A Comparative Study Of Cloud Environments and the Development of a Framework for the Automatic Deployment of Scalable Cloud-Based Applications

by

Joyce Mlawanda

Thesis presented in partial fulfilment of the requirements for the degree Master of Science in Engineering at Stellenbosch University

Supervisor: Dr Gert-Jan van Rooyen
Department of Electrical and Electronic Engineering

March 2012
Declaration

By submitting this thesis electronically, I declare that the entirety of the work contained therein is my own, original work, that I am the sole author thereof (save to the extent explicitly otherwise stated), that reproduction and publication thereof by Stellenbosch University will not infringe any third party rights and that I have not previously in its entirety or in part submitted it for obtaining any qualification.

March 2012
Acknowledgements

• I would like to thank Dr Van Rooyen the guidance he offered throughout my studies. Thank you and God bless.

• I would like to thank my family for all the support.

• Lastly but not least I would like to thank God for giving me the ability and mind to carry out this project.
Abstract

Modern-day online applications are required to deal with an ever-increasing number of users without decreasing in performance. This implies that the applications should be scalable. Applications hosted on static servers are inflexible in terms of scalability. Cloud computing is an alternative to the traditional paradigm of static application hosting and offers an illusion of infinite compute and storage resources. It is a way of computing whereby computing resources are provided by a large pool of virtualised servers hosted on the Internet. By virtually removing scalability, infrastructure and installation constraints, cloud computing provides a very attractive platform for hosting online applications. This thesis compares the cloud computing infrastructures Google App Engine and Amazon Web Services for hosting web applications and assesses their scalability performance compared to traditionally hosted servers. After the comparison of the three application hosting solutions, a proof-of-concept software framework for the provisioning and deployment of automatically scaling applications is built on Amazon Web Services which is shown to be best suited for the development of such a framework.
Terms of reference

This project has been commissioned by Dr Van Rooyen and he has given specific instructions that the project should:

- Evaluate cloud computing for hosting web applications.
- Compare solutions by Google and Amazon to the traditional single-server approach by building a simple web application on all the three platforms.
- Measure and compare scalability when many users load the system.
- Build a proof-of-concept framework for the automatic provisioning of infrastructure and deployment of automatically scaling applications on the infrastructure, on the cloud computing platform which is best suited for the development of such a software framework.
List of symbols

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IT</td>
<td>Information Technology</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>RSS</td>
<td>Really Simple Syndication</td>
</tr>
<tr>
<td>EC2</td>
<td>Elastic Cloud Compute</td>
</tr>
<tr>
<td>HTML</td>
<td>Hypertext Markup Language</td>
</tr>
<tr>
<td>CSS</td>
<td>Cascading Style Sheets</td>
</tr>
<tr>
<td>XML</td>
<td>eXtensible Markup Language</td>
</tr>
<tr>
<td>CGI</td>
<td>Common Gateway Interface</td>
</tr>
<tr>
<td>WSGI</td>
<td>Web Server Gateway Interface</td>
</tr>
<tr>
<td>AMI</td>
<td>Amazon Machine Image</td>
</tr>
<tr>
<td>S3</td>
<td>Simple Storage Solution</td>
</tr>
<tr>
<td>SimpleDB</td>
<td>Simple Database</td>
</tr>
<tr>
<td>SQS</td>
<td>Simple Queue Service</td>
</tr>
<tr>
<td>PHP</td>
<td>PHP: Hypertext Preprocess</td>
</tr>
<tr>
<td>URL</td>
<td>Uniform Resource Locator</td>
</tr>
<tr>
<td>HTTP</td>
<td>Hypertext Transfer Protocol</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>LAMP</td>
<td>Linux, Apache, MySQL and PHP</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
</tbody>
</table>
# Contents

Declaration .................................................. i

Acknowledgements ........................................... ii

Abstract .................................................... iii

Terms of reference .......................................... iv

List of symbols ............................................... v

1 Introduction ............................................... 1
   1.1 Background ........................................ 1
   1.2 Motivation ........................................ 2
   1.3 Objectives ......................................... 2
   1.4 Related work ..................................... 3
   1.5 Overview of the study ............................. 4

2 Literature study ........................................ 6
   2.1 Cloud computing ................................... 6
   2.2 Google App Engine (GAE) .......................... 9
       2.2.1 Databases .................................. 10
   2.3 Amazon Web Services ............................... 10
       2.3.1 Amazon Elastic Compute Cloud (EC2) ....... 11
       2.3.2 Elastic Block Store (EBS) ................... 13
       2.3.3 Simple Storage Service (S3) ............... 13
       2.3.4 Simple Database (SimpleDB) .............. 13
       2.3.5 Simple Queue Service(SQS) ............... 14
       2.3.6 Amazon CloudWatch .......................... 15
       2.3.7 Automatic scaling ............................ 15
   2.4 Static server application hosting ............... 16
3 A comparative study of cloud-based environments

3.1 The web application ............................................ 17
3.2 Static server hosting ............................................ 18
3.3 Google App Engine hosting ................................. 18
   3.3.1 Provisioning of infrastructure ....................... 18
   3.3.2 Application development environment and deployment 19
   3.3.3 Data storage ............................................. 19
   3.3.4 Flexibility ............................................... 19
   3.3.5 Scalability .............................................. 20
3.4 Amazon Web Services (AWS) hosting ....................... 20
   3.4.1 Architecture ............................................ 20
   3.4.2 Provisioning of infrastructure ....................... 24
   3.4.3 Application development environment and deployment 24
   3.4.4 Data storage ............................................. 25
   3.4.5 Flexibility ............................................... 25
   3.4.6 Scalability .............................................. 25
3.5 Measuring the scalability performance of the different hosting solutions .. 26
3.6 Overall comparison of the hosting solutions .................. 27

4 Requirements of the framework .................................. 29
4.1 Technical requirements ........................................ 29
4.2 Functional requirements ...................................... 30
   4.2.1 Use case: Logging into the application ............... 30
   4.2.2 Use case: Viewing running applications .............. 30
   4.2.3 Use case: Deploying an application on AWS .......... 31
   4.2.4 Use case: Rebooting running servers .................. 32
   4.2.5 Use case: Terminating a running application ......... 32
4.3 Non-functional requirements ................................ 32
   4.3.1 Framework application .................................. 33
   4.3.2 Deployed application .................................. 33

5 System design and implementation ............................ 34
5.1 Deployed application ......................................... 34
   5.1.1 Master server .......................................... 34
   5.1.2 Interface server ........................................ 40
   5.1.3 Slave server ........................................... 42
5.2 Framework application ...................................... 45
   5.2.1 Architecture ............................................ 45
6 Testing

6.1 Framework application ........................................... 62
  6.1.1 Controller ................................................. 62
  6.1.2 Deploying an application .................................. 64
  6.1.3 Viewing running applications ............................. 70
  6.1.4 Terminating an application ................................. 70
  6.1.5 Rebooting a running server ............................... 72

