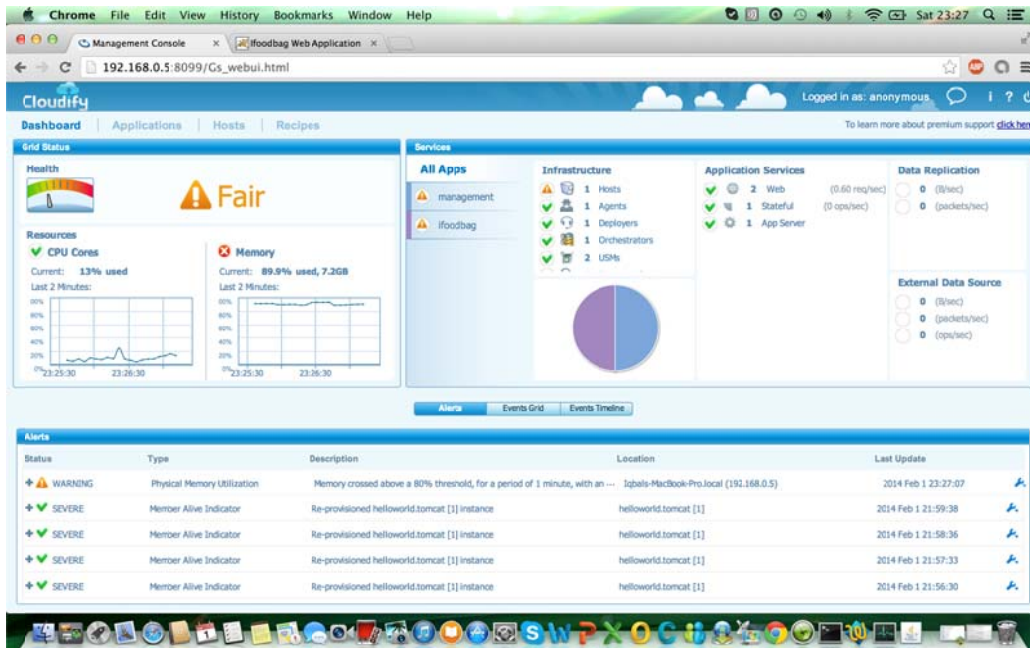


Figure 5-8: Cloudfify Web Management Console

5.5 Motivation for choosing AWS

Rapid changes in technology and business processes over the past decade have created an ongoing IT infrastructure challenge for experts to manage as the infrastructure needs to constantly (or periodically) change. Indeed, the typical business application architecture has completely changed in last ten years, as it evolved from desktop-centric installation to client/server solutions, followed by loosely coupled web services and service-oriented architectures (SOA) and virtualization. Reducing cost and increasing reliability are major issues that must be addressed by enterprise IT. While each evolutionary step built on the previous technology the evolution has introduced new challenges, required changes in strategy, and offers opportunity. Cloud computing has introduced new challenges for the latest IT infrastructure (as discussed in Chapter 2). Amazon Web Services (AWS) [12] provides a complete set of cloud computing services that enable developers to build sophisticated, scalable applications by exploiting a highly reliable and scalable infrastructure to deploy web-scale solutions with minimal support and administration costs, and greater flexibility that available using one's own infrastructure or a datacenter facility [105].

5.5.1 *The differences that distinguish AWS*

The most important features that distinguish AWS from other vendors offering traditional IT computing infrastructures are [105, 106]:

- Flexible** The first key difference between AWS and other IT models is increased flexibility. AWS enables organizations to use resources (such as programming models, OSs, databases, and architectures) that they already have experience with. In addition, flexibility helps organizations to deliver IT solutions when demands arise in order to serve their diverse business needs. Finally, AWS provides flexibility when provisioning new services. Instead of spending time to plan, budget, procure, set up, deploy, operate, and hiring personnel for a new project, you can easily deploy servers on the cloud as and when you need.
- Cost-effective** Cost is one of the most complex elements of delivering contemporary IT solutions. For example, developing and deploying an e-commerce application such as 'Ifoodbag' can be a low budget and successful deployment, followed by cost that track with the number of users and their usage of the application. In contrast, owning and operating your own infrastructure can incur considerable initial cost. The cloud provides an on-demand infrastructure that enables organizations to only consume the resources as they actually need and pay only for the resources that they use, avoiding the need to make any long-term commitments.
- Scalable and elastic** In a traditional IT organization, scalability and elasticity often involved considerable investment in infrastructure. The term 'elasticity' used by AWS means scaling up and down of computer resources to follow business demand. For example, if traffic to a traditional e-commerce shop increases unexpectedly during a short period (for example, during a special offer period), then the administrator needs to be confident that the existing infrastructure can handle this traffic load and that there will not be any interference with normal business operations. In contrast, by using an elastic load balancer and dynamic scaling AWS cloud based resources can automatically be scaled up to meet unexpected demand and then these resources can be scaled down as demand decreases. AWS allows organizations to add or subtract resources to their applications in order to meet customer demand, while managing costs.

Secure AWS ensures the confidentiality, integrity, and availability of your data and promises to maintain your trust and confidence. In order to provide end-to-end security and end-to-end privacy, AWS builds services following security best practices.

Experienced The AWS cloud provides levels of scale, security, reliability, and privacy for an application implemented in Amazon's cloud. AWS has built an infrastructure based on the lessons they have learned from over sixteen years of experience in delivering large-scale infrastructure by following reliable, secure methods.

5.5.2 Introduction to AWS

AWS is a comprehensive cloud service platform that offers compute power, storage, content delivery, and other functionality that organizations can use to deploy applications and services cost effectively with flexibility, scalability, and reliability. Today AWS offers a variety of infrastructure services. The AWS services described in the following subsections are suggested for the implementation of the 'Ifoodbag' cloud infrastructure.

5.5.2.1 Amazon Elastic Compute Cloud (Amazon EC2)

Amazon Elastic Compute Cloud (Amazon EC2) is a web service that allows you to buy compute capacity in the cloud, which is resizable whenever you want. You can create a bundle including an OS, application software, and associated configuration settings as an Amazon Machine Image (AMI). Then you can use these AMIs to provision multiple virtualized instances or decommission them using simple web service calls to scale capacity up and down quickly, as your business operation requirements change. Amazon EC2 is designed to make computing easier for developers and system administrators. It has a simple web service interface that allows creating instance and configuring capacity with minimal friction. Instances can be launched in one or more geographical regions. Each region contains multiple "Availability Zones" with distinct locations. If failures occurs in a specific availability zone, then another availability zone in the same region can provide services via low latency network connectivity [105], [106].

5.5.2.2 Elastic Load Balancing

When traffic grows for an application, it is best practice to **not** allocate all the traffic to a single instance. Instead the incoming traffic should be distributed by an elastic load balancer using the Elastic Load Balancing service. Incoming traffic is automatically distributed across multiple instances through elastic load balancing. This load balancing also enables you to achieve even greater fault tolerance in your applications by providing the necessary amount of load balancing capacity needed according to the current traffic load. The elastic load balancer detects unhealthy instances and automatically reroutes traffic to healthy instances until the unhealthy instances have been restored. Elastic Load Balancing can be enabled within a single availability zone or across multiple zones (for even more consistent application performance).

5.5.2.3 Amazon Virtual Private Cloud (Amazon VPC)

Amazon Virtual Private Cloud lets you create a virtual network as a logically isolated area within the AWS cloud. You can launch resources in this network that you have defined. You can enable IPsec tunnel mode to ensure secure connection between one or more gateways in your data center to a gateway of your VPC. You can allocate your own IP address range and subnets; configure a route table, default route, and network gateways to suit your own preferences – giving you complete control over your virtual networking environment. Changes to the network configuration of your Amazon VPC are easy. For example, you can create a subnet for your web server that has access to the Internet and place your database or application

server in another subnet without Internet access. Amazon also allows you to create a hardware virtual private network (VPN) connection between your corporate data center and your VPC.

5.5.2.4 Amazon ElastiCache

Performance and response time to a request is the most important factor in delivering any IT solution. Amazon ElastiCache is a web service that improves the performance of web applications by allowing you to retrieve information from in-memory caching system. Such in-memory caching systems are faster than disk-based databases. The Amazon ElastiCache service can be used to reduce the overhead associated with data storage infrastructures and also provides a more resilient system that can mitigate the risk of an overload that could result in slow response times to requests. Additionally, ElastiCache provides enhanced visibility of the key performance metrics associated with your memcached or redis nodes. Further details about memcached were given in section 4.4.

5.5.2.5 Amazon Route 53

Amazon Route53 is a highly scalable Domain Name System (DNS) web service that allows you manage all of the DNS records for every domain that you would like to manage. Route53 was designed for organizations to provide an extremely reliable and cost-effective way to route traffic to an application that a user want to access by translating a domain name (such as www.lfoodbag.se) into the IP addresses that the computer user will use to interact with the application. Route 53 is used to connect user requests to the relevant element of an infrastructure running in AWS, such as an EC2 instance, an elastic load balancer, or database. A dynamic DNS is needed since the AWS infrastructure can be scaled up and down, hence services will not have fixed IP addresses.

5.5.2.6 Amazon Elastic Block Storage (EBS)

Amazon Elastic Block Store (EBS) provides block level storage volumes for use with Amazon EC2 instances. EBS is network-attached and the content persists independently from the life of an instance. EBS volumes are exposed as a device within the EC2 instance running on AWS. An EBS volume provides highly available, highly reliable, predictable storage volumes. Amazon EBS particularly well suited for an application that requires a database, file system, or access to raw storage. In addition, snapshots of EBS volumes can be created and stored on Amazon Simple Storage Service (Amazon S3).

5.5.2.7 Amazon Relational Database Service (Amazon RDS)

Amazon Relational Database Service (Amazon RDS) is a web service that makes it easy to set up, administer, and scale a relational database in the cloud. It offers cost-efficient and resizable capacity, while minimizing time consuming database administration tasks, freeing up resources to focus on your application and business. Amazon RDS also allow you to access most well-known databases implemented with MySQL, Oracle, SQL Server, and PostgreSQL. This means that the code, applications, and tools you already use today with your existing databases can be used with Amazon RDS. Amazon RDS automatically patches the database and keeps backups your database, storing the backups for a configured retention period. In addition, Amazon RDS makes it easy to use replication to enhance availability and reliability for databases, while scaling out beyond the capacity of a single database deployment for read-heavy database workloads.

5.5.3 Amazon EC2 instance types

Instances are the primary building blocks in the AWS cloud. Instances are virtual servers that run your application(s). Instances are created from an Amazon Machine Image (AMI). You choose an appropriate instance type to instantiate, depending upon your current business need. An AMI is a template containing a software configuration and OS. You can either use an AMI

provided by AWS or create (and share) your own AMIs. A single AMI can be used to launch as many instances as you want. When you launch an instance of your application, each instance type is associated with different types of hardware offering different capabilities, such as compute, memory, and storage capabilities. Selecting a specific instance type depends on the requirements of the application or software that you want to run on your instances [107].

A large number of instance types are provide by Amazon EC2 for use in different use cases. These instance types define different combinations of CPU, memory, storage, and networking capacity, thus giving you the flexibility to select the appropriate combination of resources for your application.

Table 5-2 lists some of the different instance types that Amazon EC2 provides. Further details about instance types can be found in [108].

Table 5-2: Amazon EC2 instance types

Instance Family	Instance Type	Processor Architecture	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	EBS-Optimized Available	Network Performance
General Purpose	m3.medium	64-bit	1	3	3.75	1 X 4 SSD	-	Moderate
	m3.xlarge	64-bit	4	13	15	2 X 40 SSD	Yes	Moderate
	m1.small	32-bit or 64-bit	1	1	1.7	1X 160	-	Low
	m1.large	64-bit	2	4	7.5	2 X 420	Yes	Moderate
	m1.xlarge	64-bit	4	8	15.	4X 420	Yes	High
Compute optimized	C3.large	64-bit	2	7	3.75	2 X 16 SSD	-	Moderate
	C3.4xlarge	64-bit	16	55	30.	2 X 320 SSD	-	High
	cc2.8xlarge	64-bit	32	88	60.5	4 X 840	-	10 Gigabit
GPU instances	g2.2xlarge	64-bit	8	26	15.	1 X 16 SSD	Yes	High
Memory optimized	m2.lxarge	64-bit	2	6.5	17.1	1 X 420	-	Moderate
	cr1.8xlarge	64-bit	32	88	244.	2 X 120 SSD	-	10 Gigabit
Storage optimized	i2.xlarge	64-bit	4	14	30.5	1 X 800 SSD	Yes	Moderate
	i2.2xlarge	64-bit	8	27	61	2 X 800 SSD	Yes	High
	i2.8xlarge	64-bit	32	104	244.	8 X 800 SSD	Yes	10 Gigabit
Micro	t1.micro	32-bit or	1	Variable	0.615	EBS		Very Low

instances	64-bit	only
-----------	--------	------

5.5.4 Amazon EC2 pricing

One of the primary aims when building an infrastructure in the cloud is cost. As has been stated multiple times users only pay for what they use. There is no minimum fee and no initial investment is required. Prices are based on geographical location and which types of instances you are running. However, pricing also varies with which AMIs that you use. On-demand instances enable you to pay for compute capacity by the hour with no long-term commitments. Table 5-3 shows some of Amazon on-demand instance type costs per hour (in US dollars). We considered the US East (Northern Virginia) region and Linux as the OS. Detailed information can be found in [109].

Table 5-3: Amazon EC2 pricing for Linux OS and US East (N. Virginia) region

Instance Type	vCPU	ECU	Memory (GiB)	Instance Storage (GB)	Linux /UNIX Usage
m3.medium	1	3	3.75	1 X 4 SSD	\$0.113 per hour
m3.xlarge	4	13	15	2 X 40 SSD	\$0.450 per hour
m1.small	1	1	1.7	1X 160	\$0.060 per hour
m1.large	2	4	7.5	2 X 420	\$0.240 per hour
m1.xlarge	4	8	15	4X 420	\$0.480 per hour
c3.large	2	7	3.75	2 X 16 SSD	\$0.150 per hour
c3.4xlarge	16	55	30	2 X 320 SSD	\$1.200 per hour
cc2.8xlarge	32	88	60.5	4 X 840	\$2.400 per hour
g2.2xlarge	8	26	15	60 SSD	\$0.650 per hour
m2.lxlarge	2	6.5	17.1	1 X 420	\$0.410 per hour
cr1.8xlarge	32	88	244	2 X 120 SSD	\$3.500 per hour
i2.xlarge	4	14	30.5	1 X 800 SSD	\$0.853 per hour
i2.2xlarge	8	27	61	2 X 800 SSD	\$1.705 per hour
i2.8xlarge	32	104.	244	8 X 800 SSD	\$6.820 per hour
t1.micro	1	Variable	0.615	EBS only	\$0.020 per hour

5.5.5 EC2 cloud setup for Cloudify

In order to work with Cloudify and launching our application on EC2 clouds, we first setup an EC2 cloud account by:

- First creating an account through Amazon Web Services [110].
- Next we select a Machine Image ID (for our experiment we used “ami-6a56b81d”), Hardware ID (for our experiment we used “m1.small[59]”), Location ID (for our experiment we used the Europe West location “eu-west-1”), and key pairs (including the secret keys for the account).

Figure 5-11, Figure 5-12, and Figure 5-13, shows how to create a key pair, access key ID, and secret access key via the Amazon EC2 management console. These security credentials are used for launching new instances as well as connect to EC2 instances. The AWS security credentials also can be used to verify who you are and whether you have permission to access the resources or not.