6.2 The deployed application ..................................... 72
  6.2.1 Getting the application configuration file .............. 72
  6.2.2 Interface server ......................................... 74
  6.2.3 Slave server ........................................... 74
  6.2.4 Master server .......................................... 76
  6.2.5 Overall request processing system ...................... 78
  6.2.6 Automatic scaling of applications ....................... 80

7 Conclusion

7.1 Objectives ..................................................... 85
7.2 Results ......................................................... 85
  7.2.1 Comparative study of the Amazon Web Services and Google App
       Engine platforms ........................................... 85
  7.2.2 Framework ............................................... 87
7.3 Limitations and further work ................................. 87
7.4 Conclusion .................................................... 88
List of Figures

2.1 Diagram showing key elements of cloud computing. ......................... 8
3.1 Architecture of the application hosted on a static server. .................. 18
3.2 AWS application architecture. ............................................. 21
3.3 Scalability performance of GAE, AWS and statically hosted applications. .. 26
4.1 System use case diagram. .................................................... 30
5.1 The components of the master server ....................................... 35
5.2 Interaction diagram of the master server components. ....................... 36
5.3 The activities of the master EC2 server. .................................... 39
5.4 The components of the interface server. .................................... 40
5.5 The components hosted by the slave server. ................................ 42
5.6 Interaction diagram of the slave server. .................................... 44
5.7 Block diagram showing the architectural components of the framework. .. 45
5.8 How the frameworkApp class interacts with other components. .......... 47
5.9 The program flow for the controller module. ............................ 50
5.10 A UML component diagram showing the controller's data and functions. . 51
5.11 A screen shot of the login page. ............................................ 51
5.12 The program flow for the deploy module. .................................. 53
5.13 A screen shot of the application deployment web page. ..................... 54
5.14 The program flow for the terminate module. ................................ 56
5.15 The program flow for the reboot module. .................................. 57
5.16 The program flow for the view_running_applications module. .......... 58
5.17 A screen shot of the view_running_applications web page. ............... 59
5.18 The basic layout of the web pages of the application. ...................... 60
5.19 The web page navigation structure of the application. ..................... 61
6.1 Testing protocol for the login process. ..................................... 63
6.2 Testing protocol for passing requests to various model functions. .......... 65
6.3 Testing protocol for the get_user_input function of the deploy module. 65
**LIST OF FIGURES**

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4</td>
<td>Protocol followed when testing the <code>validate_user_input</code> function of the <code>deploy</code> module.</td>
<td>67</td>
</tr>
<tr>
<td>6.5</td>
<td>Continuation of the protocol followed when testing the <code>validate_user_input</code> function of the <code>deploy</code> module.</td>
<td>68</td>
</tr>
<tr>
<td>6.6</td>
<td>Protocol followed when testing the <code>get_user_data.py</code> Python script.</td>
<td>73</td>
</tr>
<tr>
<td>6.7</td>
<td>Protocol followed when testing the functions of the interface server.</td>
<td>75</td>
</tr>
<tr>
<td>6.8</td>
<td>Protocol followed when testing the functions of the slave server.</td>
<td>77</td>
</tr>
<tr>
<td>6.9</td>
<td>The interaction diagram for testing master server functions.</td>
<td>79</td>
</tr>
<tr>
<td>6.10</td>
<td>The graph showing results from tests run on the prime factors generation application.</td>
<td>82</td>
</tr>
<tr>
<td>6.11</td>
<td>The graph showing results from tests run on the ticket application.</td>
<td>83</td>
</tr>
</tbody>
</table>
List of Tables

2.1 AWS EC2 server instance types ................................. 12

3.1 A comparison of AWS, GAE and static server hosting solutions ........ 28

6.1 Table showing the recovery times of the ticket application under different request rates. ........................................ 83
Chapter 1

Introduction

1.1 Background

Cloud computing has recently emerged as a new computing paradigm [19, 30, 39]. This new way of computing diverges from the traditional way of computing whereby local servers or traditional data centers host computing resources. With cloud computing, compute power, data storage, application development platforms and software applications are offered on demand as services over the Internet and users only pay for the resources (such as CPU time or storage space) that they use. These computing resources are hosted on large pools of virtualised machines located in large distributed data centers [15].

Businesses, governments and individuals alike are migrating to the cloud because of its attractive features [21]. Because of the ever-increasing need for compute and storage resources, cloud computing offers an attractive solution. With its illusion of infinite computing resources [33], the cloud can flexibly adjust to online applications resource requirements.

A developer that wishes to launch an application on cloud infrastructure needs no upfront capital investment in infrastructure, as resources are rented out on a-as-needed basis, just like public utilities such as water and electricity [16, 23]. Cloud computing companies manage the costs of hardware infrastructure and software. Users do not need to buy hardware, pay for electricity to power and cool servers or buy software licences [47]. Because of the advantage of economies of scale, cloud computing companies are able to spread costs over many customers, thereby cutting costs for individual users. Additionally, since users only pay for what they use, they can cut costs by releasing system resources when no longer needed.

In a typical cloud-computing environment such as Amazon Web Services (AWS) or Google App Engine (GAE) resources can be rapidly provisioned and released through a self-service interface with minimal management effort [39]. This allows hosted applica-
tions to quickly grow or shrink, thereby achieving elasticity in resource usage. Elasticity enables scalability on the cloud. Scalability refers to a computing system’s ability to continually maintain its availability, reliability and performance with an increasing number of concurrent user requests to the system [22]. With scalability, application owners no longer need to carefully anticipate application peak loads. During this project we will compare the scalability of two dominant cloud computing platforms.

There are many companies offering cloud computing services. Amazon and Google are two of the major players in the cloud computing industry [16, 31], in which they offer Amazon Web Services (AWS) and Google App Engine (GAE) cloud computing platforms, respectively. This project will compare and contrast these different kinds of cloud computing.

1.2 Motivation

Cloud computing features can be exploited to build highly scalable, reliable and fault tolerant applications ranging from simple applications to huge CPU intensive applications that perform complex computations [39]. One of the benefits of the cloud is the ability to automate the provisioning of resources, deployment of applications on the provisioned resources as well as scaling of deployed applications [32]. It is desirable for developers to have an automated process that helps with all these activities concerned with hosting applications on the cloud. Having an automated application hosting process helps in reducing errors and facilitates in having an efficient process [32].

For the project, we exploit prospects of developing an automated process for hosting applications on either Amazon Web Services (AWS) or Google App Engine (GAE) cloud platforms. We choose these platforms since they represent the two major players [31, 16] in the cloud computing industry. We study, evaluate and compare the two different cloud computing platforms with the static hosting solution as the baseline from which the comparisons are made. We also look at how applications are deployed and scaled on the platforms.

After the study and comparison, a proof-of-concept software framework that manages the automatic provisioning of infrastructure, deployment of applications on provisioned infrastructure, and scaling of deployed applications is developed on the platform that is shown to be the best suited for such a framework.