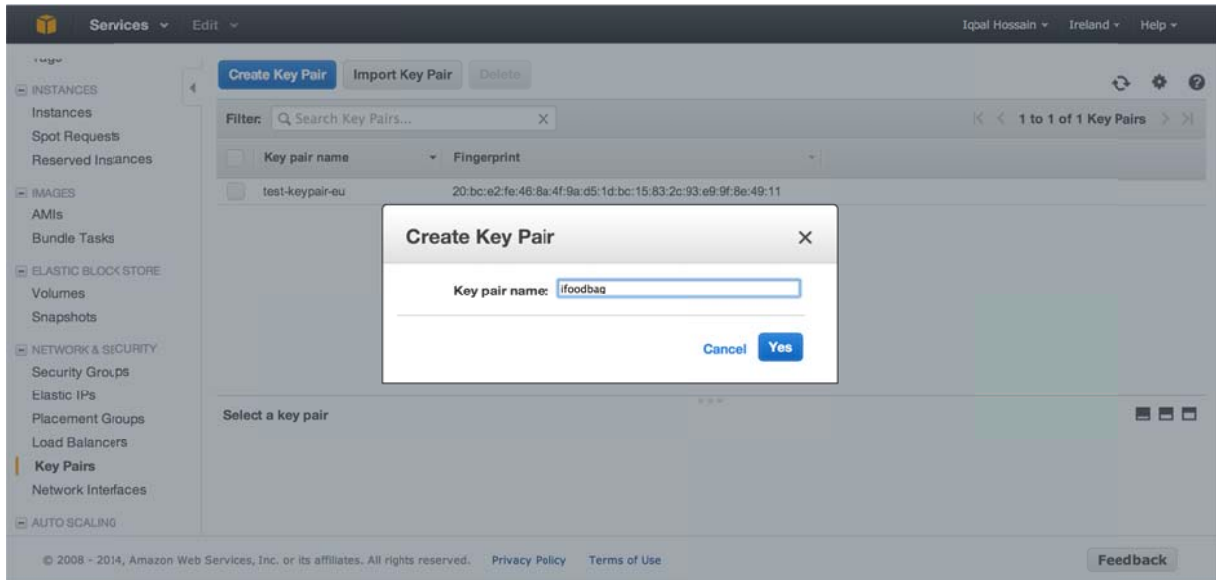


Figure 5-11: Create new a key pair for Amazon EC2

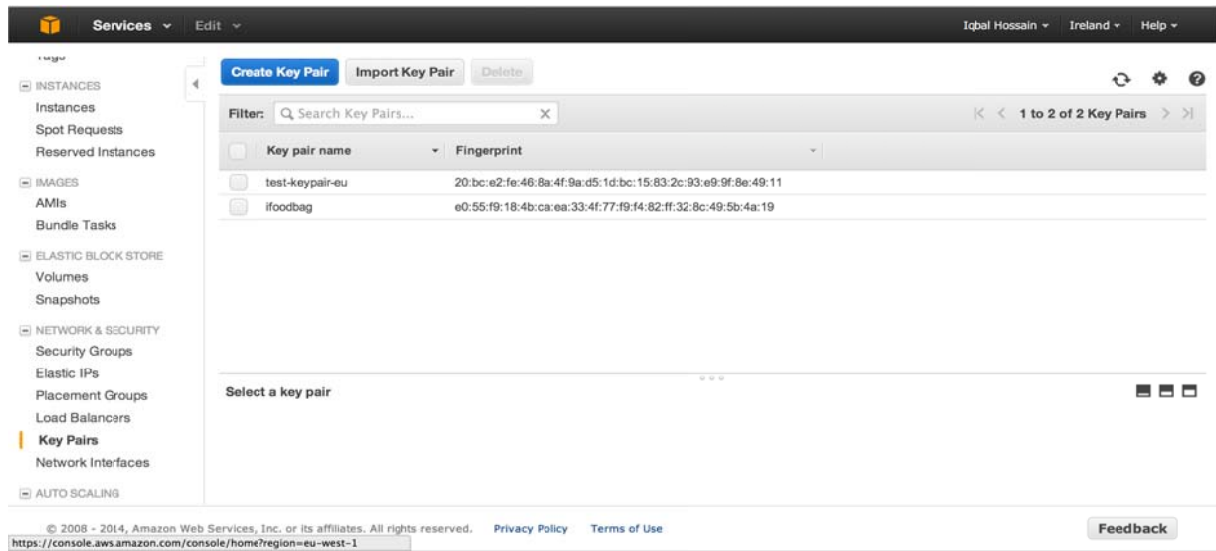


Figure 5-12: Added a new key pair named ifoodbag with a secret key

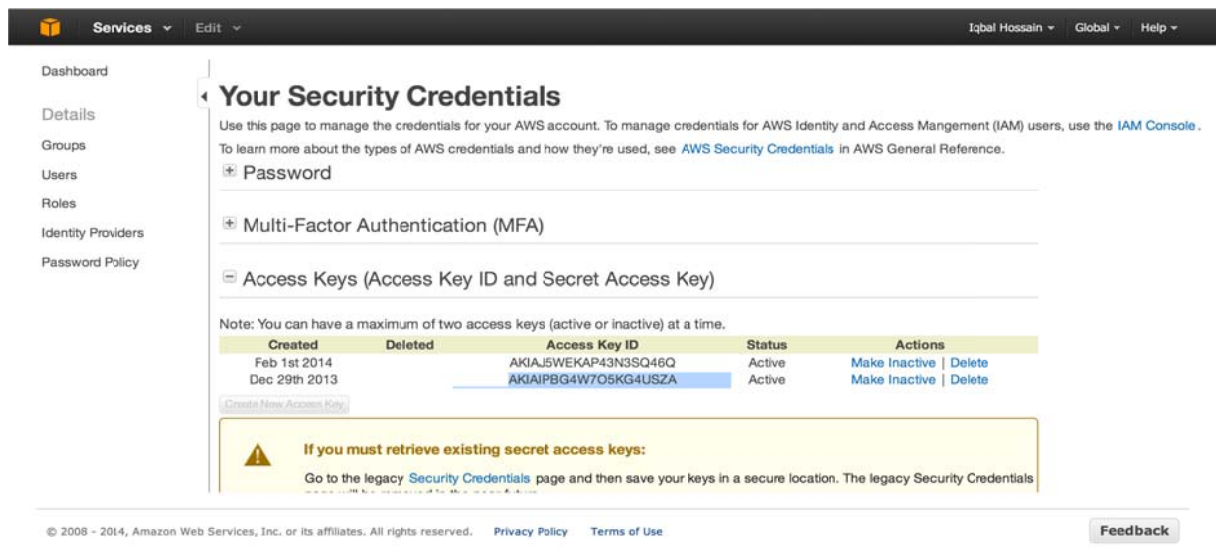


Figure 5-13: Creating an Access Key ID in Amazon EC2

5.6 Webserver load or performance measurement tool - Httperf

“httperf” is a tool for UNIX-like OSs to measure web server performance and to produce a traffic load on a webserver. David Mosberger wrote it initially for Hewlett Packard (HP) [111]. It provides an extensive facility to generate a number of HTTP workloads for measuring webserver performance. The aim of “httperf” is not to produce one particular benchmark, but rather to provide a robust, high performance tool to carryout both micro and macro level benchmarking. The three distinguishing characteristics of httperf are [111, 112]:

- Robustness, which includes the ability to generate and sustain server load,
- Support for the HTTP/1.1 and SSL protocols, and
- Extensibility to new workload generators and performance measurement.

Httperf only tests the standard HTTP payload for your application by default, which means, that it is similar to curl in that it does not load additional assets (images, javascript, or CSS) in the default test. In order to ensure correct results Httperf should be run on the same machine as the web server (to avoid any effects due to networking limitations) and you must ensure that the test tool and the web server are the only processes consuming significant CPU resources on the client machine. The sample command line [113] shown below generates a load on the indicated webserver to measure its performance:

```
httperf --server www.Ifoodbag.se --port 80 --uri /index.php --rate 150 --num-conn 27000 --num-call 1 --timeout 5
```

Following the above command line, httperf sends HTTP get requests for the index.php page to a web service running on the Ifoodbag.se webserver. The maximum number of HTTP requests that should be generated is num-call*rate. The other parameters are:

<i>server</i>	IP address or hostname of the machine where the service is running
<i>port</i>	port the service is running
<i>uri</i>	The context path of the service on the server
<i>rate</i>	Number of connections created per second to make requests to the service.
<i>num-con</i>	Number of test calls made to the service
<i>num-call</i>	Number of calls per TCP connection.
<i>time-out</i>	This is the maximum amount of time that httperf waits for a successful response.

For our experiment we sent the traffic load to our Ifoodbag application launched on EC2 cloud by issuing the following commands.

```
httperf --hog --server 54.194.238.66 --port 8082 --uri /ifoodbag --wsess=5,5,2 --num-conns 1000 --rate 10
```

```
httperf --hog --server 54.194.238.66 --port 8082 --uri /ifoodbag --wsess=20,10,2 --num-conns 10000 --rate 30
```

```
httperf --hog --server 54.194.238.66 --port 8082 --uri /ifoodbag --wsess=20,20,10 --num-conns 20000 --rate 100 --timeout 15
```

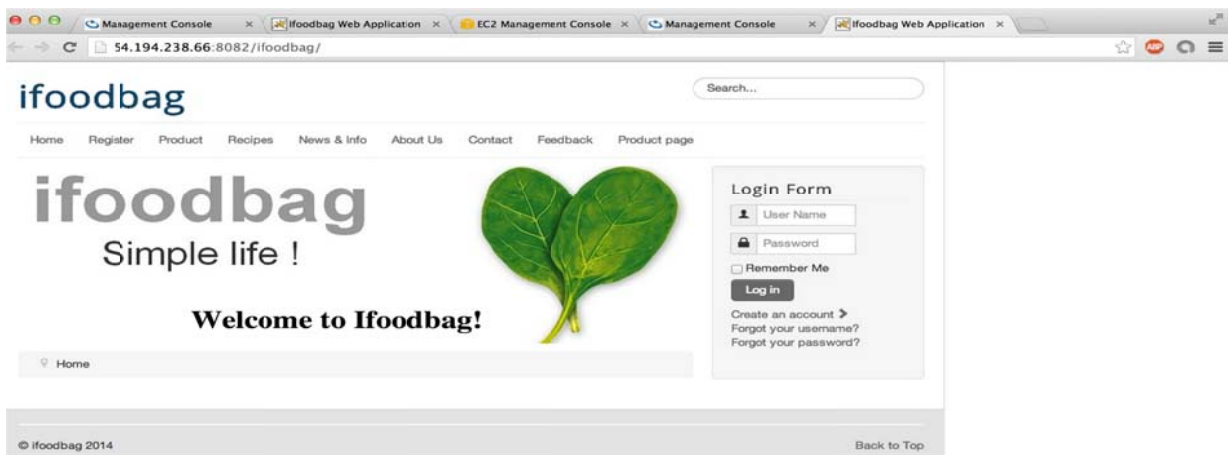
In our experiment, we have created different numbers of sessions with different loads to test the scaling of our servers. Three-parameter are needed to create sessions. The syntax is: --wsess=N1,N2,X. Where, N1: number of sessions, N2: Number of calls per session, and X: delay between calls (sec). The results of these tests are given in section 6.2.

6 Results and analysis

In this chapter, we describe and analyze our findings concerning installing and scaling the Ifoodbag application on the EC2 cloud. We give a cost analysis based on our implemented solution in comparison with traditional infrastructure solutions. Additionally, we compare our proposed solution with some other solutions [3, 4, 21, 114] including some proposed optimizing techniques [115]. Finally the chapter offers some recommendations and guidelines.

6.1 Successful deployment of the Ifoodbag application on EC2 from the management machine

As we explained earlier we started by deploying the Ifoodbag application on a local-cloud running in our personal computer and configured with our resource provisioning and scaling rules in order to verify that this application worked as expected. In this chapter we describe the deployment of the application server in the EC2 cloud using the management machine of our experimental setup as described in our proposed architecture. Figure 6-1 shows the successful deployment of the Ifoodbag application on the EC2 cloud.



Yippe! You've just deployed your first application to the cloud.

Sun Feb 02 03:13:49 UTC 2014

Figure 6-1: Ifoodbag application on EC2 cloud

Figure 6-2 shows the Cloudify web-management console for the Ifoodbag application on EC2 cloud.

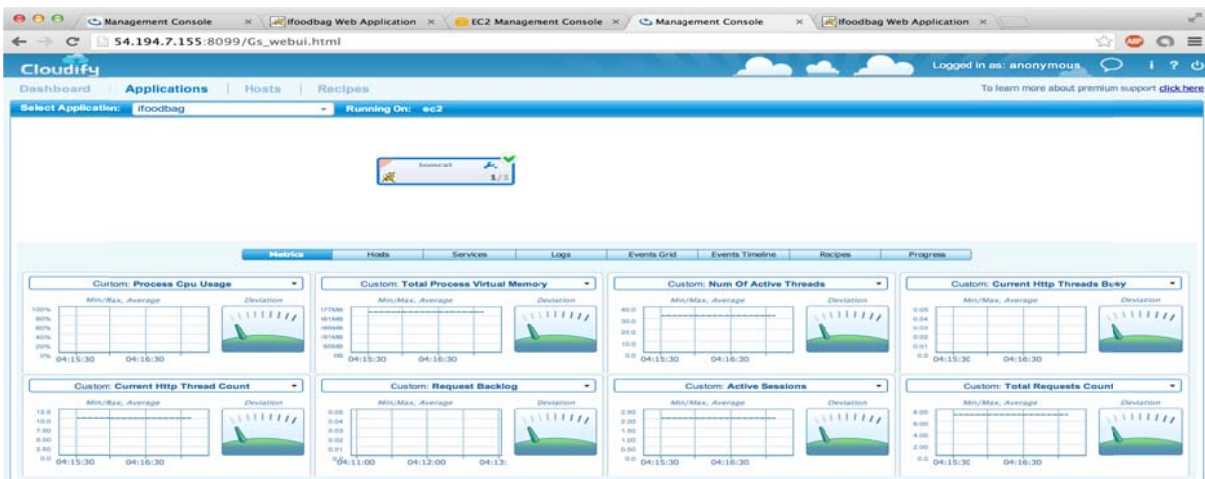


Figure 6-2: Cloudify web-management console for Ifoodbag application

Now that our application has been successfully launched in the cloud architecture, it is time to evaluate the scaling mechanisms, analyze our findings, do cost evaluations, and define recommendations / best practices. We begin in section 6.2 by systematically examining scaling to exploit the elasticity of the cloud.

6.2 Scalability Guidelines and Analysis

In this section we describe the results of our tests and findings concerning (both up and down) scalability mechanisms. To dynamically scale or monitor the ifoodbag application we defined a number of metrics, specifically: Active sessions, Request Counts, CPU/Memory utilization, HTTP thread count, Request Backlog, and etc. Figure 6-3 shows the metrics we have defined for the Ifoodbag application as they are shown in the Cloudify web-management console.

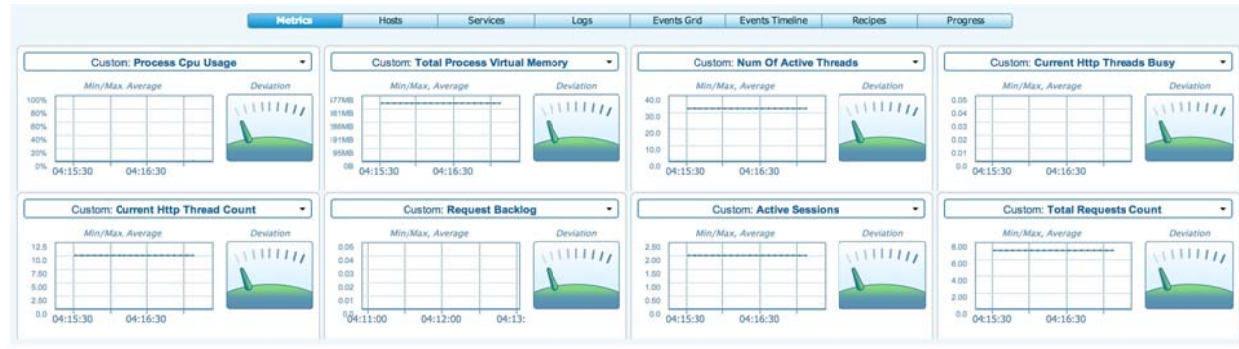


Figure 6-3: Defined metrics for Ifoodbag application

The results presented in this chapter will be based on the “*Total Request Count*” metric. We used the incoming RPS (Request Per Second) metric to drive auto-scaling, as this metric is *independent* of the application but is directly related to throughput. Note that an Auto Scaling action is invoked when the specified metric remains above the specified threshold value for a specified number of time periods as defined in the scaling policy. This is to ensure that a scaling action is not triggered due to a sudden spike in the value of a metric. In addition, we specified a *cooldown* time (using Amazon’s definition of this term - see below) to ensure that a new scaling process will be triggered *only* after completing the previous scaling process.

“Cooldown is the period of time after auto-scaling initiates a scaling activity during which no other scaling activity can take place. A cooldown period allows the effect of a scaling activity to become visible in the metrics that originally triggered the activity. This period is configurable, and gives the system time to perform and adjust to any new scaling activities (such as scale-in and scale-out) that affect capacity.”[116, 117]

6.2.1 Scalability Guidelines

To optimize scaling it is quite important to study and understand the traffic load on the application in a production environment. During our thesis work, Ifoodbag AB also leveraging resources to develop the application, hence it was not ready to go on live, and because of that we could not get access to the live traffic pattern of the Ifoodbag application, we assumed a traffic pattern as illustrated in Figure 6-4. In this pattern the traffic load remains high during the day time and is at a level higher than at night. In order to get this traffic pattern we have simulated this for a 24hrs time frame, where the traffic load represented as request per second in the y-axis and time frame is shown in x-axis. We assumed this traffic pattern, as Ifoodbag is mainly a food delivery application where people will use such application mainly during the day time than at night. However, in the future the live traffic pattern can be examined when Ifoodbag application will be launched to the public.

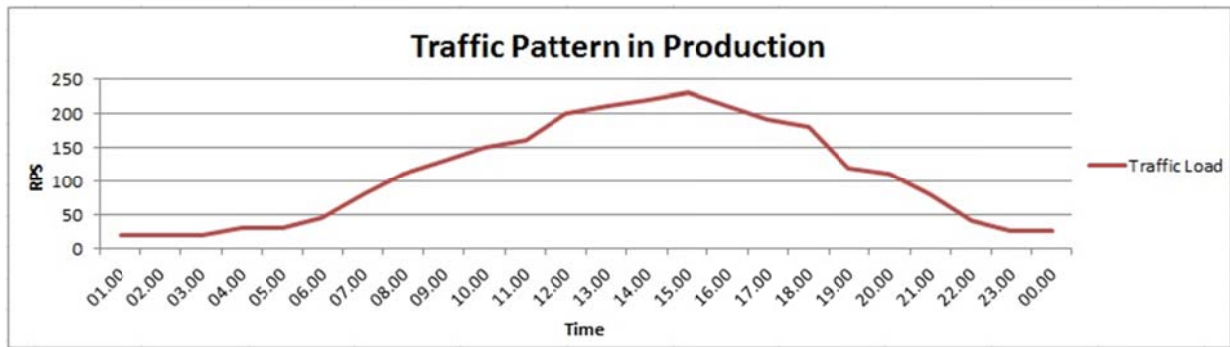


Figure 6-4: Assumed traffic pattern of a production version of the iFoodbag application

Even though this varying nature of traffic patterns makes it more difficult to optimize the exploitation of the elasticity in the cloud environment it gives us some insights, hence it allows us to propose guidelines for the use of scaling techniques. Some of these guidelines include *Avoiding Ping-Pong Effect*; *Being Proactive, Not Reactive*; and *Aggressive Scale Up, Conservative Scale Down*. Each of these will be described below.

- Avoiding Ping-Pong Effect*: To verify our basic scaling mechanism we measured the scaling process for 10 minutes *without* specifying the cooldown timer and optimized metrics threshold value by producing HTTP requests (as described in section 5.6) to the iFoodbag application server. As we have not specified a cooldown time this load resulted in the Ping-Pong effect due to alternating scale up and scale down events, as illustrated in Figure 6-5. This occurs because when the RPS increases it triggers the scaling up event, thus adding a number of new machines (as defined in the scaling policies). However, now that there are more nodes the RPS *per node* decreases and when it falls below the threshold specified for scaling down a scale down event is triggered. Now that there are fewer nodes the RPS per node will increase and the cycle will repeat! From this experience and the recommendations stated in [115, 116] we observed that the Ping-Pong effect can potentially result in increased latency and, in the worst case, may even cause a violation of the service level agreement (SLA) of the service. Hence it is a recommended best practice when defining scaling policies to ensure that the policy is not susceptible to the Ping-Pong effect.

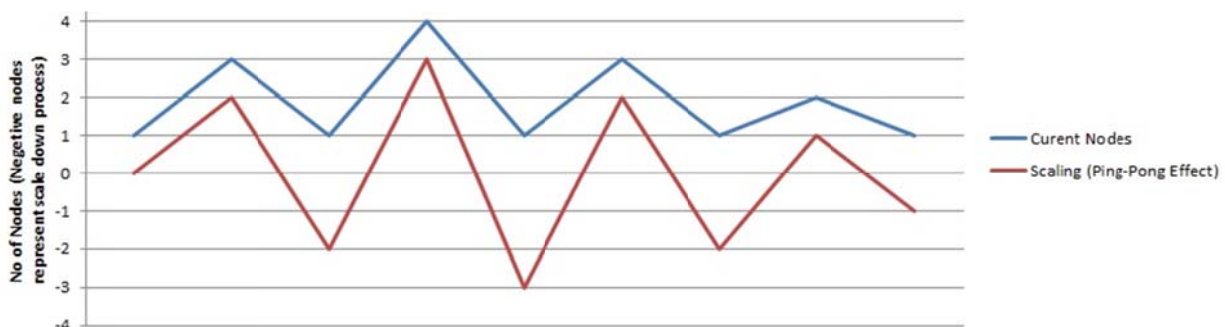


Figure 6-5: Ping-Pong Effect

- Being Proactive, Not Reactive*: During the scaling process we observed that each successful addition of a new application server took ~42 seconds. Table 6-1 illustrates the measurement of the ASG (auto scaling group). With the scaling group size of two VMs it takes 50 seconds to scale up both VMs in comparison with the 42 seconds that it takes to scale up a single VM sized ASG. Table 6-2 illustrates that handling a scale down event takes almost the same time as scaling up. From data shown in both tables we can see that scaling up proactively reduces latency and improves the user experience, by ensuring the ASG is provisioned sufficiently before the time the service latency approaches the limits of the SLA – thus the SLA is never violated! However, proactive scaling up increases the operational cost of the business. This cost will be considered in section 6.3.

Table 6-1: Amazon EC2 Scale up Time

ASG (Scale UP VMs)	Machine Start Time	Machine Ready Time	Total Time To make VMs ready (Seconds)
1	05:05:04	05:05:46	42
2	05:07:40	05:08:30	50
3	05:11:30	05:12:25	55

Table 6-2: Amazon EC2 Scale Down Time

ASG (Scale Down VMs)	Termination Start	Termination End	Successful Termination (Seconds)
1	5.16.15	5.17.02	47
2	5.20.02	5.20.57	55

- Aggressive Scale Up, Conservative Scale Down:* Delivering the best user experience is critical for any business. Hence, we deployed an aggressive scale up policy to be able to handle more than the expected traffic, thus providing a better end-user experience. An aggressive scale up approach provides a buffer for increased traffic during the cooldown period, i.e., having scaled up we will not immediately scale down – hence the system remains in a scaled up state for longer. For this reason an aggressive scale up approach may result in over provisioning and unnecessarily higher operational costs.

In contrast, we use a conservative scale down policy to adapt more slowly (than the historical trend) to decreases in traffic. On the other hand, an aggressive scale down policy may result in accidentally under provisioning, thereby adversely impacting response latency and decreasing throughput (in the worst case, the service may become unavailable). For all of these reasons we have rejected an aggressive scale down policy as it is likely to lead to degraded end-user’s experience. Furthermore, from a corporate standpoint lower throughput would adversely impact the bottom line as poor performance is likely to causes the end-users to take their business elsewhere (this holds true in general for any end-user facing service). Figure 6-6 illustrates the desired the auto-scaling profile when operating in the production environment under the load illustrated in Figure 6-4. For better understanding we placed both figures side by side together as below, where left side figure represents the assumed traffic pattern of a production version of the iFoodbag application and right side figure implicates its desired auto scaling resources.

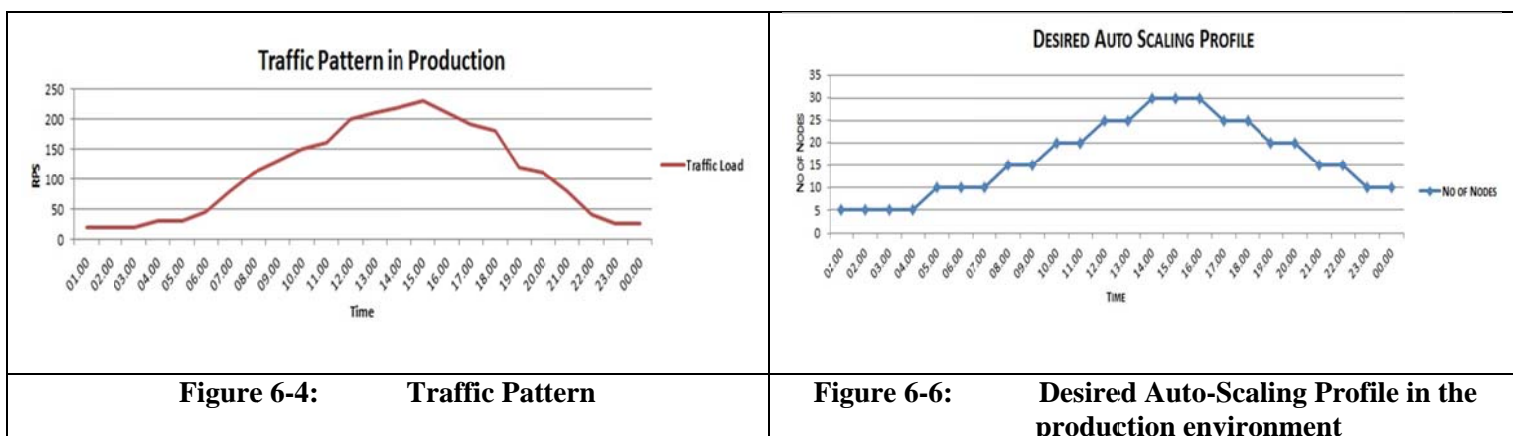


Figure 6-4: Traffic Pattern

Figure 6-6: Desired Auto-Scaling Profile in the production environment

6.2.2 Scalability Analysis

For dynamic or auto scalability it is very important to determine the appropriate threshold value for the selected metric. A low threshold will result in underutilization of the nodes defined in the ASG. On the other hand, a high threshold may result in higher latency, hence degrading the end-user's experience. Therefore, we define the threshold value by considering the throughput corresponding just meeting the SLA for the application. To determine the appropriate threshold value, we consider scaling results based on algorithm number 1 proposed by Kejariwal in [115]. This algorithm must satisfy the following properties in accordance with the guidelines stated in section 6.2.1.

- Property I** RPS per node after scale up should be greater than the scale down threshold (T_D).
This property ensures that a scale up avoids the Ping-Pong effect (see section 6.2.1).
- Property II** RPS per node after scale down should be less than the scale up threshold (T_U).
Similar to Property I, Property II ensures that a scale down avoids the Ping-Pong effect.

Algorithm for scaling Up/Down: As Ifoodbag is just getting started in the market, we tried to define an algorithm that could scale a certain number nodes both upwards and downwards according to the guidelines defined in section 6.2.1. In this algorithm the management system will deploy a *ChangeInCapacity* number of nodes (to scale Up) or decommission this number of nodes (to scale Down). The direction of the change will be based on the *AdjustmentType* during the scaling process. For example, in our experiment the current capacity of the given ASG is 1 and the *ChangeInCapacity* is set to 3. So given a scale up event we will add 3 more nodes to the ASG. The details of the parameter and steps to determine the scaling thresholds (for scaling both up and scaling down) are:

Input: application parameters.

Parameters:

D Scale down value

U Scale up value

T_D Scale down threshold (RPS per node)

T_U Scale up threshold (RPS per node)

N_{\min} Minimum number of nodes in the ASG

Let T (SLA) return the maximum RPS per node for the specified SLA.

$T_U \leftarrow 0.90 \times T$ (SLA)

$T_D \leftarrow 0.50 \times T_U$

Let RPS_{Peak} , RPS_{min} denote the peak and minimum RPS observed for the ASG over the last, say, two weeks

Let N_c , RPS_n denote the current number of nodes and RPS per node respectively

L1: /* Scale Up (if $RPS_n > T_U$) */

repeat

$RPS_{\text{ASG}} \leftarrow N_c \times RPS_n$

$N_c \leftarrow N_c + U$

$RPS_n \leftarrow RPS_{\text{ASG}}/N_c$

until $RPS_n \times N_c \leq RPS_{\text{Peak}}$

L2: /* Scale Down (if $RPS_n < T_D$) */

repeat

$RPS_{\text{ASG}} \leftarrow N_c \times RPS_n$

$N_c \leftarrow \max(N_{\min}, N_c - D)$

$RPS_n \leftarrow RPS_{\text{ASG}}/N_c$

until $RPS_n \times N_c \geq RPS_{\text{min}}$ or $N_c = N_{\min}$

if Properties **I** and/or **II** are not satisfied for each scale up and scale down respectively then

Adjust D , U , T_D , T_U incrementally

Revisit L1 and L2

end if

In this algorithm, the scale down value D and the scale up value U are the main inputs. Kejariwal states that the constants (0.90 and 0.50) used in defining T_U and T_D were determined empirically to minimize the negative impact on end-user experience and minimize ASG underutilization. Loop L1 in the algorithm scaling up an ASG when the incoming traffic increases, while Loop L2 scales down an ASG when the incoming traffic decreases. If Properties I and/or II (defined earlier) are not satisfied then the algorithm adjusts the parameters D , U , T_D , T_U in an incremental fashion and iterates through the loops L1 and L2.

After implementing this algorithm in our Ifoodbag application recipe for about 30m of the simulation in the EC2 cloud when generating traffic load on our application server and using the management machine we obtained the results shown in Table 6-3.

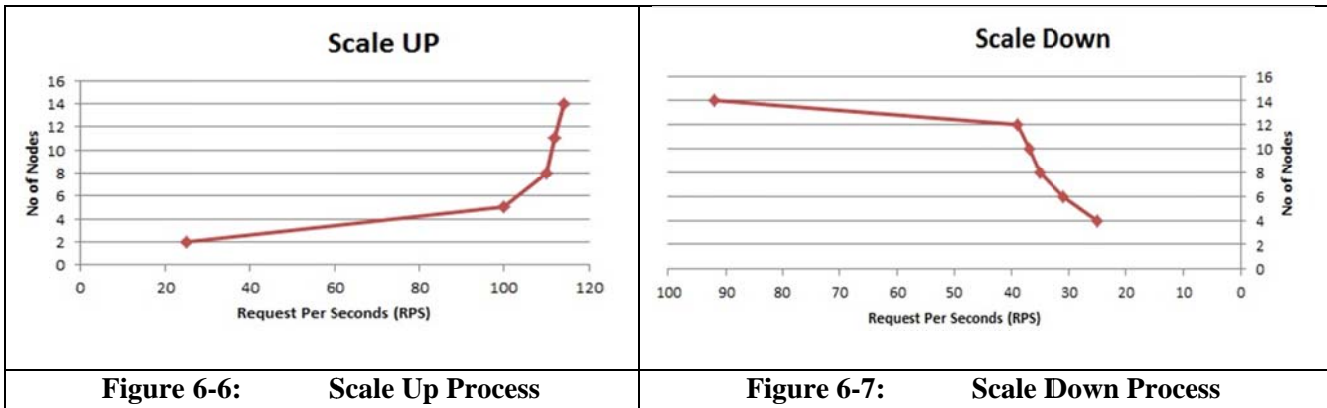
For our experiment, we defined $RPS_{Peak}=1300$, $RPS_{min}=50$, D (scale down value) = 2, U (scale up value) = 3, Scale down threshold $T_D=40$, and Scale up threshold $T_U=80$. In our experiment we initialized RPS_{ASG} to the value 50 and N_c to 2, then we increased RPS_{ASG} to 200 and according to our policy rules since RPS_n approaches T_U ($100 > 80$), an auto-scaling up event is triggered, hence ($U=$) 3 nodes are added to the ASG. Successively the ASG scales up until $RPS_n \times N_c \leq RPS_{Peak}$. Note that column six the New RPS_n value satisfies Property I defined earlier.

Conversely, during the scale down process, we considered $RPS_{ASG}=1300$ and $N_c=14$ and as we simulated to decrease the RPS_{ASG} value to 550, hence RPS_n approaches to T_D ($39 < 40$). Thereby auto-scaling down event triggered and removed ($D=$) 2 nodes from the ASG. Successively the ASG scales down until $RPS_n \times N_c \geq RPS_{Min}$ or $N_c = N_{min}$. It should be noted that in column twelve the New RPS_n value satisfies Property II defined earlier.

Table 6-3: Results of implementing the algorithm with $RPS_{Peak}=1300$, $RPS_{Min}=50$, $D=2$, $U=3$, $T_D=40$, $T_U=80$

Scale Up						Scale Down					
Current Nodes	Nodes Added	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n	Current Nodes	Nodes Removed	RPS_{ASG}	RPS_n	Total Nodes	New RPS_n
2		50	25	2		14		1300	92.85		
		200						550			
	3			5	66.67		2			12	45.83
		550						450			
	3			8	68.75		2			10	45.00
		900						350			
	3			11	81.81		2			8	43.75
		1250						250			
	3			14	89.25		2			6	41.67

Figure 6-6 and Figure 6-7 shows our approach of implementing auto scaling policies both upwards and downwards. When the load increases the RPS value triggers the scaling up event with as scaling up value of 3 as defined in the policy rules. In contrast, when the RPS value decreases it triggers the scaling down event with a scaling down value of 2 as defined in the above mentioned policy rules. This set of parameters ensures the two properties defined earlier.



Although we have demonstrated the policy implementations by using a fixed number of nodes in our scaling rules and the *Total Request Count* metric, Kejariwal [115] presents other algorithms which dimension the size of the change as a percentage of the current capacity and based on average throughput. The choice of algorithm and step size should be based on business requirements, in accordance with different metrics defined in section 6.2. We recommended employing the *Aggressive Scale Up, Conservative Scale Down* approach, as it ensures better end-user experience. Addition, we believe that this approach better adapts the throughput of the application in the cloud architecture to the demand. In the next section we evaluate different cost factors for the proposed cloud architecture to provide a cost analysis comparing a traditional infrastructure with the proposed cloud architecture.

6.3 Cost analysis

Companies need an elastic, reliable, flexible, and low cost infrastructure to provide their services to end-users. Low cost and reliable services are major concerns for businesses (whether the business is a startup or an established business). Fortunately, according to Kondo, et al. a cloud provides scalability, 99.999% reliability, and high performance with a minimum complexity of IT infrastructure [118]. These capabilities are provided by a cloud at relatively low cost when compared to a traditional / dedicated infrastructure. In this project ‘Ifodbag’, a Stockholm based startup company, wants to start their e-commerce business. To make their business cost effective with a highly scalable infrastructure, we propose that they build their infrastructure using a cloud platform. In the following subsection we present (based upon [119]) a comparative cost analysis of realizing this infrastructure via a cloud platform versus traditional infrastructure alternatives.

6.3.1 Utility style pricing for cloud

A cloud can provide a range of services for their users. More over utilizing a cloud allows you to pay for exactly those resources that you have used for your business process. Additionally, you can scale your resources whenever you want as your business needs change, while only needing to pay for these resources when you actually have used them. Table 6-4 described this utility style pricing for a cloud.

Table 6-4: Utility Style Pricing [120, 121]

Pay as you go	You pay on an hourly basis from the time when you launch a resource until the time you terminate it. No long-term contract is required. The cloud replaces your upfront capital expense with a low variable cost as you pay only for what you use in terms of the underlying infrastructure and services that you use.
Pay less when you reserve	You can invest in reserved capacity, but because of the low upfront investment you get a significant discount rate. Depending on the types of instances you reserve, the overall savings ranges between 42% and 71% over on-demand capacity.
Pay even less per unit by using more	You can save even more as your business grows larger. For example, you pay less for per gigabyte as you use more (as the marginal price decreases).
Pay even less as cloud grows	This is the most attractive feature of a cloud. Each cloud provider always tries to reduce their data center and hardware costs, improve their operational efficiencies, and reduce their costs – hence reducing your cost of business.
Custom pricing	If none of pricing model works for your business, then a cloud provider might offer custom pricing for high volume projects with unique requirement.