1.3 Objectives

The main objective of this project is to study, evaluate and compare how applications are hosted and run on the cloud computing platforms Google App Engine and Amazon
Web Services, before identifying the platform best suited for the development of a proof-of-concept software framework that implements an automated process for provisioning resources, deployment of applications on the resources, and scaling of the applications.

The different web hosting solutions are compared in terms of complexity, flexibility, possible application configuration, and ease of use. This is done by building a simple web application and hosting the application on the different computing platforms. Additionally, the scalability of the web application is measured against an increased system load on each platform.

From the analysis described in chapter 3, we choose to develop the proof-of-concept software framework on the AWS platform. This platform is chosen (rather than Google App Engine), because the Google App Engine platform already have an automated process for provisioning and deployment of developer applications in place. Also on GAE, the applications are automatically scaled. The proof-of-concept framework does the following:

- automatic provisioning of various AWS infrastructure necessary to host an application;
- automatic deployment of a developer application on the provisioned infrastructure; and
- implementing automatic scaling of deployed applications according to parameters specified by developers.

1.4 Related work

There has been an increase in research activity in cloud computing in recent years [40]. There are various research papers addressing different aspects of cloud computing. Armbrust et al defines cloud computing as both the applications delivered as services over the Internet and the hardware and systems software in the datacenters that provide those services [13]. The article discusses some of the available cloud computing platforms such as Google App Engine, Amazon Web Services and Azure, distinguishing them based on the level of abstraction presented to the programmer and the level of management of the resources. The work covered in the article is quite similar to the objectives of this thesis i.e to study and understand cloud computing and compare different cloud computing platforms. However, [13] concentrates mainly on the theoretical aspects of cloud computing, whilst our project mainly focuses on the practical steps involved when hosting applications on Amazon Web Services and Google App Engine cloud platforms.

In another article, Wang et-al contribute to the cloud computing concept in terms of definition, functionality and typical applications [48]. The article also compares and differentiates cloud computing from grid computing. Buuya et-al [16] describe what cloud
computing is and the current trends in cloud computing. In the same article, some emerging cloud computing platforms are described.

In terms of the development of a framework for the automatic provisioning of cloud resources and scaling of deployed applications, RightScale has objectives comparative to this study [47]. RightScale is a cloud management platform that offers developers the ability to automatically provision infrastructure, deploy web applications on the provisioned infrastructure and manage the applications on cloud computing platforms such as Amazon Web Services, FlexiScale and GoGrid [47]. Using this management platform, developers can easily migrate between cloud computing platforms. The management platform offers a management dashboard where developers can deploy and manage their applications, including configuring their applications to automatically scale according to predefined rules. The proof-of-concept framework developed during this project is similar to RightScale, as it is also an automated system management tool for cloud based applications. However, the framework is much simpler than RightScale as it mainly serves as a proof-of-concept and focuses on one of the major cloud computing platforms, unlike RightScale that is a generic tool for deploying and managing applications across multiple cloud platforms.

Close to the work involved in this project is Scalr, an open source framework for the deployment of fully redundant, self-healing and automatically scaling websites on the Amazon Web Services cloud computing platform [22]. Scalr manages the deployment of applications on one cloud platform (Amazon Web Services), unlike RightScale that manages applications on multiple cloud platforms. Scalr provides a comprehensive tool for the management and deployment of applications on AWS using pre-built machine images for databases, load balancing, web servers and a generic machine image that can be customised to suit developer needs [22]. Developers spend time setting up the different virtual servers that make up the application. The framework application is a simple proof-of-concept and does not offer redundancy and self-healing of applications like Scalr. Also, unlike Scalr, developers do not need to set up the different virtual servers that make up the application system; they must simply submit the AWS keys needed to identify a particular user and details indicating how the application should be scaled.

1.5 Overview of the study

The rest of the report is structured as follows: Chapter 2 studies the available literature on cloud computing and the different cloud platforms. The comparison done on the different application hosting solutions is documented in chapter 3. The basic requirements of the framework to be built on the platform with the most favourable conditions for building such a framework are documented in chapter 4. Chapter 5 documents how the framework system is designed and implemented according to the requirements described in chapter...
4. In chapter 6, all the details about the tests and experiments performed to verify the correct working of the framework are documented. Lastly chapter 7 gives a summary of the whole project.
Chapter 2

Literature study

This chapter gives an overview of the concepts and technologies involved in the project.

2.1 Cloud computing

There are many definitions of cloud computing [21, 36, 16, 41]. However, from all the numerous attempts to define cloud computing, there are some common and recurring aspects of this type of computing and some of these aspects are described in the paragraphs that follow.

Cloud computing refers to the provision of computing resources as services over the Internet [15]. The Internet is a key element of cloud computing and its advancement, as well as the availability of broadband and wireless networking [19], has fueled cloud computing’s development. On network diagrams, the Internet is often depicted by an image of a cloud, and the term cloud computing is a reference to this practice [47, 46].

There are various companies providing cloud computing platforms that offer various computing resources as services [6, 26]. To access the cloud services, cloud platforms provide various web and non-web self-service interfaces and the services offered can be accessed anytime from anywhere in the world using various computing devices, ranging from mobile devices to desktop computers [47]. With cloud computing service delivery, consumers do not need to buy delivery infrastructure or software just like water and electricity service delivery. This eliminates the need for upfront capital investments and there is no on-going commitment to the use of rented computing resources, as consumers can quickly relinquish use of the services.

Cloud computing resources are housed in large data centers that are typically distributed across geographically separate regions of the world [15]. Distributing the data centers in different places provides fault isolation, since if one data center fails, the other data centers will not be affected. This also offers flexibility to application owners in terms of the location of their applications. For example consumers can choose to locate their
applications in data centers near their locations to reduce network latency.

Elasticity is a key characteristic of cloud computing. Because computing resources can be rapidly provisioned and relinquished in an automated manner, cloud applications can rapidly grow or decrease in terms computing capacity, thus making scalability of applications easy. This means that application owners no longer need to provision infrastructure for future needs in anticipation of increases in system demand. Likewise, system resources are not underutilised when user demand decreases. Elasticity is a desirable feature in online applications such as social networks where system peak loads and load growth can not be predetermined.

A cloud computing data center consists of large pools of abstracted virtualised servers, thus providing an illusion of infinite computing resources [13]. Virtualisation is an important aspect of cloud computing as it enables easy, scalable, efficient and flexible management and provisioning of resources on cloud-based platforms [15, 43, 41]. It abstracts the coupling between the operating system and underlying hardware, such that raw compute, network and storage resources can be unified as a large pool of resources [21, 14]. By using virtualisation, cloud computing data centers are able to serve a large number of people with different needs as it is possible to dynamically configure and provision resources for different applications [21, 23] from the unified large pool of resources. Virtualisation distinguishes cloud computing from traditional data centers as the latter offers raw physical hardware resources that cannot be configured dynamically.

Cloud computing has grown out of and uses the same infrastructure support as grid computing [21]. Grid computing refers to geographically distributed, parallel, complete computers that work together to provide computing resources such as compute power and storage for solving large scientific or technical problems [43]. Distributed computing is a key characteristic of both grid and cloud computing and the latter makes use of grid computing’s standard protocols for aggregation of distributed resources and transparent access to the resources to manage their distributed data centers [15]. However, grid computing and cloud computing are different as they were developed to address different problems and have different assumptions [21]. For example, grid computing does not rely on virtualisation to manage the provisioning of resources as in cloud computing [21].

In addition to virtualisation and grid computing, service-oriented architecture (SOA) is another technology that is used in the design of cloud computing services [43, 49]. SOA is a software design architectural style which presents business functionalities as a set of self-contained modules called services. The modules (services) have well-defined interfaces and are invoked through messages [44, 43, 15]. Cloud computing makes use of SOA to provide computing resources and software applications as a set of services called web services [44]. The behavior of web services is governed by a collection of standards that are designed to support inter-operable machine-to-machine interaction over a network.
Web services expose their capabilities through a web services interface [44].

Cloud computing’s key elements are the various computing devices (clients), the Internet and distributed large data centers consisting of virtualised servers [47] as shown in fig 2.1.

Cloud computing services generally fall into three categories [36]: infrastructure, software application and platform service categories.

**Software as a service (SAAS):** With this type of cloud computing, software is offered as a service over the Internet. Developers configure and use the software according to their needs without having to buy or code the software and they pay a fee to use the service. SalesForce.com is a cloud computing company that offers business software as a service on a subscription basis [14].

**Platform as a service (PAAS):** This service offers a platform for developing, testing and hosting of applications [50]. This service supports the full application development cycle. In this type of service, developers use APIs to develop applications on an abstracted platform, therefore they are not aware of the underlying cloud infrastructure. Google App Engine is a cloud based platform for developing and hosting applications [28].

**Infrastructure as a service (IAAS)** IAAS offers computing infrastructure such as storage and compute servers as a service. Using this service, consumers can save on costs associated with space required to house the infrastructure, power and cooling needs and personnel required to install, configure and maintain the infrastructure. Amazon Web
Services offers storage, compute and communication infrastructure as a service over the Internet.

This project focuses on the two cloud computing platforms Google App Engine and Amazon Web services. The following sections give an overview of these platforms.

### 2.2 Google App Engine (GAE)

Google App Engine (GAE) is a runtime platform that allows developers to host and run web applications and store data on infrastructure housed in a number of Google-owned large data centers distributed around the world [26, 45]. On the platform, developers develop their applications on their local machines before uploading the applications to the GAE infrastructure. The platform takes care of the provisioning of the required infrastructure, deployment of the uploaded applications on the infrastructure and automatic scaling of the deployed applications [45]. Developers do not need to maintain and administrate servers as this is taken over by the platform [25].

In order for developers to start using the platform, they need to sign up for a free user account with Google. The platform provides a web-based administration console for the management of the deployed applications, including the registering of new applications. To develop applications, one has to download and install GAE software development kit (SDK) for the programming language of choice on a local machine. The SDK simulates the GAE’s environment, including database operations and mail handling, on the developer’s local computer [28]. One can develop and test an application using the SDK on a local computer before deploying it on the GAE. The SDK includes a tool to upload the application to GAE once an application is ready for deployment. Additionally, one has to install the appropriate programming language for the corresponding SDK.

Hosting and running applications on the Google platform is free within set quotas. At present, GAE offers 1GB storage and enough CPU and bandwidth for about 5 million pageviews per month within the free quota limit [28]. Once applications begin to exceed the free limits for bandwidth, storage, or computation, consumers can upgrade their accounts to pay for additional resources on a metered basis [45].

Google provides APIs that can be used to integrate developer applications with Google-based user authentication and email services. Developers can allow their application’s users to log on to their applications using the Google accounts system instead of creating their own account systems [26]. Also, developer applications can use Google’s mail service to communicate with users [26].

GAE supports application development using either Python or Java. Applications on the platform operate in a secure sandbox and have no access to the underlying server operating system or hardware [28]. An application can not write to the server file system...
and can only read from files that were uploaded together with the application code [25, 45]. Developers can only store data on the platform by storing it on Google’s databases [47].

2.2.1 Databases

Previously, GAE did not support any traditional relational database systems [45], however they have recently introduced Google Cloud SQL [29]. Google Cloud SQL is a web service that allows developers to integrate their GAE applications with a MySQL database [29]. This service has been introduced to enable developers to easily move their applications and data into and out of the Google App Engine cloud platform.

Before introducing Google Cloud SQL, the platform only supported its proprietary database system BigTable. BigTable focuses on efficiently providing a data storage system that scales to very large sizes across thousands of servers [17]. This datastore system is based upon a dynamic and distributed architecture [17]. This technology has successfully proved itself in Google products such as Google Earth and Google Finance which place huge and varied workload demands on the system, by providing a flexible and high-performance data storage solution [17].

BigTable shares many implementation strategies with relational databases [17]. As with traditional relational databases, data is modeled using tables. However, unlike in traditional relational database tables, data layout and format are not rigidly enforced in BigTable tables, and users have dynamic control over these. A BigTable table is a sparse, distributed, persistent multidimensional sorted map [17]. Each map value is indexed using a row key, column key and a timestamp and the value is an uninterpreted string. BigTable also differs from relational databases in the way it is accessed. It offers a different interface to its users in the form of a Datastore API. The API provides functions for creating, changing and deleting data models [17]. Also, there are functions for changing column metadata such as access control rights [17]. Transactions and queries can be run over database entities adding or deleting values from the datastore. Values can also be looked up from rows and iterations can be run across a subset of data in the datastore.

2.3 Amazon Web Services

Amazon Web Services (AWS) is a cloud computing platform that offers scalable computing resources such as storage, communication and compute power as services over the Internet [2]. Consumers can provision and have these services up and running within minutes [47] using a web services interface.

Before sending a request for any services offered by Amazon Web Services, a developer should sign up for an AWS account. An AWS account enables developers to use any Amazon Web Services products. When a developer registers for an AWS account, the
developer is issued with two keys namely, Access Key and Secret Access Key [2]. Any service offered over the Internet, requires a particular user to acquire a user ID and password for unique identification. Similarly, the Access Key and Secret Access Key serve the purpose of a user ID and Password to access AWS. The two keys are described below:

- **Access Key ID**: This key plays the role of the username and is 20-character, alphanumeric string that uniquely identifies a particular developer.

- **Secret Access Key**: This is a 40-character sequence which is used to calculate a digital signature that is sent along with the Access Key ID. The calculated digital signature acts as a password that confirms the identity of a user. This key is called secret because it is assumed that only the owner of an AWS account knows it.