6.3.2 Cost factors

When owning and operating a data center the most important factor is cost. Therefore there needs to be a detailed and careful analysis before start to build your own infrastructure. In reality it is not as simple as summing up the hardware expenses, as one must consider the utility pricing of resources. Several financial metrics can be used to calculate the Total Cost of Ownership (TCO) which includes both direct and indirect costs of a product or a service. It is challenging to accurately estimate the cost difference between owning an on-premises infrastructure and buying resources in a cloud infrastructure. In this section, we presented a comparative analysis of several different direct costs and indirect costs. Direct costs of ownership can be classified in to different categories, such as hardware cost, assets utilization, power efficiency, data redundancy, security, supply chain management, and personnel costs.

Table 6-5 described these different cost factors in the ownership of an IT infrastructure.

Table 6-5: Different types of cost factors [119, 121]

Infrastructure Costs	Detail Descriptions
Hardware costs	Upfront investment is always critical for enterprises to build an IT infrastructure. The investment required can easily be millions of dollars. Furthermore, expensive ongoing upgrades of resources (servers, storage devices, and load balancers) may be needed on top of the large initial capital investment. In contrast, using a cloud allows you to take advantage of the cloud provider’s purchase of large volumes of hardware at very low marginal cost. Cloud customers enjoy the benefits of this decreased cost to increasing their capacity and performance via enhanced functionality over time.
Asset utilization	Asset or resource utilization is major difference between two models (cloud and traditional). Some research shows that, annually average server utilization in traditional enterprises’ own data center is 5%-20% [34]. If you invest in virtualization and related technology to increase utilization, it is possible to achieve 20%-25% utilization rates. On other hand, when using the cloud’s pay for utility pricing model customers are only charged for resources they actually use, as a result customer can achieve close to 100% utilization.
Power efficiency	Based on numerous industry reports, the average Power Usages Effectiveness (PUE) of a data center is 2.5, thus for every 1 watt of power that is delivered to

Infrastructure Costs	Detail Descriptions
	<p>the servers, 1.5 watts are wasted in heat and other overhead. To build energy efficient dedicated IT infrastructures requires the most efficient equipment and adhering to industry best practices which are frequently prohibitively expensive for enterprises. . If a business attempts to realize their own energy efficient data center they would need to invest heavily to decrease their PUE ratio. However, a cloud infrastructure is likely to invest far more in order to decrease their PUE ratio (since they benefit from this investment with every additional site and rack of equipment), hence they can be far more energy efficient than the average enterprise data center. For example, Facebook’s Prineville, OR datacenter had a PUE of 1.06 at 18:00 GMT on 2014.02.11 (data from https://www.facebook.com/PrinevilleDataCenter/app_399244020173259) while their Forest City, NC Data Center had a PUE of 1.10 at the same time (data from https://www.facebook.com/ForestCityDataCenter/app_288655784601722).</p>
Data redundancy	<p>A highly reliable IT infrastructure requires that you maintain reliable storage & backup devices and operate a reliable redundant network, transit connections, and physical connections between data centers. In order to realize high reliability enterprises need to account for all of these issues and incur the related costs to achieve such as redundant infrastructure. However, utilizing a cloud enables customers to easily deploy servers in multiple availability zones with redundant network facilities, with the cost incurred as operating costs rather than capital costs.</p>
Security	<p>Ensuring security, such as confidentiality, integrity, and availability of business data, is another direct cost of having your own infrastructure. Security costs include purchasing network security devices, security software licenses, smart card for access control, and so on. A cloud can provide these services in keeping with best security practices along with features to provide end-to-end security and end-to-end privacy in conjunction with their cloud platform. Additional details can be found in [63].</p>
Supply Chain Management	<p>In traditional enterprises, cost increases when purchasing hardware because time passes from when hardware is ordered to when it is brought online - often it takes a few months. This long lead time can lead to excess capacity and unnecessarily increased costs. Cloud providers minimize this excess capacity by devoting significant resources to managing their supply chain in conjunction with their large installed hardware base and their continuous (or periodic) expansion of this base.</p>
Personnel	<p>Different IT infrastructure teams are needed to handle heterogeneous hardware and related supply chains, continuously upgrading the data center’s design, operating the data center, scaling and managing physical growth, and so on. All of these personnel costs are necessary in order to achieve low infrastructure costs for each enterprise while these costs can be amortized over a large based in the case of a cloud provider.</p>
Indirect Costs	<p>There are a many indirect costs to build an infrastructure; the result is that enterprises are increasingly attracted to build their infrastructure virtually on top of a cloud platform. Running a large scale and highly availability infrastructure requires highly talented staff and the dedicated attention of management – both of these are areas where the cloud provider has an advantage.</p>

6.3.3 Instance type selection

In this project, we used AWS as our cloud provider and considered AWS’s services costs in our cost analysis. We proposed a cost effective way to save costs when ‘Ifoodbag’ starts to building their infrastructure in cloud. Amazon provides different ways to purchase instances in the cloud. These different types of instances in the AWS cloud offer different cost saving as Reserved Instances, On-Demand Instances, and Spot instances have different prices and time scales for provisioning. The actual functionalities are the same for all of these instance types. Table 6-6 describes these three different instance types according to their potential cost savings [121].

Table 6-6: Types of instances according to costs saving

Reserved Instances	The reserved instances pricing option allows you make a low and one time upfront investment for each instance that you want to reserve. The customer receives a significant discount on their hourly usages charges for these instances and they gain a specific guaranteed capacity. Additionally, you have the flexibility to turn them off when you do not need them, hence you do not even have to pay the discounted hourly rate for those you turn off.
On-demand Instances	The on-demand instance pricing option allows you to purchase an instance by the hour whenever you need without making any long-term commitment. Additionally, you can turn this instance on and off rapidly.
Spot Instances	The spot instance pricing option allows you to bid for unused EC2 compute capacity. The price for spot instances fluctuates depending on the supply and demand for spot instance capacity.

We highly recommended that ‘Ifoodbag’ use at least a minimum number of the reserved instance pricing option instances in their infrastructure because the company can save more by using these types of instances. AWS offers instances depending on the amount of an instance’s resources, described as Light, Medium, and Heavy utilization. If a company needs a consistent service for their users the heavy utilization type of instance is the best option. Table 6-7 shows how much enterprises can save by using reserved instances compared with running on-demand instances. We assume in these computations that the on-demand instances have 100% utilization.

Table 6-7: Saving of reserved instance types over on-demand instances

Reserve Instance type	Saving over On-Demand Instances	
	(1-year)	(3-year)
Light utilization	up to 42%	up to 56%
Medium utilization	up to 49%	up to 66%
Heavy utilization	up to 54%	up to 71%

6.3.4 Total Cost of Ownership (TCO) of running a web application in a cloud

To estimate of total cost we need to consider usage patterns because the actual traffic load can dramatically affect the TCO of a web application. We considered the nature of Ifoodbag’s web application and in this TCO analysis we assume it has a constant level of traffic over time. AWS cloud provides a range of options to reduce costs while flexibility and scalability benefits remain same. In this section we described a comparative costs analysis for Ifoodbag’s web application running in an on-premises infrastructure versus on the AWS cloud platform. We assume that the company wants to deploy its web application for access via the internet to

interact with prospective customers, existing customers, and partners. We assumed the website has hundreds of thousands of visitors every month and is regularly accessed by thousands of customers with a traffic flow that is fairly steady state. The website is a three tier web application with open source content management software stores and serves a large amount of cooking recipes through a content delivery network. To handle this website and provide a good user experience, we assume the following resources are needed [109, 120, 121, 122]:

- 2 Linux based server for web servers
- 2 Linux based application servers
- 2 Linux based MySQL database servers

Table 6-8 compares the TCO of the on-premise alternative costs versus an AWS cloud infrastructure’s costs. Figure 6-8 shows a graphical comparison of the monthly TCO for traditional infrastructures versus a cloud.

Table 6-8: TCO of on-premises infrastructure vs. cloud infrastructure

TCO	Web application infrastructure costs	
	On-Premises	AWS cloud All Reserved (3 year heavy)
Amortized monthly cost over 3 years		
<i>Compute / server costs</i>		
Server Hardware	\$306	\$0
Network Hardware	\$62	\$0
Hardware maintenance	\$47	\$0
Power and cooling	\$172	\$0
Data center space	\$144	\$0
Personnel	\$1200	\$0
AWS instances	\$0	\$429
Total –per month	\$1,931	\$429
Total -3 years	\$69,516	\$15,444
<i>Savings over On-Premises</i>		77%

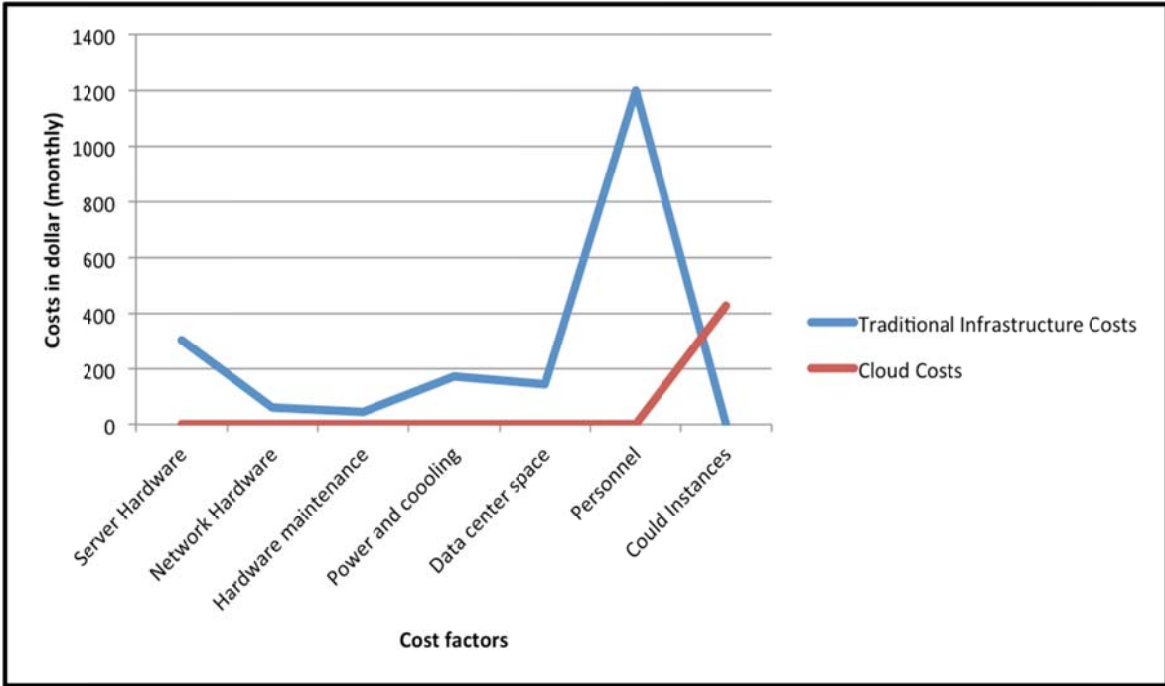


Figure 6-8: Monthly TCO of traditional infrastructure versus cloud

According to our assumptions, we considered the costs of different cost factors for traditional infrastructures based upon market prices. Our assumptions of hardware configurations are as follows. For on-premises infrastructures, we have used prices from [121] and also we assumed equipment and personnel are located in United states. We have assumed that servers are not virtualized without virtualization software licensing and management costs. In this analysis we assumed a total annual cost per person \$120,000 person that is included both salary and benefits. There are significant one-time costs (\$2,492 per server*) when setting up the hardware. For a fair comparison between on-premises versus a cloud, we amortized the one-time costs monthly over a 3 year period. Our hardware assumptions for on premises infrastructure are:

Server hardware	Dell PowerEdge R310 configuration, equivalent of a High-Memory Extra Large (m2.xlarge) Amazon EC2 Instance.
Network hardware	Dell PowerEdge Rack Chassis Dell PowerConnect Switches and a management switch.
Hardware maintenance	3-year Dell ProSupport.
Power and cooling	Power/cooling for 1 server, with a data center PUE of 2.5 and an electricity price of \$0.09 per kW hour.
Data center space	\$23,000 per kW of redundant IT power and \$300 per square foot of space divided by useful life of 15 years.
Personnel	IT infrastructures teams are needed, such as operations staffs for a 24/4/365 facility, database administration teams needed are for managing the MySQL database, and so on.

For Amazon’s cloud platform, we considered Amazon EC2 reserved instances, which are 3 -year heavy utilization. We have chosen high memory extra large (m2.xlarge) EC2 instances for the webserver and application servers and storage optimized (i2.xlarge) EC2 instances for

* We have used prices from [121] in this analysis.

the database server. Since we assumed our website would have a steady state workload we are planning that all of these instances operate 24 hours per day, then a heavy utilization reserved instances is most cost effective option. To calculate the AWS instances costs we used AWS's simple monthly cost calculator [122]. A total of six reserved instances with their costs amortized over a 3 years period are as follows:

2 webservers and 2 application servers	High-memory extra large (m2.xlarge) instances are used in the US East region at a rate of \$0.086 per hour with one time upfront fee of \$1,922.
2 database server	Memory optimized (i2.xlarge) instances are used in the US east region at a rate of \$0.121 per hour with one time upfront fee ~\$2,740.

In table 6-8, we have not added small amount of upfront fee for 3 years reserved instances. In order to calculate total amount of costs for Ifoodbag web application, we need to add upfront fee for with total 3 years monthly cost for these types of instances. The total calculated cost for running the Ifoodbag web application (both compute and database) on reserve instances for 3 years = \$1,5444 (\$429 per month) + \$13,168 (upfront investment for reserved instances) = \$28,612.

Based on our description above, we calculated the total cost of ownership of a traditional versus cloud solution for 5 years period. Figure 6-9 shows the yearly total cost comparison between the traditional infrastructure and cloud platform. Our assumptions for these resources and costs are:

- For first year, we assumed infrastructures needed 2 web servers, 2 application servers, and 2 database servers to serve their users. All hardware configurations same as described above for both platforms. No hardware replacement cost in the first year.
- In the second year, the company would needed to add hardware as their business grows. We assumed 1 webserver, 1 application server, and 1 database server would be added to their server pool. In the cloud we assumed that the first 2 servers of each tier are utilized 100% and the rest of servers will scale according to load. We assumed the remaining servers would be 50% utilized.
- In the third year, we assumed that new hardware would be added as well as replacements of some hardware due to failures or upgrades of the capabilities of the hardware. We assumed 30% additional hardware cost to replace old hardware. New instances would be added to the cloud platform.
- In the fourth year we assumed that 50% of additional hardware would be needed and 50% of the existing hardware would be replaced and that the cloud would be scaled accordingly.
- The traditional infrastructure needs continued upgrades of resources to ensure better performance and we also needed to add new hardware to the infrastructure as traffic to the web application increases. We assumed new hardware is added and 50% of resources are upgraded in the traditional infrastructure. The cloud is assumed to scale its resources to match business needs.

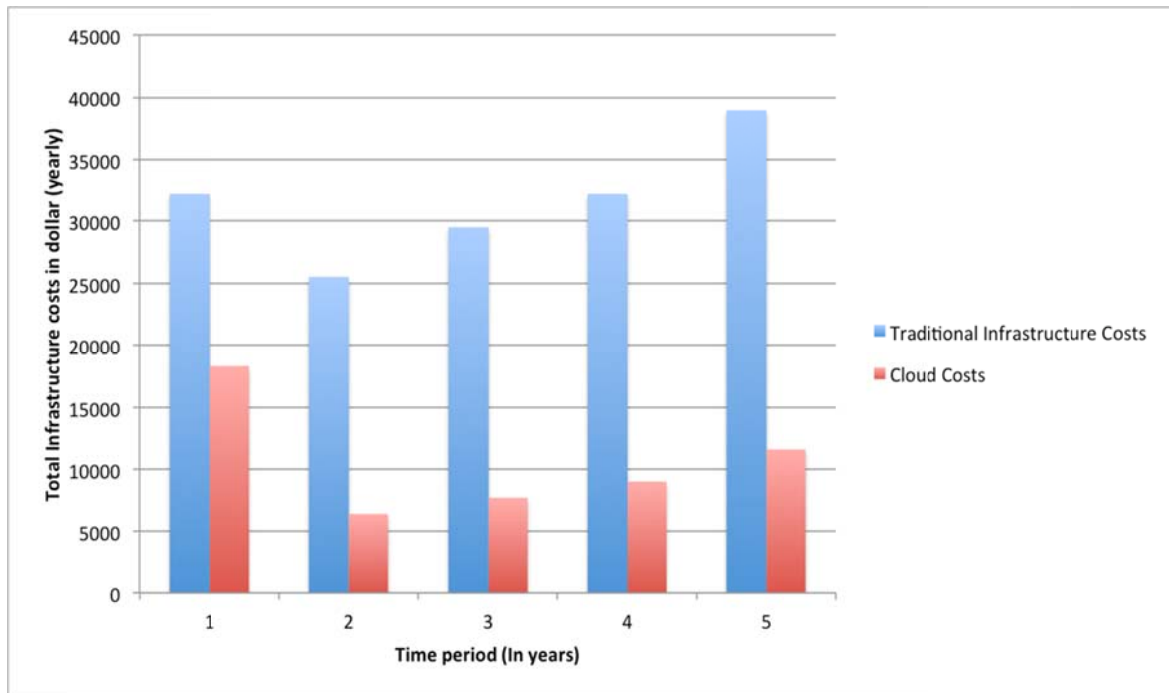


Figure 6-9: Yearly TCO of traditional infrastructure versus a cloud

6.3.5 Cost Analysis Summary

We found that with a cloud you can save 59% of costs in comparison with an on-premises alternative. The maximum benefit and cost effective instances would be achieved by purchasing 3-year heavy utilization reserved instances of Amazon EC2 and Amazon RDS instances. The analysis above seems to indicate that TCO in a dedicated environment can be much higher than the TCO of a cloud. Furthermore, we found large costs differences between traditional and cloud infrastructures over a 5 year period. Traditional infrastructures costs depend on many cost factors that cause the total infrastructure cost to be quite high. One of the major components of this cost is the cost of the personnel to manage and upgrade the traditional infrastructure. In contrast, a cloud allows the customer to deploy resources as need and to scale with business demands. This avoids the extra cost for personnel and upgrades since the cloud providers are responsible for ensuring reliability and performance. Although cloud providers rapidly upgrade their infrastructure for better performance, their costs decrease with time and due to their high volumes of purchases. Companies gain economic benefits by deploying a web application in the cloud while benefiting from a scalable and robust infrastructure in contrast with hosting a web application in an on-premises infrastructure.

6.4 Comparison with some other solutions and some recommendations

As we mentioned in Chapter 1 there are some other solutions [3, 4, 18, 101]. These solutions are mainly proprietary solution with additional licensing and support costs. In contrast our solution is fully open-source based and publicly available. An experienced person with good programming skills and little cloud architecture knowledge* can easily implement and manage their own business application in the cloud architecture. Although the proposed solution is open-source and publicly available, it is also possible to get premium support for Cloudify from a company called GigaSpaces [123].

* Such as can be gained by reading our thesis!

Some recommendations for the deployment of the solution in the production are: for the load-balancing tier it is recommended to use two LBs in order to ensure redundancy and reliability. However, initially it is possible to run the business with a single LB. It is also recommended to run these LBs on m1.large instance types, for details see [59], as these instance types provide 2 virtual cores, 7.5 GB of memory, and a 64-bit platform. Extensive testing confirms that each such LB has the capacity to handle approximately 5,000 requests per second, thus two LBs support a total of about 10,000 requests per second. For our experiments we deployed one management server instance in the cloud but it is recommended to run two instances in a production environment. According to our proposed architectural model we also recommend deploying two stand-alone application server instances in a production environment in order to provide high service availability.

7 Conclusions and Future Work

This chapter describes our achievement in this thesis project in terms of fulfilling our goals (as initially defined). This chapter also suggests further research possibilities to build upon this thesis project. The chapter concludes with some reflections on the social, economic, sustainability, and ethical aspects of this thesis project.

7.1 Conclusions

The aim of this thesis project was to design, implement, demonstrate, and evaluation a highly scalable cloud based architectures for Ifoodbag's web application. This thesis also examined how to dynamically scale the proposed solution both up and down, since for a cloud-based application, especially one which is used by people in a local area, there is going to be a fluctuation of users throughout the day and there is no reason to allocate more or less resources than actually needed. In this we also focused on the fact that a scalable cloud-based architecture can provide great flexibility and enable rapid market share growth for a new business without their need to investing in an expensive infrastructure.

We have achieved our goals as we proposed a scalable cloud architecture model, which clearly provides the dynamic scalability both upwards and downwards. We have demonstrated different guidelines and techniques in order to scale up and down based on RPS (the number of requests per second) at the application server(s). In our scalability policies we have utilized a cooldown time and ASG (Auto scaling group) properties with some predefined parameters, and then clearly shown that the solution satisfies desirable properties such that the RPS per node after a scale up should be more than the scale down threshold (T_d) and RPS per node after scale down should be less than the scale up threshold (T_u). These properties were shown to avoid the Ping-Pong effect during dynamic scaling. This avoidance is a very important consideration in an auto-scaling approach, as otherwise there system could deliver a bad end-user experience and in worst case leads to service unavailability. Furthermore, this dynamic scalability approach also illustrated that it only utilizes resources when needed, avoiding overutilization and underutilization and as a result clearly reduces the operational cost for the business. Our findings and cost analysis also shows that a newly introduced business, such as Ifoodbag AB, can potentially save up to approximate 90% of the upfront investment for the infrastructure setup and save around 50% to 60% of the monthly operational cost for managing applications by using the proposed cloud architecture rather than owning their own on-premises infrastructure.

During this course of project, we have learned various cloud architectures, dynamic scaling mechanisms, and their implementations. The project also gave use very useful experience in working with a cloud environment - as we demonstrated our experiment on the EC2 cloud and deployed our management node using the Cloudify open-source management stack. We have customized different cloud configuration files and developed our own polices in order to deploy Ifoodbag's application in a cloud environment. This thesis project proved to be a very good experience for both of us. Apart from that, we are confident that this thesis project will help us drive our own future career towards cloud technology, as cloud computing is today a very hot topic for the future IT solutions.

7.2 Future Work

Due to the limited time and resources during the course of this thesis project, it was not possible to perform all the tasks defined in our proposed architecture. Moreover, the observations and findings during the course of this thesis project suggest some areas for further research. We plan to carry out some of these tasks in the near future. However, other thesis project students and researchers may also want to explore one or more of these topics.

In this thesis we presented our findings based upon implementing the proposed solution in a single cloud availability zone. However, there is a clear need for empirical testing of the proposed solution across different availability zones in order to find the limits of scalability of the application.

In our demonstration, most of the data and parameter values were based assumptions and the experience & observations from various research papers. In order to get a better results and further optimize the proposed solution, one could study this proposed solution using an actual production environment or at the very least data from a production environment.

In our thesis project, we did not complete the implementation of all the components defined in our proposed architecture. For example, we did not realize the security guidelines defined by Sabrina Ali Tandra and Sarwarul Islam Rizvi in their thesis [52], An obvious further study would implement and evaluate all of the components defined in our proposed architecture including all of the security guidelines that they have proposed.

In our experiment, we evaluated a dynamic scaling mechanism using only one metric (Total Request Count), however other metrics such as CPU/Memory utilization, number of active sessions, etc. or a combination of two or three metrics should be studied to learn what are the most important metrics and how (or if) they should be combined to provide highly dynamic and cost effective scaling of the system for a production version of the web application.

In our demonstration, we deployed the solution only in an EC2 cloud with small instances. Further study and improvements should be made using an implementation of the solution in another public or private cloud using large instances.

7.3 Reflections

This section explores a number of social, economic, legal, and ethical aspects of this thesis project.

7.3.1 *Social aspects*

The proposed infrastructure and dynamic scaling solution could be deployed by any company (not just “Ifoodbag”) to build their infrastructure on a cloud platform. We have made our infrastructure highly scalable, robust, and reliable which ensure 99.99% service availability for the end-users. Our dynamic scaling solution allows enterprises to automatically scale up and down their cloud infrastructure as their traffic changes (and in the best case grows). Having a dynamically scaled infrastructure, enterprises might be able to shift their attention from their service’s availability to concentrate on sales or improve other parts of their business. Because the dynamically scaled infrastructure will ensure service availability during periods of high demand or as traffic increases, Ifoodbag’s users will enjoy smooth, fast, and reliable service that should increase the quality of their user experience. Customers will receive promotions to buy their daily or weekly meals from ‘Ifoodbag’ which will reduce the use of their valuable time of buying food from a grocery store. There are also some social opportunities concerning giving the customer people suggestions about new combinations of foods via new recipes, avoiding unhealthy food, and fostering discussions about healthy food. If any user is unsatisfied with some product, then they can immediately contact the company or authorities to take appropriate action regarding problem.

7.3.2 *Economic aspects*

Cost is always an important concern for enterprises. In our analysis, we proposed a cost effective way to build an IT infrastructure for any business that is considering or using a web application. Such enterprises could save 50%-60% of their monthly operating cost by building their infrastructure using the proposed cloud solution. Furthermore, the analysis of our solution

suggests that enterprises can save even more when using a cloud platform by using 3-year reserve instances as compared to other types of instances in the cloud. We clearly showed the cost difference between traditional infrastructures and the proposed cloud solution. By deploying web application in the cloud companies can avoid the need to make a large initial investment in IT infrastructure, this money can instead be used by the company to improve their product(s) and/or increase the number of customers by investing in marketing. In the proposed cloud platform, there is no maintenance, upgrade, or capital hardware cost. Companies can deploy instances on new hardware as they needed without making any upfront investment. The cloud platform is cost effective, highly scalable, robust, and reliable that can provide a highly available service, which can benefit many enterprises.

7.3.3 Sustainability aspects

Adopting a best practice cloud based solution can have a significant impact on improving the sustainability of the business in terms of reducing electrical power consumption, as was described section 6.3.2 in terms of the greatly improved PUE that Facebook and Google have shown in comparison to on-premises best practice data centers. These same firms have shown that additional savings can be realized in terms of improved water usage effectiveness (thus decreasing the data center's needs for water).

7.3.4 Legal and ethical aspects

We have used information that was open to the public in our thesis work. We ensured that no commercially sensitive information was revealed or used in our work. The applications and tools that we have used are all open source, free, and publically available under GNU General Public License [124] or similar license. We used the Amazon cloud to perform our experiments. The use of this cloud was paid for by Ifoodbag. We created our own strategy to perform these experiments. The experiment results were not fabricated and sufficient details are provided in the thesis and the appendices to allow others to replicate our results. Additionally, our experimental data are available to others upon request. We have proposed an architecture to build and IT infrastructure for 'Ifoodbag', but it could also be used by other enterprises to make their web application service more reliable and to increase the quality of their end-users' experience when using this service. We did not explore the question of the existence of any requirements to disclose business or customer information to governmental authorities (for example for regulator or law enforcement purposes) in our thesis project, hence this remains for future work.

References

- [1] L. M. Vaquero, L. Rodero-Merino, and R. Buyya, "Dynamically scaling applications in the cloud," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 41, no. 1, pp. 45–52, 2011.
- [2] A. Draganov, "Exploiting Private and Hybrid Clouds for Compute Intensive Web Applications," Master's thesis, The University of Edinburgh, August 2011. Available: <https://www.epcc.ed.ac.uk/sites/default/files/Dissertations/2010-2011/AleksandarDraganov.pdf>
- [3] Amazon, "Auto Scaling." [Online]. Available: <http://aws.amazon.com/autoscaling/>. [Accessed: 17-May-2013].
- [4] "Cloud Management for Public and Private Clouds by RightScale." [Online]. Available: http://www.rightscale.com/?utm_expid=3535964-35. [Accessed: 17-May-2013].
- [5] "Cloudify - Deploy a Multi-Tier App on EC2." [Online]. Available: http://www.cloudifysource.org/guide/2.5/qsg/quick_start_guide_ec2. [Accessed: 17-May-2013].
- [6] Nakul E. Sibiraj, "Managing the Cloud with Open Source Tools," *Comput. Sci. Eng. Univ. Calicut*, A seminar report for a Bachelor of Technology, 2011. Available from: <http://www.slideshare.net/nakule/seminar-report-managing-the-cloud-with-open-source-tools>
- [7] J. Jung, B. Krishnamurthy, and M. Rabinovich, "Flash crowds and denial of service attacks: characterization and implications for CDNs and web sites," in *Proceedings of the 11th international conference on World Wide Web*, New York, NY, USA, 2002, pp. 293–304.
- [8] "Alzheimer Europe - Research - Understanding dementia research - Types of research - The four main approaches." [Online]. Available: <http://www.alzheimer-europe.org/Research/Understanding-dementia-research/Types-of-research/The-four-main-approaches>. [Accessed: 13-Feb-2014].
- [9] S. Khandani, "Engineering Design Process," Aug. 2005. Available: <http://www.saylor.org/site/wp-content/uploads/2012/09/ME101-4.1-Engineering-Design-Process.pdf>
- [10] "Research Methods/Types of Research - Wikibooks, open books for an open world." [Online]. Available: http://en.wikibooks.org/wiki/Research_Methods/Types_of_Research. [Accessed: 13-Feb-2014].
- [11] Salesforce.com, "What is Cloud Computing Technology? - salesforce.com." [Online]. Available: <http://www.salesforce.com/cloudcomputing/>. [Accessed: 12-May-2013].
- [12] Amazon, "What is Cloud Computing by Amazon Web Services | AWS." [Online]. Available: <http://aws.amazon.com/what-is-cloud-computing/>. [Accessed: 07-May-2013].
- [13] US Department of Commerce, "Final Version of NIST Cloud Computing Definition Published." [Online]. Available: <http://www.nist.gov/itl/csd/cloud-102511.cfm>. [Accessed: 12-May-2013].
- [14] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities," in *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, 2008, pp. 5–13.
- [15] Cisco Systems, "Cloud Computing - Overview." [Online]. Available: <http://www.cisco.com/web/solutions/trends/cloud/index.html>. [Accessed: 07-May-2013].
- [16] J. Hurwitz, R. Bloor, M. Kaufman, and F. Halper, "What Is Cloud Computing? - For Dummies." [Online]. Available: <http://www.dummies.com/how-to/content/what-is-cloud-computing.html>. [Accessed: 07-May-2013].
- [17] Heather Boothe, "The Difference Between Cloud Computing and Virtualization," Blog, 19-Feb-2013. [Online]. Available: <http://www.virtualcommand.com/virtualization-cloud-computing-difference/>. [Accessed: 12-May-2013].
- [18] Joe Schulz, "Key Features Of Cloud Computing, Blog, CloudTweaks," 03-Sep-2012. [Online]. Available: <http://www.cloudtweaks.com/2012/09/key-features-of-cloud-computing/>. [Accessed: 12-May-2013].
- [19] V. Delgado, "Exploring the limits of cloud computing," Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, TRITA-ICT-EX-2010:277, November 2010. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-27002>

- [20] J. Hur, R. B. M. Kaufman, and F. Ha, "Cloud Computing Characteristics - For Dummies." [Online]. Available: <http://www.dummies.com/how-to/content/cloud-computing-characteristics.html?cid=embedlink>. [Accessed: 12-May-2013].
- [21] Joyent Inc, "Performance and Scale in Cloud Computing", White Paper, <http://www.joyent.com/content/06-developers/01-resources/07-performance-and-scale-in-cloud-computing/performance-scale-cloud-computing.pdf>
- [22] W. Lau, "An Introduction to Cloud Computing Characteristics and Service/Deployment Models | Cloud Zone," 16-May-2012. [Online]. Available: <http://cloud.dzone.com/articles/introduction-cloud-computing>. [Accessed: 12-May-2013].
- [23] D. Wischik, M. Handley, and M. B. Braun, "The Resource Pooling Principle," *SIGCOMM Comput Commun Rev*, vol. 38, no. 5, pp. 47–52, Sep. 2008.
- [24] D. M. Surgient, "The five defining characteristics of cloud computing," *ZDNet*. 9 April 2009 [Online]. Available: <http://www.zdnet.com/news/the-five-defining-characteristics-of-cloud-computing/287001>. [Accessed: 12-May-2013].
- [25] J. Medaugh, "How Powerful APIs Leverage Cloud Computing," *Enterprise Cloud Blog*. 8 October 2013 [Online]. Available: <http://www.terremark.com/blog/powerful-apis-leverage-cloud-computing/>. [Accessed: 13-Feb-2014].
- [26] D. Petcu, C. Craciun, and M. Rak, "Towards a Cross Platform Cloud API - Components for Cloud Federation," Proceedings of the 1st International Conference on Cloud Computing and Services Science (CLOSER 2011), Noordwijkerhout, Netherlands, 7-9 May, 2011. SciTePress 2011, ISBN 978-989-8425-52-2, pp. 166–169, 2011.
- [27] S. Lindskog, *Modeling and tuning security from a quality of service perspective*. Doctoral dissertation, Chalmers University of Technology, Institutionen för data- och informationsteknik, Göteborg, Sweden, ISBN 91-7291-578-1, 2005. Available: http://www.cs.kau.se/~stefan/publications/PhD05/full_text.pdf
- [28] N. Limrungsi, J. Zhao, Y. Xiang, T. Lan, H. H. Huang, and S. Subramaniam, "Providing reliability as an elastic service in cloud computing," in *Communications (ICC), 2012 IEEE International Conference on*, 2012, pp. 2912–2917.
- [29] A. Huth and J. Cebula, "The Basics of Cloud Computing," United States Computer Emergency Readiness Team, 2011. Available: <http://www.us-cert.gov/sites/default/files/publications/CloudComputingHuthCebula.pdf>
- [30] B. L. Sahu and R. Tiwari, "A Comprehensive Study on Cloud Computing," *Int. J.*, vol. 2, no. 9, 2012.
- [31] W. Voorsluys, J. Broberg, and R. Buyya, "Introduction to Cloud Computing," in *Cloud Computing*, R. Buyya, J. Broberg, and A. Goscinski, Eds. John Wiley & Sons, Inc., 2011, pp. 1–41.
- [32] C. Barnatt, "Cloud Computing: ExplainingComputers.com." Blog, Last modified 13 September 2012 [Online]. Available: <http://explainingcomputers.com/cloud.html>. [Accessed: 12-May-2013].
- [33] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, and M. Zaharia, "Above the Clouds: A Berkeley View of Cloud Computing," University of California, EECS Department, Berkeley, California Technical Report No. UCB/EECS-2009-28, 10 February 2009. Available: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html>
- [34] A. Fox, R. Griffith, A. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, and I. Stoica, "Above the clouds: A Berkeley view of cloud computing," *Dept Electr. Eng Comput Sci. Univ. Calif. Berkeley Rep UCBECS*, vol. 28, 2009.
- [35] "Defining Cloud Computing's Key Characteristics, Deployment and Delivery Types | Tek-Tips Whitepaper Library." [Online]. Available: <http://tek-tips.nethawk.net/defining-cloud-computings-key-characteristics-deployment-and-delivery-types/>. [Accessed: 12-May-2013].
- [36] "Gartner IT Glossary - Virtualization." [Online]. Available: <http://www.gartner.com/it-glossary/virtualization/>. [Accessed: 12-May-2013].
- [37] B. Hill, "Virtualization - Beginner's Guide," 12-Mar-2012. [Online]. Available: <http://www.petri.co.il/intro-to-virtualization.htm>. [Accessed: 12-May-2013].