For communication with AWS there are various web tools, browser plugins and standalone applications that provide an interface to AWS. Also, AWS requests can be integrated into applications written in most of the available modern programming languages. To make programming tasks easier for application developers using AWS, Application Programming Interfaces (APIs) for several programming languages are provided and can be downloaded and integrated into an application. The various tools and APIs implement the web services interface that is used to communicate with AWS [35].

### 2.3.1 Amazon Elastic Compute Cloud (EC2)

This is a service that offers compute capacity in the form of compute virtual servers called Elastic Compute Cloud (EC2) servers. Amazon consumers are able to create, launch and terminate servers on demand [42]. This service is elastic in the sense that applications can rapidly grow or shrink in size on demand by rapidly launching more and terminating some server instances respectively. Animoto.com is a real-world example of this capability of the EC2 service. The company owns a web application hosted on AWS that was able to start with only 50 EC2 servers on its inception, but managed to scale up to 3 500 servers in three days as the application rapidly became popular [12, 31].

Amazon EC2 server instances are launched from server templates called Amazon Machine Images (AMIs). AMIs contain software configurations such as an operating system, application software and associated settings [6]. The machine images come in a variety of configurations offering a wide range of web and non-web application platforms and support all major operating systems such as Windows, Linux and OpenSolaris [2]. The servers also vary in terms of CPU power, memory and input/output performance [31].

There are two distinct types of servers, namely standard and high-CPU instances [43, 42]. Standard instances come in small, large and extra-large sizes. Of the standard
instances, small instances have the least amount of memory, storage and CPU power, whilst extra-large instances have the highest amount of these resources [42]. High-CPU instances come in medium and extra-large sizes [43]. Table 3.1 shows the different EC2 servers available.

New EC2 instances can be launched within a few minutes [47]. On a launched EC2 server, developers can control the entire server’s software including the kernel [13] and the software can be customized to suit developers’ needs similar to an ordinary local server. Developers have privileges to start and stop the servers, install software packages or attach virtual disks to the servers just like a local server [15]. In order to save changes made to a running server instance, the instance’s configurations have to be bundled and saved into an AMI, otherwise these changes will be lost when the server is terminated [2].

Amazon EC2 offers lower levels of abstraction than Google App Engine [13]. Scalability is not automatically implemented as with GAE, therefore developers have to implement scalability themselves. This can be done in a variety of ways and other services on the platform such as CloudWatch, Auto Scaling, and Elastic Load Balancing can be combined to implement scalability [47]. Also the EC2 service can be combined with the SQS service to provide a highly scalable system.

Amazon data centers are located in different regions of the world (e.g. Europe, Asia, etc.), [6], therefore EC2 server instances can be launched from different regions. Each region contains multiple availability zones and each zone is separated from failures from other zones. Developers can launch EC2 instances in multiple availability zones to protect applications from failure in a single location.

When launched, Amazon EC2 instances are allocated two dynamic IP addresses (a private and a public IP addresses). Private addresses are used internally in the Amazon EC2

<table>
<thead>
<tr>
<th>Instance type</th>
<th>Memory capacity</th>
<th>Compute units</th>
<th>Storage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard small</td>
<td>1.7 GB</td>
<td>1 EC2 Compute Unit</td>
<td>160 GB</td>
</tr>
<tr>
<td>Standard large</td>
<td>7.5 GB</td>
<td>4 EC2 Compute Units</td>
<td>850 GB</td>
</tr>
<tr>
<td>Standard extra-large</td>
<td>15 GB</td>
<td>8 EC2 Compute Units</td>
<td>1,690 GB</td>
</tr>
<tr>
<td>High-CPU medium</td>
<td>1.7 GB</td>
<td>5 EC2 Compute Units</td>
<td>350 GB</td>
</tr>
<tr>
<td>High-CPU extra-large</td>
<td>7 GB</td>
<td>20 EC2 Compute Units</td>
<td>1690 GB</td>
</tr>
</tbody>
</table>

Table 2.1: AWS EC2 server instance types
domain, whilst the public addresses are reachable over the Internet. Fixed IP addresses are not available by default and can be obtained at an additional cost [15]. Elastic IP addresses are associated with a developer AWS account, not a particular server instance [42]. Elastic IP addresses are necessary because server instance are allocated dynamically to an application and the application can be using a different server instance at different times [42, 2]. One IP address can be mapped to different server instances at different times, in case one instance is replaced by another.

Amazon EC2 supports relational databases such as MySQL and SQLite. Data can also be persistently stored using other Amazon services such as the Elastic Block Store (EBS), the Simple Storage Service (S3) and SimpleDB [6].

2.3.2 Elastic Block Store (EBS)

Amazon Elastic Block Store is a virtual storage system designed to work with EC2 instances. With this system, storage volumes are mounted as devices by EC2 instances and appear as unformatted external storage devices. Developers are limited to 5000 EBS volumes or 20TiB storage size [5]. An Elastic Block Store can be attached to any server instance in the same availability zone. This storage exists separately from the actual instances and persists until deleted [43].

2.3.3 Simple Storage Service (S3)

Another service offered by AWS is the Simple Storage Service (S3). This service offers scalable storage of data on servers hosted on the Amazon cloud infrastructure [11].

Data is stored as objects on S3. Each object consists of a key, data and metadata. The metadata part of the object consists of name-value pairs that define properties of the object. Developers can specify custom metadata in addition to default properties such as "content-type" and "date last modified". Data in each object can be up to 5 terabytes [11]. An S3 object key uniquely identifies an object in a storage namespace. The key name should be up to 1024 bytes in length.

S3 data objects are stored in buckets. A bucket is a folder hosted on S3 and partitions the namespace of S3 objects at the top level [11]. An S3 bucket can be assigned any bucket name but the name must be unique across all of Amazon S3, similarly to Internet domain names whereby a domain name belongs to only one owner.

2.3.4 Simple Database (SimpleDB)

Amazon SimpleDB is a scalable data store provided by Amazon Web Services [32]. It provides a web services interface to create and store multiple data sets, query data and return the results and data is automatically indexed.
Amazon SimpleDB stores relatively small amounts of data of up to 1024kB. If there is need to store large amounts of data, it should be stored on Amazon S3, while storing metadata and pointers on the SimpleDB database since it provides quick access through indexing.

Data in this database is stored in domains and queries can be run across a domain. Unlike traditional relational databases, Amazon SimpleDB is schema-less as there is no need to define the structure of an entity beforehand [32, 43, 42]. Data is stored as items and each item is an array of attribute-value pairs. Before storing data, users need to first create domains, and then store items as attribute-value pairs. Each item should belong to a particular domain. After storing the items, the domains can be queried to retrieve data.

2.3.5 Simple Queue Service (SQS)

In AWS Simple Queue Service (SQS), messages are stored in a queue hosted by Amazon Web Services as they travel between computers [7]. Computers (which can also be EC2 instances) that access the queue can be located anywhere on the Web and do not need any special software to access the queue. These computers do not need to be synchronized and can access the queue independently [43, 7]. Amazon SQS makes it possible for distributed components of an application to exchange data without requiring each component to be available. This is unlike regular network communications where each component has to be available for communication to take place.