- [38] VMware, Inc, “Understanding Full Virtualization, Paravirtualization, and Hardware Assist.” 10 November 2007. Available: http://www.vmware.com/files/pdf/VMware_paravirtualization.pdf
- [39] “Kernel Based Virtual Machine (KVM).” [Online]. Available: http://www.linux-kvm.org/page/Main_Page. [Accessed: 12-May-2013].
- [40] Red Hat, Inc., “RED HAT ENTERPRISE VIRTUALIZATION FOR SERVERS 2.2: FEATURE COMPARISON,” 2011. Available: http://www.redhat.com/f/pdf/rhev/final2.2/DOC103_RHEV_FeatureMatrix_3073747_0610_ma_web.pdf
- [41] CloudStack, “8.1. KVM Hypervisor Host Installation.” [Online]. Available: http://cloudstack.apache.org/docs/en-US/Apache_CloudStack/4.0.0-incubating/html/Installation_Guide/hypervisor-kvm-install-flow.html. [Accessed: 05-Aug-2013].
- [42] “LAMP (Software Bundle), Why LAMP (Linux Apache MySQL PHP) is the best.” [Online]. Available: <http://linuxsolutions.org.in/lamp.html>. [Accessed: 12-May-2013].
- [43] Red Hat Inc., “Scaling the LAMP Stack in a Red Hat Enterprise Virtualization Environment,” Aug. 2009. Available: <http://www.redhat.com/rhecm/rest-rhecm/jcr/repository/collaboration/jcr:system/jcr:versionStorage/54a4560b0a070d5442cedf28799bff35/1/jcr:frozenNode/rh.resourceFile>
- [44] Bodvoc Ltd., “An Overview of a Web Server,” | Bodvoc’s Blog, 02-Jul-2010. [Online]. Available: <http://bodvoc.wordpress.com/2010/07/02/an-overview-of-a-web-server/>. [Accessed: 12-May-2013].
- [45] C. Janseen, “What is Amazon Web Services (AWS)? - Definition from Techopedia,” *Techopedia.com*. [Online]. Available: <http://www.techopedia.com/definition/26426/amazon-web-services-aws>. [Accessed: 12-May-2013].
- [46] Amazon, “What is Amazon Web Services? - Getting Started with AWS.” [Online]. Available: <http://docs.aws.amazon.com/gettingstarted/latest/awsgsg-intro.html>. [Accessed: 12-May-2013].
- [47] Salesforce.com, Inc., “Force.com: A Comprehensive Look at the World’s Premier Cloud-Computing Platform.” Whitepaper, 2009. Available: http://www.developerforce.com/media/Forcedotcom_Whitepaper/WP_Forcedotcom-InDepth_040709_WEB.pdf
- [48] “OpenNebula - Open Source Data Center Virtualization.” [Online]. Available: <http://opennebula.org/about:technology>. [Accessed: 12-May-2013].
- [49] D. Kaur, K. Kaur, and S. Dilbag Singh, “Evaluating performance of web services in cloud computing environment with high availability,” *Glob. J. Comput. Sci. Technol.*, vol. 12, no. 11-B, 2012.
- [50] T. C. Chieu, A. Mohindra, A. A. Karve, and A. Segal, “Dynamic Scaling of Web Applications in a Virtualized Cloud Computing Environment,” in *IEEE International Conference on e-Business Engineering, 2009. ICEBE '09*, 2009, pp. 281–286.
- [51] C.-L. Hung, Y.-C. Hu, and K.-C. Li, “Auto-Scaling Model for Cloud Computing System,” *Int. J. Hybrid Inf. Technol.*, vol. 5, no. 2, 2012.
- [52] A. Wolke and G. Meixner, “TwoSpot: A Cloud Platform for Scaling Out Web Applications Dynamically,” in *Towards a Service-Based Internet*, E. D. Nitto and R. Yahyapour, Eds. Springer Berlin Heidelberg, 2010, pp. 13–24.
- [53] Z. Micskei, “Dynamically Scalable Applications Cloud Environment,” Dissertation, Budapest University of Technology and Economics, Budapest, Hungary, Available: http://mit.bme.hu/~micskeiz/education/onlab/siklosi_zsolt/siklosi-zsolt_dynamically-scalable-applications-in-cloud-environment.pdf
- [54] “ifoodbag.” [Online]. Available: <http://www.ifoodbag.se/>. [Accessed: 18-May-2013].
- [55] D. Occhipinti, “Building Scalable Web Sites – Scalability | Linux, PHP, LAMP, The Web in a blog.” Blog, 20 January 2009 [Online]. Available: <http://www.danieleocchipinti.com/blog-linux-php-lamp-web/linux/linux-command-line/building-scalable-web-sites-scalability>. [Accessed: 18-May-2013].

- [56] Oracle, "Scaling WikiPedia with LAMP: 7 billion page views per month (Alka Gupta's Cloud)." [Online]. Available: https://blogs.oracle.com/WebScale/entry/scaling_wikipedia_with_lamp_7. [Accessed: 18-May-2013].
- [57] "HAProxy - The Reliable, High Performance TCP/HTTP Load Balancer." Last modified 17 December 2013. [Online]. Available: <http://haproxy.1wt.eu/>. [Accessed: 18-May-2013].
- [58] Amazon, "Amazon Elastic Compute Cloud (Amazon EC2), Cloud Computing Servers." [Online]. Available: <http://aws.amazon.com/ec2/>. [Accessed: 18-May-2013].
- [59] Amazon, "Amazon EC2 Instances." [Online]. Available: <http://aws.amazon.com/ec2/instance-types/#selecting-instance-types>. [Accessed: 18-May-2013].
- [60] J. Leishman, B. Robison, and J. Taylor, "Memcached." . Available: http://xecanson.jp/memcached/memcached_BestDoc_English.pdf
- [61] "OpenNode – About." [Online]. Available: <http://opennodecloud.com/about/>. [Accessed: 19-May-2013].
- [62] "Overview of Eucalyptus." [Online]. Available: http://www.eucalyptus.com/docs/3.2/ag/euca_oview.html. [Accessed: 12-May-2013].
- [63] S. Ali Tandra and S. Islam Rizvi, "Security for cloud based services." Master's thesis, KTH Royal Institute of Technology, School of Information and Communication Technology, Stockholm, Sweden, January-2014. Available: <http://urn.kb.se/resolve?urn=urn:nbn:se:kth:diva-140601>
- [64] "How the domain name system works - Bravenet Wiki." [Online]. Available: http://wiki.bravenet.com/How_the_domain_name_system_works. [Accessed: 16-Jan-2014].
- [65] "What is domain name system (DNS)? - Definition from WhatIs.com." [Online]. Available: <http://searchnetworking.techtarget.com/definition/domain-name-system>. [Accessed: 16-Jan-2014].
- [66] R. Chandramouli and S. Rose, "Secure domain name system (DNS) deployment guide," *Recomm. Natl. Inst. Stand. Technol.*, 2009.
- [67] "How DNS Works-The Pharming Guide - Whitepapers - www.technicalinfo.net." [Online]. Available: <http://www.technicalinfo.net/papers/Pharming2.html>. [Accessed: 17-Jan-2014].
- [68] "DNS Amplification Attack - Nirlog.com - Technology, Life and other stuff that come along...." [Online]. Available: <http://nirlog.com/2006/03/28/dns-amplification-attack/>. [Accessed: 17-Jan-2014].
- [69] A. Kotelnikov, "Doman Name Server (DNS) : Sequence of DNS lookups." Lecture materials for the course "Linux for Engineering and Information Technology Applications", Rutgers University, Department of Mechanical and Aerospace Engineering, March 2009 [Online]. Available: http://coewww.rutgers.edu/www1/linuxclass2009/lessons/lesson9/sec_2.php. [Accessed: 17-Jan-2014].
- [70] Citrix Systems, Inc. "What is load balancing?," *Citrix.com*. [Online]. Available: http://www.citrix.com/content/citrix/en_us/glossary/load-balancing.html. [Accessed: 17-Jan-2014].
- [71] V. Viswanathan, "Load Balancing Web Applications," O'Reilly Media, 28 September 2001. [Online]. Available: <http://www.onjava.com/pub/a/onjava/2001/09/26/load.html>. [Accessed: 17-Jan-2014].
- [72] P. M. Sangal, "Load Balancing for Web Application Performance and Scalability," Jul-2009. [Online]. Available: <http://www.devx.com/enterprise/Article/42332>. [Accessed: 17-Jan-2014].
- [73] "jetNEXUS ADC and Load Balancing Platforms." [Online]. Available: <http://www.jetnexus.com/load-balancing-platforms.html>. [Accessed: 17-Jan-2014].
- [74] P. Sevcik and R. Wetzel, "Field Guide to Application Delivery Systems." NetForecast, Inc., September 2006. Available: <http://www.netforecast.com/wp-content/uploads/2012/06/NFR5085-Field-Guide-to-Application-Delivery-Systems.pdf>
- [75] R. Campbell and K. Alstad, "Performance: Scaling Strategies for ASP.NET Applications," Microsoft, April 2008. [Online]. Available: <http://msdn.microsoft.com/en-us/magazine/cc500561.aspx>. [Accessed: 17-Jan-2014].

- [76] writer02, "Different Types of Load Balancers in Computer Networking," *HubPages*. 24 July 2011 [Online]. Available: <http://writer02.hubpages.com/hub/Different-Types-of-Load-Balancers-in-Computer-Networking>. [Accessed: 20-Jan-2014].
- [77] Parker Samp, "HOWTO: Load balance HTTP with Linux and Squid." [Online]. Available: <http://parkersamp.com/2010/11/howto-load-balance-http-with-linux-and-squid/>. [Accessed: 17-Jan-2014].
- [78] T. Northcutt, "Implementing Web Server Load Balancing, Failover, and State with Squid." [Online]. Available: <http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/SA/v15/i01/a4.htm>. [Accessed: 20-Jan-2014].
- [79] AJEET, "Benefits of Web Server Apache | Downgraf." 11 October 2012 [Online]. Available: [http://www.downgraf.com/all-articles/benefits-of-web-server-apache/#prettyphoto\[13924\]/0/](http://www.downgraf.com/all-articles/benefits-of-web-server-apache/#prettyphoto[13924]/0/). [Accessed: 27-Jan-2014].
- [80] Boz Zashev (editor), "Server cluster definition." last edited on 29 September 2010 [Online]. Available: <http://wordframe.com/docs/wiki/server-cluster-definition/>. [Accessed: 17-Jan-2014].
- [81] Microsoft, "What Is a Server Cluster?: Server Clusters (MSCS)." [Online]. Available: [http://technet.microsoft.com/en-us/library/cc785197\(v=ws.10\).aspx](http://technet.microsoft.com/en-us/library/cc785197(v=ws.10).aspx). [Accessed: 17-Jan-2014].
- [82] tutorialspoint., "Web - Server Types." [Online]. Available: http://www.tutorialspoint.com/web_developers_guide/web_server_types.htm. [Accessed: 20-Jan-2014].
- [83] A. Sharma, "Tomcat - Is This an Application Server ? | Javalobby." DZone, 6 September 2008 [Online]. Available: <http://java.dzone.com/articles/tomcat-is-application-server-0>. [Accessed: 20-Jan-2014].
- [84] E. Geier, "6 Excellent Linux/Open Source Web Servers - Apache, Nginx, Lighttpd - Reviews," LinuxPlanet QuinStreet Inc., 6 December 2010. [Online]. Available: <http://www.linuxplanet.com/linuxplanet/reviews/7239/1>. [Accessed: 13-Feb-2014].
- [85] Microsoft, "Microsoft Web Platform - Server, IIS, Internet Information Services." [Online]. Available: <http://www.microsoft.com/web/platform/server.aspx>. [Accessed: 13-Feb-2014].
- [86] J. Persyn, "Introduction to Memcached", 27 May-2010. Available: <http://www.jurriaanpersyn.com/archives/2010/05/27/introduction-to-memcached/>
- [87] "memcached - a distributed memory object caching system." [Online]. Available: <http://memcached.org/>. [Accessed: 17-Jan-2014].
- [88] Marina Sprava, "Database Master-Slave Replication in the Cloud," *Jelastic Blog*. 15 January 2013 by [Online]. Available: <http://blog.jelastic.com/2013/01/15/database-master-slave-replication-in-the-cloud/>. [Accessed: 17-Jan-2014].
- [89] ProfitBricks, "Cloud Lexicon," *ProfitBricks*. [Online]. Available: <http://www.profitbricks.co.uk/cloud-lexicon>. [Accessed: 14-Feb-2014].
- [90] ProfitBricks, "Your Knowledge Base about Cloud Computing:Cloud Lexicon," *ProfitBricks*. [Online]. Available: <http://www.profitbricks.co.uk/cloud-lexicon>. [Accessed: 17-Jan-2014].
- [91] Team Parascale, "Defining Cloud Storage : Three Key Characteristics," *ITProPortal*. 3 December 2008 [Online]. Available: <http://www.itproportal.com/2008/12/03/defining-cloud-storage-three-key-characteristics/>. [Accessed: 17-Jan-2014].
- [92] Rackspace Support, "Create and Use Cloud Block Storage Snapshots." Rackspace US, Inc., Article ID: 3138, 4 November 2013 [Online]. Available: http://www.rackspace.com/knowledge_center/article/create-and-use-cloud-block-storage-snapshots. [Accessed: 22-Jan-2014].
- [93] "What is Cloud Management? A Definition from Webopedia.com.", QuinStreet Inc. [Online]. Available: http://www.webopedia.com/TERM/C/cloud_management.html. [Accessed: 22-Jan-2014].
- [94] IBM, "Cloud Management." [Online]. Available: <http://www-03.ibm.com/software/products/en/category/SWU20>. [Accessed: 22-Jan-2014].
- [95] "Apache License, Version 2.0." [Online]. Available: <http://www.apache.org/licenses/LICENSE-2.0>. [Accessed: 11-Jan-2014].
- [96] "Cloudify - Deploy a Simple Application Locally." [Online]. Available: <http://www.cloudifysource.org/guide/2.6/qsg/qsg>. [Accessed: 11-Jan-2014].

- [97] “Cloudify - Cloudify Installation and Setup.” [Online]. Available: http://www.cloudifysource.org/guide/2.6/setup/installation_and_setup. [Accessed: 26-Dec-2013].
- [98] “Bare Bones Software | Welcome.” [Online]. Available: <http://www.barebones.com/>. [Accessed: 10-Jan-2014].
- [99] “Groovy - Home.” [Online]. Available: <http://groovy.codehaus.org/>. [Accessed: 11-Jan-2014].
- [100] “Cloudify - The Open PaaS Stack.” [Online]. Available: <http://www.cloudifysource.org/>. [Accessed: 11-Jan-2014].
- [101] “What is Cloudbursting?- Trend Cloud Security Blog – Cloud Computing Experts.” [Online]. Available: <http://cloud.trendmicro.com/what-is-cloudbursting/>. [Accessed: 27-Jan-2014].
- [102] “Cloudify - Installing the Cloudify Shell.” [Online]. Available: http://www.cloudifysource.org/guide/2.6/setup/installing_the_cloudify_client. [Accessed: 27-Jan-2014].
- [103] “Cloudify - Cloudify Shell Prerequisites.” [Online]. Available: http://www.cloudifysource.org/guide/2.6/setup/cloudify_prerequisites. [Accessed: 27-Jan-2014].
- [104] “Cloudify - Get Cloudify.” [Online]. Available: http://www.cloudifysource.org/downloads/get_cloudify?utm_source=CloudifySource%25252BCommunity&utm_medium=Download%25252BButton&utm_campaign=Free%25252BDownload. [Accessed: 27-Jan-2014].
- [105] J. Varia and S. Mathew, “Amazon Web Services: Overview of Amazon Web Services.” January 2014. Available: http://d36cz9buwrutt.cloudfront.net/AWS_Overview.pdf
- [106] J. Varia, “Amazon Web Services - Architecting for The Cloud: Best Practices.” January 2011. Available: http://media.amazonwebservices.com/AWS_Cloud_Best_Practices.pdf
- [107] “Instance Types - Amazon Elastic Compute Cloud.” [Online]. Available: <http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-types.html>. [Accessed: 23-January 2014].
- [108] “AWS | Amazon EC2 | Instance Types.” [Online]. Available: <http://aws.amazon.com/ec2/instance-types/>. [Accessed: 23-Jan-2014].
- [109] “Amazon EC2 pricing.” [Online]. Available: <http://aws.amazon.com/ec2/pricing/>. [Accessed: 24-Jan-2014].
- [110] “Amazon Web Services Sign In.” [Online]. Available: https://www.amazon.com/ap/signin?openid.assoc_handle=aws&openid.return_to=https%3A%2F%2Fsignin.aws.amazon.com%2Foauth%3Fresponse_type%3Dcode%26client_id%3Darn%253Aaws%253Aiam%253A%253A015428540659%253Auser%252Fec2%26redirect_uri%3Dhttps%253A%252F%252Fconsole.aws.amazon.com%252Fec2%252F%253Fstate%253DhashArgs%252523%2526isauthcode%253Dtrue%26noAuthCookie%3Dtrue&openid.mode=checkid_setup&openid.ns=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0&openid.identity=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2Fidentifier_select&openid.claimed_id=http%3A%2F%2Fspecs.openid.net%2Fauth%2F2.0%2Fidentifier_select&action=&disableCorpSignUp=&clientContext=&marketPlaceId=&poolName=&authCookies=&pageId=aws.ssop&siteState=awscustomer&accountStatusPolicy=P1&sso=&openid.pape.preferred_auth_policies=MultifactorPhysical&openid.pape.max_auth_age=120&openid.ns.pape=http%3A%2F%2Fspecs.openid.net%2Fextensions%2Fpape%2F1.0&server=%2Fap%2Fsignin%3Fie%3DUTF8&accountPoolAlias=&forceMobileApp=0&forceMobileLayout=0. [Accessed: 01-Feb-2014].
- [111] D. Mos and T. Jin, “httpref -A Tool for Measuring Web Server Performance,” *HP Research Labs*. Technical report HPL-98-61, March 1998. Available: <http://www.hpl.hp.com/techreports/98/HPL-98-61.pdf>
- [112] N. Jauhari, “Load / Performance Testing Web Application - Httpperf | Linux Blog.” [Online]. Available: <http://linuxpoison.blogspot.se/2011/10/load-performance-testing-web.html>. [Accessed: 25-Jan-2014].
- [113] D. Kumaraage, “Benchmark testing with httpperf | Damitha’s Web Log.” Blog, 15 March 2009 [Online]. Available: <http://damithakumaraage.wordpress.com/2009/03/15/benchmark-testing-with-httpperf/>. [Accessed: 25-Jan-2014].

- [114] VMware, Inc., “VDI Server Sizing and Scaling.” VMware, Inc, Aug-2008. Available: https://www.vmware.com/pdf/vdi_sizing_vi3.pdf
- [115] A. Kejariwal, “Techniques for Optimizing Cloud Footprint.” In proceedings of IEEE International Conference on Cloud Engineering (IC2E), DOI: 10.1109/IC2E.2013.14, March 2013, pp. 258–268.
- [116] Amazon, “Auto Scaling.” [Online]. Available: http://docs.aws.amazon.com/AutoScaling/2010-08-01/DeveloperGuide/index.html?AS_Concepts.html. [Accessed: 06-Feb-2014].
- [117] T. Hassanov, “Web Application Scaling in Amazon Cloud.” B.Sc. Thesis, University of Tartu, Faculty of Mathematics and Computer Science, May-2012. Available: http://comserv.cs.ut.ee/forms/ati_report/downloader.php?file=6321A0D6E5723DBF1DBBC7F2E5BB5041B4AF668C
- [118] D. Kondo, B. Javadi, P. Malecot, F. Cappello, and D. P. Anderson, “Cost-benefit analysis of cloud computing versus desktop grids,” in *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, 2009, pp. 1–12.
- [119] Amazon, “The Economics of the AWS Cloud vs. Owned IT Infrastructure.” Amazon web services, 7 December 2009. Available: http://media.amazonwebservices.com/The_Economics_of_the_AWS_Cloud_vs_Owned_IT_Infrastructure.pdf
- [120] Amazon “How AWS pricing works.” Amazon web services, March 2012. Available: http://media.amazonwebservices.com/AWS_Pricing_Overview.pdf
- [121] V. Jinesh, “The Total Cost of (Non) Ownership of Web Applications in the Cloud.” Amazon web services, August 2012. Available: http://media.amazonwebservices.com/AWS_TCO_Web_Applications.pdf
- [122] Amazon “Amazon Web Services Simple Monthly Calculator.” [Online]. Available: <http://calculator.s3.amazonaws.com/calc5.html>. [Accessed: 31-Jan-2014].
- [123] “GigaSpaces | XAP In-Memory Computing software platform | Cloudify - Deploy, Manage & Scale your apps on the cloud.” [Online]. Available: http://www.gigaspaces.com/?utm_source=Google&utm_medium=PPC&utm_term=FromCloud&utm_content=DeployManageScale&utm_campaign=GigaSpaces%2BBrand&gclid=COO8kOivrLwCFYiQcgodgAcA2g. [Accessed: 08-Feb-2014].
- [124] “GNU General Public License v2.0 - GNU Project - Free Software Foundation.” [Online]. Available: <http://www.gnu.org/licenses/gpl-2.0.html>. [Accessed: 08-Feb-2014].

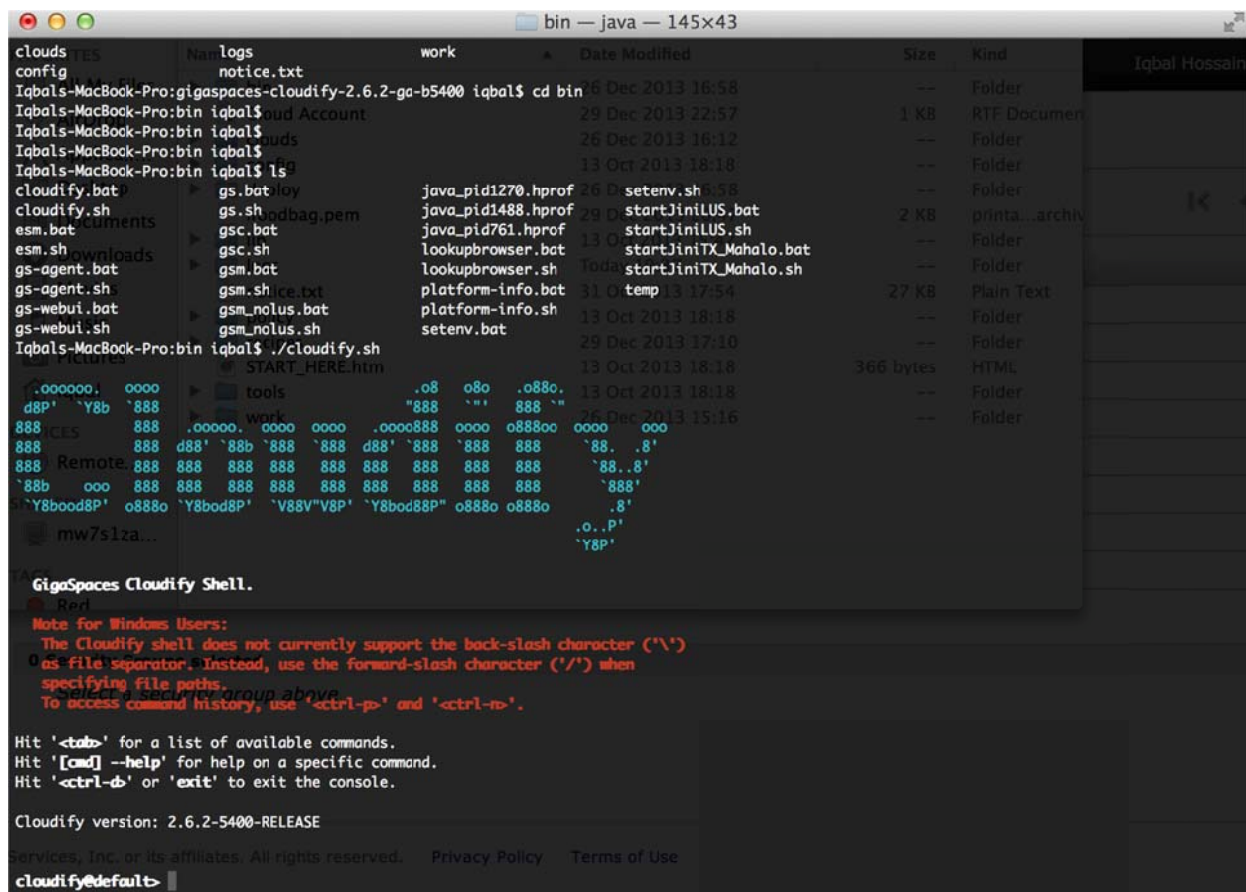
Appendix A: Installation of Cloudify

Prerequisites: Before you install the Cloudify shell, make sure that your environment meets all the minimum requirements. The following minimum requirements are for both Windows and *nix machines.

- [JDK 1.6](#) or higher—download the latest update available (e.g. JDK 6 Update 32)
- The `JAVA_HOME` environment variable must point to the correct **JDK** (not JRE) directory. For example, `D:\Java\jdk1.6.0_32`.
- `%JAVA_HOME%\bin` must be added to the beginning of the `PATH` environment variable. For example, `%JAVA_HOME%\bin;%SystemRoot%\System32;%SystemRoot%;`

Download and unzip the [Cloudify distribution file](#)

Then browse the `bin` directory of the distribution and run the “`./cloudify.sh` (for .nix)” or “`cloudify.bat` (for Windows)”



```
bin — java — 145x43
clouds  YES      Name logs          work          Date Modified      Size Kind
config  YES      notice.txt
Iqbal-MacBook-Pro:gigaspace cloudify-2.6.2-ga-b5400 iqbal$ cd bin 26 Dec 2013 16:58 -- Folder
Iqbal-MacBook-Pro:bin iqbal$ ls -la 29 Dec 2013 22:57 1 KB RTF Document
Iqbal-MacBook-Pro:bin iqbal$ cd .. 26 Dec 2013 16:12 -- Folder
Iqbal-MacBook-Pro:bin iqbal$ ls -la 13 Oct 2013 18:18 -- Folder
cloudify.bat  gs.bat  java_pid1270.hprof  setenv.sh  26 Dec 2013 17:10 -- Folder
cloudify.sh  gs.sh  java_pid1488.hprof  startJiniLUS.bat  29 Dec 2013 17:10 2 KB print...archiv
esm.bat  gs.sh  java_pid761.hprof  startJiniLUS.sh  13 Oct 2013 18:18 -- Folder
esm.sh  gs-agent.bat  gs.bat  lookupbrowser.bat  13 Oct 2013 18:18 -- Folder
gs-agent.bat  gs-agent.sh  gs.bat  lookupbrowser.sh  13 Oct 2013 18:18 -- Folder
gs-agent.sh  gs-agent.sh  gs.bat  platform-info.bat  13 Oct 2013 18:18 -- Folder
gs-webui.bat  gs-webui.sh  gs_m_nolus.bat  platform-info.sh  13 Oct 2013 18:18 -- Folder
gs-webui.sh  gs_m_nolus.sh  setenv.bat  setenv.bat  29 Dec 2013 17:10 -- Folder
Iqbal-MacBook-Pro:bin iqbal$ ./cloudify.sh 29 Dec 2013 17:10 -- Folder
START_HERE.htm  13 Oct 2013 18:18 366 bytes HTML
tools  13 Oct 2013 18:18 -- Folder
work  26 Dec 2013 15:16 -- Folder
GigaSpaces Cloudify Shell.
Note for Windows Users:
The Cloudify shell does not currently support the back-slash character ('\')
as file separator. Instead, use the forward-slash character ('/') when
specifying file paths.
To access command history, use 'ctrl-p' and 'ctrl-r'.

Hit 'ctrl-l' for a list of available commands.
Hit '[cmd] --help' for help on a specific command.
Hit 'ctrl-d' or 'exit' to exit the console.

Cloudify version: 2.6.2-5400-RELEASE
services, Inc. or its affiliates. All rights reserved. Privacy Policy Terms of Use
cloudify@default>
```


Appendix B: Configuration of Cloud controllers and cloud drivers

EC2 Cloud Configurations file called (ec2-cloud.groovy):

```

/*****
 * Cloud configuration file for the Amazon ec2 cloud. Uses the default jclouds-based cloud driver.
 * See org.cloudifysource.dsl.cloud.Cloud for more details.
 */
cloud {
    // Mandatory. The name of the cloud, as it will appear in the Cloudify UI.
    name = "ec2"
    /*****
     * General configuration information about the cloud driver implementation.
     */
    configuration {
        // Optional. The cloud implementation class. Defaults to the build in jclouds-based provisioning
        // driver.
        className "org.cloudifysource.esc.driver.provisioning.jclouds.DefaultProvisioningDriver"
        storageClassName "org.cloudifysource.esc.driver.provisioning.storage.aws.EbsStorageDriver"
        // Optional. The template name for the management machines. Defaults to the first template in
        // the templates section below.
        managementMachineTemplate "SMALL_LINUX"
        // Optional. Indicates whether internal cluster communications should use the machine private IP.
        // Defaults to true.
        connectToPrivateIp true
        // Optional. Path to folder where management state will be written. Null indicates state will not be
        // written.
        persistentStoragePath persistencePath
    }
    /*****
     * Provider specific information.
     */
    provider {
        // Mandatory. The name of the provider.
        // When using the default cloud driver, maps to the Compute Service Context provider name.
        provider "aws-ec2"
        // Mandatory. The prefix for new machines started for services.
        machineNamePrefix "cloudify-agent-"
        // Optional. Defaults to true. Specifies whether cloudify should try to deploy services on the
        // management machine.
        // Do not change this unless you know EXACTLY what you are doing.
        managementOnlyFiles ([])
        // Optional. Logging level for the internal cloud provider logger. Defaults to INFO.
        sshLogLevel "WARNING"
        // Mandatory. Name of the new machine/s started as cloudify management machines. Names are
        // case-insensitive.
        managementGroup "cloudify-manager"
        // Mandatory. Number of management machines to start on bootstrap-cloud. In production,
        // should be 2. Can be 1 for dev.
        numberOfManagementMachines 1
        reservedMemoryCapacityPerMachineInMB 1024
    }
    /*****
     * Cloud authentication information
     */
    user {
        // Optional. Identity used to access cloud.
        // When used with the default driver, maps to the identity used to create the
        // ComputeServiceContext.
        user "user"
        // Optional. Key used to access cloud.
        // When used with the default driver, maps to the credential used to create the
        // ComputeServiceContext.
        apiKey "apiKey"
    }
    cloudStorage {
    templates {

```

```

SMALL_BLOCK : storageTemplate{
    deleteOnExit true
    size 5
    path "/storage"
    namePrefix "cloudify-storage-volume"
    deviceName "/dev/sdc"
    fileSystemType "ext4"
    custom ([:])
}

})
}

cloudCompute {
    /*****
    * Cloud machine templates available with this cloud.
    */
    templates ([
        // Mandatory. Template Name.
        SMALL_LINUX : computeTemplate{
            // Mandatory. Image ID.
            imageld "linuxImageld"
            // Mandatory. Files from the local directory will be copied to this directory on
            the remote machine.
            remoteDirectory "/home/ec2-user/gs-files"
            // Mandatory. Amount of RAM available to machine.
            machineMemoryMB 1600
            // Mandatory. Hardware ID.
            hardwareld "hardwareld"
            // Optional. Location ID.
            locationld "locationld"
            // Mandatory. All files from this LOCAL directory will be copied to the remote
            machine directory.
            localDirectory "upload"
            // Optional. Name of key file to use for authenticating to the remot machine.
            //Remove this line if key files are not used.
            keyFile "keyFile"
            username "ec2-user"
            // Additional template options.
            // When used with the default driver, the option names are considered
            // method names invoked on the TemplateOptions object with the value as the
            parameter.
            options ([
                "securityGroups" : ["default"]as String[],
                "keyPair" : "keyPair"
            ])
            // Optional. Overrides to default cloud driver behavior.
            // When used with the default driver, maps to the overrides properties passed
            to the ComputeServiceContext a
            overrides ([
                "jclouds.ec2.ami-query": "",
                "jclouds.ec2.cc-ami-query": ""
            ])
            // enable sudo.
            privileged true
        },
        SMALL_UBUNTU : computeTemplate{
            // Mandatory. Image ID.
            imageld "ubuntuImageld"
            remoteDirectory "/home/ubuntu/gs-files"
            machineMemoryMB 1600
            hardwareld "hardwareld"
            locationld "locationld"
            localDirectory "upload"
            keyFile "keyFile"
            username "ubuntu"
            options ([
                "securityGroups" : ["default"]as String[],
                "keyPair" : "keyPair"
            ])
        }
    ])
}

```

```

        overrides (["jclouds.ec2.ami-query":"","
                    "jclouds.ec2.cc-ami-query":"","
                    privileged true
                ],
            },
        /*****
        * Optional. Custom properties used to extend existing drivers or create new ones.
        */
        custom ([
            "org.cloudifysource.clearRemoteDirectoryOnStart" : true
        ])
    }
}

```

EC2 Cloud Drivers (ec2-cloud.properties):

```

// Credentials - You must enter your cloud provider account credentials
user="XXXXXX"
apiKey="XXXXXXXX"
keyFile="XXXXXXXXX"
keyPair="XXXXXXXX"

// Advanced usage
hardwareId="m1.small"
locationId="us-east-1"
linuxImageId="us-east-1/ami-1624987f"
ubuntuImageId="us-east-1/ami-82fa58eb"
// Management persistence configuration. Replace with a string path to activate. 'null' indicates no persistence.
persistencePath=null

```


Appendix C: Writing Ifoodbag Application Recipe

Ifoodbag-application.groovy file:

```
application {
    name="Ifoodbag"
    service {
        name = "tomcat"
    }
}
```