Each message stored on Amazon SQS is a string of up to 8kB and the number of messages that can be added to the queue is unlimited [7]. Messages on the queue can be sent and read at the same time. When a message is received by one computer it is locked so that other computers do not access the message whilst it is being processed. The lock time can be changed via the ChangeMessageVisibility operation of the SQS service [7]. After a computer has processed a message, it has to be deleted from the queue so that it cannot be reprocessed by other computers if the Visibility timeout expires.

Amazon SQS works hand-in-hand with Amazon EC2 to produce highly scalable applications. An Amazon queue may be used to determine the load on an application hosted on Amazon EC2 instances. The backlog of messages in a queue can be used to determine an application’s load, where each message represents a user request. Alternatively, the amount of time spend in a queue by a user request can be used to determine the responsiveness of an application. Using either of these metrics, Amazon EC2 instances can be instantiated to meet demand or terminated so that computing resources aren’t wasted. In this project we will explore the idea of using SQS queues to implement the scalability of applications.
2.3.6 Amazon CloudWatch

Amazon CloudWatch is a web service that enables developers to monitor their EC2 servers in real time [3]. The service collects metrics such as network traffic, CPU utilisation and disk reads and writes. Using the service, developers are able to manage various EC2 server metrics as well as configure alarm conditions based on the collected metrics [4].

With the CloudWatch service, there is an alarm object that watches over each monitored metric [4]. Developers need to create an alarm, specifying the metric to monitor, threshold values for the metric, number of evaluation periods and optionally one or more actions to perform when the alarm changes state.

The CloudWatch service can be configured to trigger an automatic scaling action, by sending an alarm trigger to the AutoScale service (discussed next) responsible for automatically scaling EC2-based applications.

2.3.7 Automatic scaling

The Amazon AutoScale service was designed for applications running on multiple EC2 servers. This service is commonly used in application scenarios where multiple copies of an application run on a group of EC2 servers [8]. The service is used to automatically instantiate new or terminate running EC2 servers in a server group, based upon policies defined by the application owners [8]. The AutoScale service is able to automatically detect changes in the system load and add more or remove compute capacity accordingly by monitoring various metrics collected from the server group. The AutoScale service can also be used to perform health checks on system servers, replacing failed or unhealthy server instances based upon policies defined by developers.

The service is based upon the concept of auto-scaling groups. An auto-scaling group is a logical grouping of EC2 server instances that share similar characteristics and are grouped together for the purposes of instance scaling and management [1]. AutoScale works by increasing or decreasing the number of instances in an auto-scaling group. The AutoScale service uses a launch configuration to launch new EC2 servers. This configuration is created by developers and it captures parameters that define the EC2 servers to be launched.

AutoScale depends upon the CloudWatch service. As described in the previous section, CloudWatch is used to collect various metrics defined by developers from running EC2 servers. Developers should define the CloudWatch service to trigger and send the alarms to the Autoscale service based on the collected metrics. They also need to define automatic scaling policies that describe what should happen when an alarm is triggered.
2.4 Static server application hosting

In the static hosting solution, compute and database servers typically reside within control of the application owners [15]. Careful capacity planning should be done to determine the right infrastructure capacity that would be able to host the application and provide acceptable performance to users. For example, the amount of storage as well as the server capacity required by an application need to be determined beforehand. This is because in case of insufficient resources, the problem cannot be quickly resolved by dynamically allocating more resources as the case with cloud computing.
Chapter 3

A comparative study of cloud-based environments

Amazon Web Services and Google App Engine cloud platforms offer different service models addressing different application hosting needs. Google App Engine offers a web application development and hosting platform, whilst AWS offers raw infrastructure as services.

In this chapter we compare how applications can be hosted on the different platforms. The comparison is achieved by building a simple web application and hosting the same application on each of the platforms. The platforms are compared in regards to; provisioning of infrastructure, application development environment and deployment, flexibility and scalability. At the end of the chapter we run tests to measure the scalability performance of each platform under increasing load.

In later chapters we describe the development of a framework for the automatic provisioning of infrastructure, deployment of an application on the provisioned infrastructure and automatic scaling of the deployed application on AWS. The comparison done in this chapter shows that the AWS platform is best suited for the development of such a framework.

3.1 The web application

To compare the different hosting solutions, a simple search application is built on each platform. On the web application, users enter in search queries and the application searches through an information database populated with data. The scalability of each system is measured as system loads are increased. The search function is chosen as it requires considerable processing power as the search is being executed on the data storage.
Chapter 3 — A comparative study of cloud-based environments

3.2 Static server hosting

Fig. 3.1 shows the architecture of the application hosted using a static local physical web server.

The application is hosted on a local physical machine running the Apache HTTP web server. Application data is stored in a MySQL database. Users access the application through their browsers and the web server receives and processes user requests. To perform the search functionality, the application searches through data stored in the database.

In order to host applications using static servers, developers have to plan for and acquire all the required compute and data storage resources in advance. They need to carefully plan for their applications' peak loads in order to acquire enough computing resources. Administrators and developers have to spend considerable time setting up and configuring web and database servers and the servers need to be maintained.

Hosting applications this way is not flexible as the application hosting environment cannot change dynamically like on cloud computing platforms as the infrastructure has to be planned for and be acquired before being used.

3.3 Google App Engine hosting

This section describes how the application is developed on Google App Engine’s virtualised web servers. This is in contrast to the static server hosting solution, where the servers are physical machines.

3.3.1 Provisioning of infrastructure

The application requires a web server to host the application and a database to store the application’s persistent data. On the platform developers do not need to provision web servers and databases before deploying an application on the platform. An application administration console is provided where application developers can create an application.
by providing an application identifier. After that a developer simply needs to upload a developed application to the platform.

### 3.3.2 Application development environment and deployment

As previously stated in section 2.2, GAE application code can only be written in Python or Java. For this application, Python is used. Although interpreted languages such as Python execute more slowly than compiled languages such as Java, we choose Python, because it is a high level language that allows us to write the same Java code in fewer lines [34, 38]. This makes the programming task easier.

Python applications on GAE communicate with web servers using the web server gateway interface (WSGI) protocol. Because writing WSGI code can be cumbersome, application frameworks that implement WSGI such as Webapp and Django and take care of the low-level details concerned with the WSGI protocol are provided to simplify the programming task [27]. In this application, the Django framework is used to communicate with the web server.

During the development of the application, a software development kit (SDK) for Python is downloaded and installed on the local computer. The SDK includes Django version 0.96 and is used in the application. When using the SDK, applications are structured in the local file system in the same way in which they will be set up in the Google servers after they are uploaded [18]. The test application is developed and tested locally before being uploaded to the GAE using the Python uploading utility included with the SDK.