Tomcat-service.groovy file:

```
import java.util.concurrent.TimeUnit;
import static JmxMonitors.*

service {
    name "tomcat"
    icon "tomcat.gif"
    type "APP_SERVER"
    elastic true
    numInstances 1
    minAllowedInstances 1
    maxAllowedInstances 3

    def instanceId = context.instanceId

    def portIncrement = context.isLocalCloud() ? instanceId-1 : 0
    def currJmxPort = jmxPort + portIncrement
    def currHttpPort = port + portIncrement
    def currAjpPort = ajpPort + portIncrement
    compute {
        template "SMALL_LINUX"
    }
    lifecycle {
        details {
            def currPublicIP = context.publicAddress
            def contextPath = context.attributes.thisInstance["contextPath"]
            if (contextPath == 'ROOT') contextPath="" // ROOT means "" by convention in Tomcat
            def applicationURL = "http://${currPublicIP}:${currHttpPort}/${contextPath}"
            println "tomcat-service.groovy: applicationURL is ${applicationURL}"
            return [
                "Application URL":<a href=\ "${applicationURL}"
                target=\ "_blank" >${applicationURL}</a>"
            ]
        }
        monitors {
            def contextPath = context.attributes.thisInstance["contextPath"]
            if (contextPath == 'ROOT') contextPath="" // ROOT means "" by convention in Tomcat
            def metricNamesToMBeansNames = [
                "Current Http Threads Busy": ["Catalina:type=ThreadPool,name=\ "http-bio-
                ${currHttpPort}\ "", "currentThreadsBusy"], "Current Http Thread Count": ["Catalina:type=ThreadPool,name=\ "http-bio-
                ${currHttpPort}\ "", "currentThreadCount"], "Backlog": ["Catalina:type=ProtocolHandler,port=${currHttpPort}", "backlog"], "Total
                Requests Count": ["Catalina:type=GlobalRequestProcessor,name=\ "http-bio-${currHttpPort}\ "", "requestCount"], "Active Sessions":
                ["Catalina:type=Manager,context=/${contextPath},host=localhost", "activeSessions"], ]
            return getJmxMetrics("127.0.0.1",currJmxPort,metricNamesToMBeansNames)
        }
    }

    init "tomcat_init.groovy"
    install "tomcat_install.groovy"
    start "tomcat_start.groovy"
    preStop "tomcat_stop.groovy"

    startDetectionTimeoutSecs 240
}
```



```

startDetection {
    println "tomcat-service.groovy(startDetection): arePortsFree
http=${currHttpPort} ajp=${currAjpPort} ..."
    !ServiceUtils.arePortsFree([currHttpPort, currAjpPort] )
}
postStart {
    if ( useLoadBalancer ) {
        println "tomcat-service.groovy: tomcat Post-start ..."
        def apacheService = context.waitForService("apacheLB", 180,
TimeUnit.SECONDS)
        println "tomcat-service.groovy: invoking add-node of apacheLB ..."
        def privateIP = context.privateAddress
        println "tomcat-service.groovy: privateIP is ${privateIP} ..."
        def contextPath = context.attributes.thisInstance["contextPath"]
        if (contextPath == 'ROOT') contextPath="" // ROOT means "" by convention in
Tomcat
        def currURL="http://${privateIP}:${currHttpPort}/${contextPath}"
        println "tomcat-service.groovy: About to add ${currURL} to apacheLB ..."
        apacheService.invoke("addNode", currURL as String, instanceId as String)
        println "tomcat-service.groovy: tomcat Post-start ended"
    }
}

postStop {
    if ( useLoadBalancer ) {
        println "tomcat-service.groovy: tomcat Post-stop ..."
        try {
            def apacheService = context.waitForService("apacheLB", 180,
TimeUnit.SECONDS)
            if ( apacheService != null ) {
                def privateIP = context.privateAddress
                println "tomcat-service.groovy: privateIP is ${privateIP} ..."
                def contextPath = context.attributes.thisInstance["contextPath"]
                if (contextPath == 'ROOT') contextPath="" // ROOT means "" by convention in
Tomcat
                def currURL="http://${privateIP}:${currHttpPort}/${contextPath}"
                println "tomcat-service.groovy: About to remove ${currURL} from apacheLB ..."
                apacheService.invoke("removeNode", currURL as String, instanceId as String)
            }
            else {
                println "tomcat-service.groovy: waitForService apacheLB returned null"
            }
        }
        catch (all) {
            println "tomcat-service.groovy: Exception in Post-stop: " + all
        }
        println "tomcat-service.groovy: tomcat Post-stop ended"
    }
}

}

customCommands ([
    "updateWar" : {warUrl ->
        println "tomcat-service.groovy(updateWar custom command): warUrl is
${warUrl}..."
        if (! warUrl) return "warUrl is null. So we do nothing."
        context.attributes.thisService["warUrl"] = "${warUrl}"
        println "tomcat-service.groovy(updateWar customCommand): invoking
updateWarFile custom command ..."
        def service = context.waitForService(context.serviceName, 60,
TimeUnit.SECONDS)
        def currentInstance = service.getInstances().find{ it.instanceId ==
context.instanceId }
        currentInstance.invoke("updateWarFile")
        println "tomcat-service.groovy(updateWar customCommand): End"
        return true
    },
    "updateWarFile" : "updateWarFile.groovy"
])

userInterface {

```

```

metricGroups = ([
    metricGroup {
        name "process"
        metrics([
            "Total Process Cpu Time",
            "Process Cpu Usage",
            "Total Process Virtual Memory",
            "Num Of Active Threads"
        ])
    },
    metricGroup {
        name "http"
        metrics([
            "Current Http Threads Busy",
            "Current Http Thread Count",
            "Backlog",
            "Total Requests Count"
        ])
    },
])

widgetGroups = ([
    widgetGroup {
        name "Process Cpu Usage"
        widgets ([
            balanceGauge{metric = "Process Cpu Usage"},
            barLineChart{
                metric "Process Cpu Usage"
                axisYUnit Unit.PERCENTAGE
            }
        ])
    },
    widgetGroup {
        name "Total Process Virtual Memory"
        widgets([
            balanceGauge{metric = "Total Process Virtual Memory"},
            barLineChart {
                metric "Total Process Virtual Memory"
                axisYUnit Unit.MEMORY
            }
        ])
    },
    widgetGroup {
        name "Num Of Active Threads"
        widgets ([
            balanceGauge{metric = "Num Of Active Threads"},
            barLineChart{
                metric "Num Of Active Threads"
                axisYUnit Unit.REGULAR
            }
        ])
    },
    widgetGroup {
        name "Current Http Threads Busy"
        widgets([
            balanceGauge{metric = "Current Http Threads Busy"},
            barLineChart {
                metric "Current Http Threads Busy"
                axisYUnit Unit.REGULAR
            }
        ])
    },
    widgetGroup {
        name "Current Http Thread Count"
        widgets([
            balanceGauge{metric = "Current Http Thread Count"},
            arLineChart {
                metric "Current Http Thread Count"
                axisYUnit Unit.REGULAR
            }
        ])
    }
])

```

```

    },
    widgetGroup {
        name "Request Backlog"
        widgets([
            balanceGauge{metric = "Backlog"},
            barLineChart {
                metric "Backlog"
                axisYUnit Unit.REGULAR
            }
        ])
    },
    widgetGroup {
        name "Active Sessions"
        widgets([
            balanceGauge{metric = "Active Sessions"},
            barLineChart {
                metric "Active Sessions"
                axisYUnit Unit.REGULAR
            }
        ])
    },
    widgetGroup {
        name "Total Requests Count"
        widgets([
            balanceGauge{metric = "Total Requests Count"},
            barLineChart {
                metric "Total Requests Count"
                axisYUnit Unit.REGULAR
            }
        ])
    },
    widgetGroup {
        name "Total Process Cpu Time"
        widgets([
            balanceGauge{metric = "Total Process Cpu Time"},
            barLineChart {
                metric "Total Process Cpu Time"
                axisYUnit Unit.REGULAR
            }
        ])
    }
    ])
}

network {
    port = currHttpPort
    protocolDescription = "HTTP"
}
}

```

Appendix D: Implementing Auto-Scaling Policies

Auto-Scaling code:

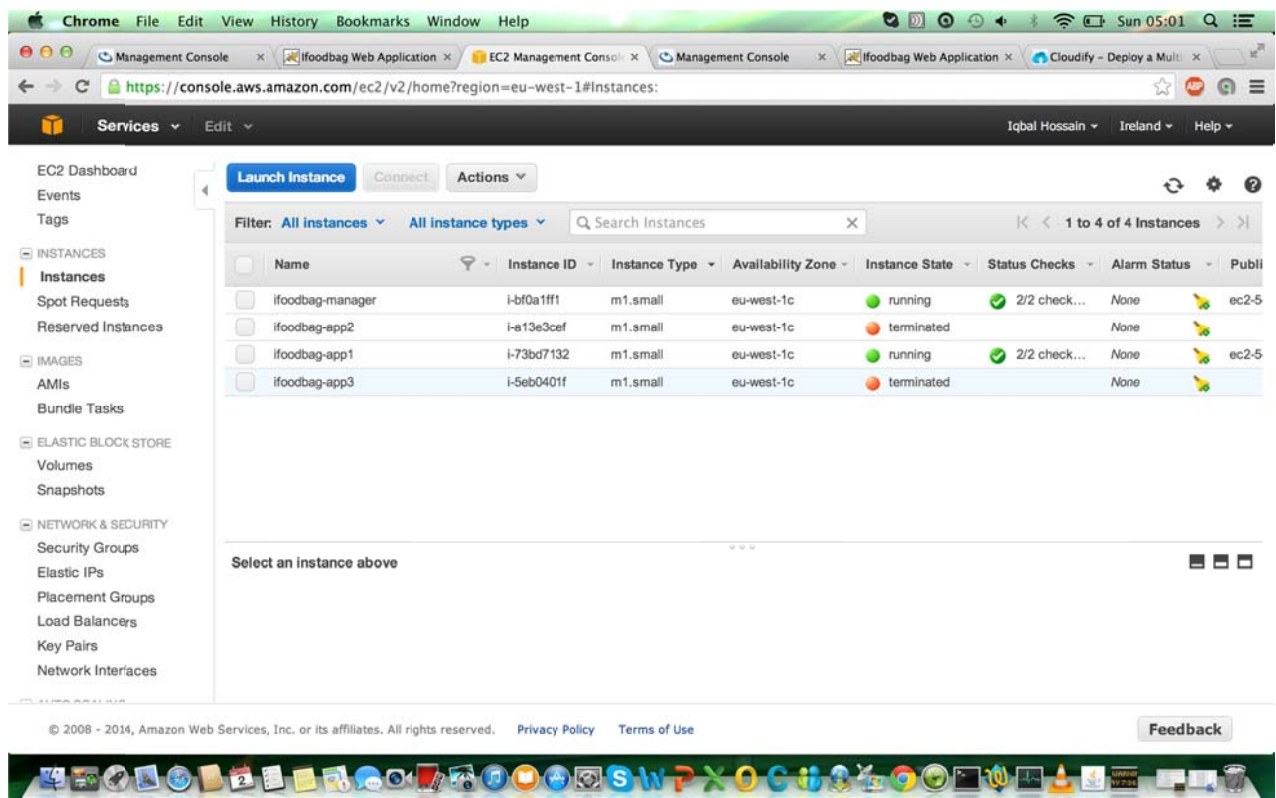
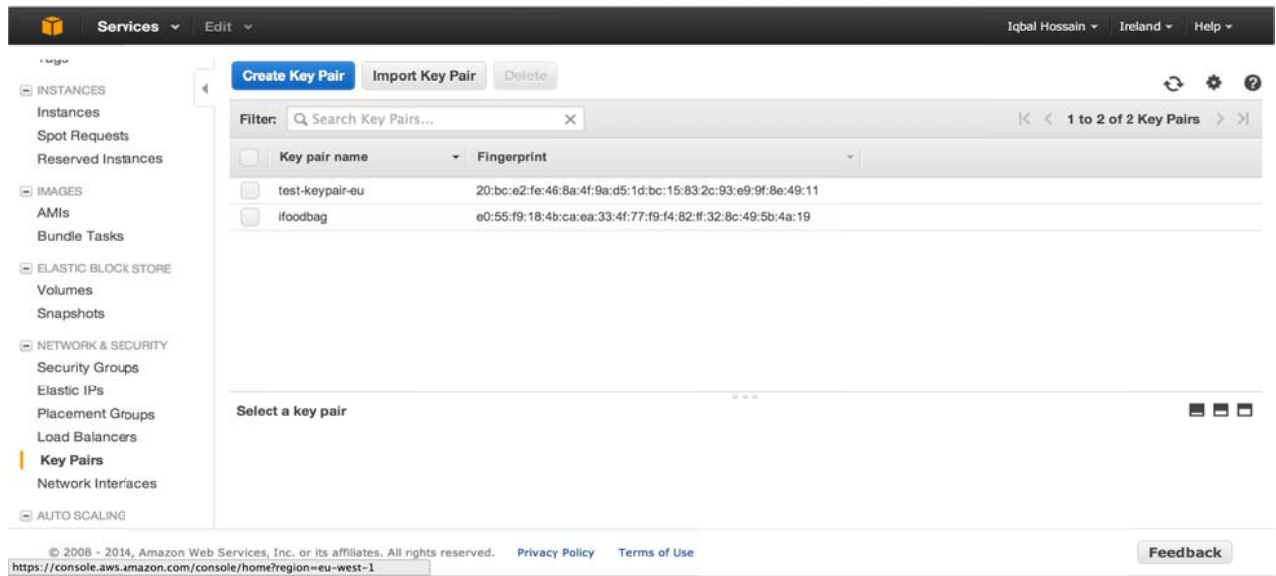
```
elastic true
minAllowedInstances 1
maxAllowedInstances 3

scaleCooldownInSeconds 60
samplingPeriodInSeconds 1

// Defines an automatic scaling rule based on "counter" metric value
scalingRules ([
  scalingRule {
    serviceStatistics {
      metric "Total Requests Count"
      statistics Statistics.maximumThroughput
      movingTimeRangeInSeconds 20
    }
    highThreshold {
      value 1
      instancesIncrease 1
    }
    lowThreshold {
      value 0.2
      instancesDecrease 1
    }
  }
])
```


Appendix F: Amazon EC2 Management Console

EC2 Management console:



Appendix G: Cloudify Web Management Console

Cloudify Web Management console

The screenshot shows the Cloudify Web Management Console dashboard. The top navigation bar includes 'Dashboard', 'Applications', 'Hosts', and 'Recipes'. The main content area is divided into several sections:

- Grid Status:** Shows a 'Health' indicator with a 'Fair' status and a warning icon. Below it are 'Resources' for 'CPU Cores' (13% used) and 'Memory' (89.9% used, 7.2GB).
- Services:** A summary of services including 'All Apps' (management, ifoodbag), 'Infrastructure' (1 Hosts, 1 Agents, 1 Deployers, 1 Orchestrators, 2 USPs), 'Application Services' (2 Web, 1 Stateful, 1 App Server), and 'Data Replication'.
- Alerts:** A table listing recent alerts.

Status	Type	Description	Location	Last Update
WARNING	Physical Memory Utilization	Memory crossed above a 80% threshold, for a period of 1 minute, with an ...	Iqbals-MacBook-Pro.local (192.168.0.5)	2014 Feb 1 23:27:07
SEVERE	Member Alive Indicator	Re-provisioned helloworld.tomcat [1] Instance	helloworld.tomcat [1]	2014 Feb 1 21:59:38
SEVERE	Member Alive Indicator	Re-provisioned helloworld.tomcat [1] Instance	helloworld.tomcat [1]	2014 Feb 1 21:58:36
SEVERE	Member Alive Indicator	Re-provisioned helloworld.tomcat [1] Instance	helloworld.tomcat [1]	2014 Feb 1 21:57:33
SEVERE	Member Alive Indicator	Re-provisioned helloworld.tomcat [1] Instance	helloworld.tomcat [1]	2014 Feb 1 21:56:30

This screenshot shows the detailed metrics for the 'ifoodbag' application. The top navigation bar includes 'Dashboard', 'Applications', 'Hosts', and 'Recipes'. The main content area is divided into several sections:

- Select Application:** 'ifoodbag' is selected, running on 'ec2'.
- Metrics:** A grid of eight custom metrics charts, each showing 'Min/Max/Average' and 'Deviation' over time.

Metric Name	Y-axis Range
Custom: Process Cpu Usage	0% to 100%
Custom: Total Process Virtual Memory	0 to 171968
Custom: Num Of Active Threads	0.0 to 40.0
Custom: Current Http Threads Busy	0.0 to 0.05
Custom: Current Http Thread Count	0.0 to 12.5
Custom: Request Backlog	0.0 to 0.05
Custom: Active Sessions	0.0 to 2.50
Custom: Total Requests Count	0.0 to 6.00

Appendix H: Simulating Auto-Scaling Process

Sending Request to the server:

```
httpperf --hog --server 54.194.238.66 --port 8082 --uri /ifoodbag --wsess=5,5,2 --num-conns 1000 --rate 10
```

```
httpperf --hog --server 54.194.238.66 --port 8082 --uri /ifoodbag --wsess=20,10,2 --num-conns 10000 --rate 30
```

```
httpperf --hog --server 54.194.238.66 --port 8082 --uri /ifoodbag --wsess=20,20,10 --num-conns 20000 --rate 100 --timeout 15
```

Figure shows adding new servers