### 3.3.3 Data storage

The application’s data is stored in the BigTable database. The database is accessible through an API provided by Google called data store [18, 17]. This API abstracts the underlying BigTable concepts from developers. Using the data store, data is defined as entities and stored in data models. Entities consists of one or more properties and each property is of a specific type. Using GAE data models, developers need to define the common properties of the data entities to be stored inside the data model before storing the data. Google App Engine data models do not enforce rigid data structures.

### 3.3.4 Flexibility

Google App Engine places restrictions on the model of applications that can be hosted on the platform. Developers do not have a choice on the development and hosting environment as the system provides a pre-built solution with less room for application customisation. Many applications do not fit into the model provided by the platform.
The platform is meant for large-scaling web applications that are built using either Java or Python. The system does not allow hosted applications to write to the file system and only allows storage of persistent data.

The platform provides a quick solution for hosting scalable web applications with less work involved at the expense of flexibility.

### 3.3.5 Scalability

All applications hosted on the platform are automatically scaled. Developer applications need not be aware of the automatic scaling mechanisms.

### 3.4 Amazon Web Services (AWS) hosting

This section describes the building and hosting of the application on the AWS infrastructure.

#### 3.4.1 Architecture
Figure 3.2: *AWS application architecture.*
Cloud computing platforms are designed to make scalability of applications easy. On AWS (unlike on GAE where hosted applications are automatically scaled), developers have to design application architectures that are scalable in order to leverage infrastructure scalability possible on the platform. Fig. 3.2 shows the architecture of the application that is built on AWS.

In order to have a scalable architecture, we separate the high-processing tasks that require more processing power from the other functions of the application. The separated tasks are implemented in separate modules. This decoupling of the high-processing tasks from the other tasks of the system allows us to scale the application by adding more instances of the high-processing module.

The developed search application communicates with users’ browsers, getting search request strings before executing the search and displaying back the processed search results to the browsers. In our application we identify the tasks associated with executing the search as the high-processing tasks and separate these tasks from the tasks concerned with communicating with the browser. We implement these different functions in different modules; business processing logic module for the search activities and presentation logic module for activities that interface the application to users’ browsers. There are no dependencies in code logic between these two modules.

The business processing logic functions as the workhorse of the application. We further describe this logic in section 5.1.3. All search requests are passed on to the workhorses by the presentation logic for processing. We automatically scale the system by adding more instances of workhorses when load increases and removing some workhorse instances when load decreases. The workhorse instances work independent of each other in serving user requests. We implement a separate automatic scaling module that detect system load changes, adding more or removing some workhorse instances accordingly. The presentation, business processing, and automatic scaling logic modules are hosted by different EC2 servers namely, interface, slave, and master servers respectively. The servers are shown in fig. 3.2.

**Request SQS queue**

SQS queues are previously described in 2.3.5. The interface server module communicates requests to slave server instances using a SQS queue. All request messages from the interface server are written to the same request queue, which is accessed by the slave servers. A request message contains a request to execute the business processing logic on the slave servers. The SQS service allows multiple devices to access a queue at the same time and this enables multiple slave servers in the processing cluster to read messages from the request queue at the same time. To prevent the slave servers from reading and processing the same message at the same time, SQS allows developers to lock messages
from being read by other devices whilst being processed by another for a time that can be configured by the developer.

**Interface server:**

The Interface server takes on the role of interfacing the system to users’ browsers. It is a web server that holds the presentation logic of an application that implements the interface to the users of the application. This logic communicates with browsers, getting user search requests, passing the requests to and getting processed responses from the business layer and displaying the responses to the browser.

The interface server communicates with the slave servers using a SQS queue. With each search request to the web server, a message representing the request is sent to the SQS queue by the presentation logic where the business processing logic in the slave servers reads the message. After processing responses, the business logic writes these responses to a SimpleDB domain, therefore after sending a request to the SQS queue, the presentation logic on the interface server monitors for the response in the expected SimpleDB domain.

**Slave server:**

The slave server is an application server that hosts the business logic of the application. The business logic module contains code that implements the search function. The logic reads request messages from the request queue and executes the search before sending back the responses to the response S3 bucket where the interface server collects the responses.

Having a number of slave servers serving requests means that we need a technique of assigning user requests to slave servers in a way that load balances the requests across all the available servers, making sure that high utilisation of all the available resources is achieved. In the application we do not implement a separate load balancing system as the request SQS queue and the slave servers provide a natural load balancing system. On the slave servers for each message picked up from the request queue, a separate thread is created for processing the request. A business logic instance only picks up a request for processing if its average CPU usage is less than 100%. If the CPU usage reaches 100% then a slave server will not pick any more requests to process until the usage drops. This gives chance to other less busy slave server instances to pick and process requests.

**Master server**

The master server hosts the automatic scaling module. The program logic in this module, determines the system load by monitoring the number of SQS messages in the request queue. This number represents the approximate number of incoming user requests as each incoming request results in a message being written to the SQS queue. The automatic
scaling mechanism instantiates more or terminates some slave EC2 servers based on the determined load, thus automatically scaling the system.

### 3.4.2 Provisioning of infrastructure

Amazon EC2 servers are accessed using the web services interface. There are various web and non-web tools that wrap around the web services interface provided for use when provisioning EC2 servers [2]. A Mozilla Firefox plugin, ElasticFox [9] and EC2 command line tools are some examples of these tools. To access the service from inside application programming code, there are various application programming interfaces (APIs) for different programming languages that implement the web services interface, available for use. To provision EC2 servers outside of programming code, we use ElasticFox. Boto is a Python API package consisting of several Python API classes that implement the web services interface to AWS [10]. We use Boto to interface with AWS from inside application programming code.

Developers can choose EC2 servers from a number of server instances with different compute capacity [42]. For the application, a standard small server instance is used for all the servers involved. There are various standard small server machine images with different software environments. A Linux server with the Apache web server, PHP and Python programming languages pre-installed is chosen. ElasticFox is used to identify the Amazon machine image of the server before instantiating the server through the tool’s web interface.

SQS queues, SimpleDB domains and S3 buckets are created as needed inside programming using the various Boto API classes.

Developers have access to the underlying operating system and server application software, unlike on Google App Engine where this is abstracted by the use of an SDK.

### 3.4.3 Application development environment and deployment

On the AWS platform, the virtual EC2 servers closely resemble physical servers from a development point of view. Developers have full control of their servers, they can write to the file system, and customise the servers’ environment. EC2 servers can be used to host either web and non-web applications, unlike on Google App Engine which was solely designed for web applications.

On the platform, developers need to set up the software environment for the application. Python is used to write the application’s code and the application communicates with web servers using mod-python. The mod-python Apache module is downloaded and installed before configuring the Apache web server configuration file to use this module when communicating with the application. Developers can either develop their applica-
tions directly on the infrastructure or on a local machine before uploading their applications to the servers.

When using AWS to host applications, developers have to do a lot of infrastructure development, before actually deploying their applications, unlike on Google App Engine where developers are not aware of the underlying infrastructure. Developers become acquainted with setting up and configuring web servers and the programming languages concerned. A significantly large amount of time is taken in making sure that everything works well than on the GAE platform.

### 3.4.4 Data storage

On the AWS platform developers have a wide range of choices in terms of databases to use for their applications. Some machine images support relational databases such as MySQL and Oracle. S3, EBS and SimpleDB data stores can also be used. For the application, a combination of S3 and SimpleDB databases is used. SimpleDB databases store relatively smaller sizes of data and are optimized for data access speed by indexing the data [42]. S3 databases store raw data and are optimized for storing large sets of data inexpensively.

### 3.4.5 Flexibility

Amazon Web Services offer a combination of computing services that can be customized to build a wide range of applications. There are a variety of EC2 server instances with varying compute capacity, operating systems and application software to choose from. Furthermore, developers are offered full EC2 servers and have full control over the servers, therefore the servers can be customised to suit needs. Applications developed on the AWS platform are aware of the underlying infrastructure. However, the flexibility that comes with this platform means that users have to invest considerable time and work in setting up the infrastructure required to host an application.

### 3.4.6 Scalability

On the AWS platform, developers have to implement a strategy that adds more or terminates some computing resources when system load increases and decreases respectively to automatically scale the system. One option is to use Amazon Autoscale in combination with the CloudWatch services. As previously discussed in section 2.3.6, CloudWatch is a service that is used to monitor EC2 servers in real time, collecting server metrics such as CPU and memory utilization, network traffic and disk reads and writes. The Autoscale service can be configured to use some or all of the metrics collected by CloudWatch to determine the system load of EC2 servers, automatically scaling the system accordingly.
Chapter 3 — A comparative study of cloud-based environments

Another option that can be used to automatically scale applications is to use SQS queues in combination with EC2 servers. This option is used in our application as described in section 3.4 above.

After implementing scalability on AWS, developers are well accustomed with what is involved when automatically scaling web applications, unlike on the Google App Engine where automatic scaling of web applications is abstracted away from developers.

3.5 Measuring the scalability performance of the different hosting solutions

The application built on each of the three hosting platforms is subjected to scalability testing to determine the scalability capabilities of the different platforms. This allows us to verify that the architecture we designed to automatically scale the AWS based application works as intended. We also show that applications hosted on static servers are limited in their abilities to continually serve increasing user loads. The testing involves simulating...
a number of concurrent user requests to the search function.

To run the tests on each platform, we develop a test script that simulates a number of simultaneous requests to the application. The request rates are varied whilst at the same time measuring the average reply rates. On each platform, if the system is scaling well, there should be a linear relationship between the request rates and their corresponding average reply rates.

The graphs in fig 3.3 show the results obtained from the tests run. The y-axis represents the average number of replies per second whilst the x-axis represents the number of request sent per second. From the graphs in fig 3.3, the static server is only able to scale with increasing user requests up to a certain point before reaching saturation point. This behavior is expected from this server as its resources are fixed and the maximum number of concurrent user requests accepted by this server without suffering degradation is limited by the computing resources available.

The Google App Engine application continues to respond favorably to increasing request rates. The good scalability performance by this server is due to the fact that applications hosted on the platform are automatically scaled. On the platform traffic to applications is monitored and if traffic to a certain application increases such that the available computing resources are not enough for the user load, then more computing resources are allocated to the application. However within the free quota, applications can only scale up to preset levels before the platform stops scaling the applications.

The AWS based application is only able to scale up to certain levels of request rates before reaching its scalability saturation point. This may be explained by the fact that the number of slave server instances that can be instantiated in response to increased request rates is limited. The response rates of this application are slightly lower than those of other applications because this application suffers delays when user requests and responses are communicated between the different components of the distributed system. Some delays are also introduced when new slave servers instances boot up when the system is being scaled.

### 3.6 Overall comparison of the hosting solutions

The table below summarizes the differences between the different hosting solutions.
### Table 3.1: A comparison of AWS, GAE and static server hosting solutions

<table>
<thead>
<tr>
<th>Feature</th>
<th>AWS</th>
<th>GAE</th>
<th>Static server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Upfront investment in infrastructure</td>
<td>no</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>External database integration</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Programming languages supported</td>
<td>any</td>
<td>Python and Java</td>
<td>any</td>
</tr>
<tr>
<td>Services offered</td>
<td>Platform as a service,</td>
<td>Platform as a service</td>
<td>Not applicable</td>
</tr>
<tr>
<td>Automatic Scaling of applications</td>
<td>Developers need to</td>
<td>Automatic scaling is</td>
<td>Automatic scaling of applications not possible.</td>
</tr>
<tr>
<td></td>
<td>explicitly configure</td>
<td>built into the Google</td>
<td></td>
</tr>
<tr>
<td></td>
<td>their applications to</td>
<td>platform</td>
<td></td>
</tr>
<tr>
<td></td>
<td>automatically scale.</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Scalability is limited</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>to 20 EC2 servers during</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>the beta period, but</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>developers can apply to</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>acquire more EC2 servers</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deployment manner</td>
<td>Command line tools,</td>
<td>Command line, web based</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Application Programming</td>
<td>administrative console</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Interfaces, web based</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>administrative console</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Access to the server file system</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Flexibility on the type of applications hosted</td>
<td>yes</td>
<td>no</td>
<td>yes</td>
</tr>
<tr>
<td>Ease of use and fast to develop</td>
<td>no</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>
Chapter 4

Requirements of the framework

The process of hosting applications on cloud platforms typically involves provisioning of cloud resources, deployment of applications on the resources and providing an automatic scaling mechanism. As previously stated in section 1.3 the main objective of this project is to build a software framework that provides a mechanism for automating the above mentioned activities. In this chapter we document the functional and non functional requirements of the software framework. The requirements serve as a basis for the design, implementation and testing of the system in later chapters.

In chapter 3 we compared AWS and GAE in terms of hosting a web application. From the analysis done, hosting applications on GAE is relatively easier as compared to AWS application hosting, since on GAE there is already an automated process in place that does the typical activities concerned with hosting applications described in the previous paragraph. Automatic scaling of applications on GAE is abstracted away from users and applications do not need to be aware of its mechanisms.

The Amazon Web Services platform is the platform that is suitable for the development of a proof-of-concept framework for the provisioning of infrastructure and deployment of automatically scaling applications on the infrastructure.

4.1 Technical requirements

The framework should satisfy the following technical requirements:

1. The system should deploy scalable web applications on Amazon Web Services and users should be able to specify how their applications should be scaled.

2. The system should support the deployment of Python based web applications using the mod_python web interface.

3. The framework should be accessed through a web interface.
4.2 Functional requirements

Fig 4.1 shows all the developer’s possible interactions with the system. The use case descriptions further clarify what is involved in each use case.

### 4.2.1 Use case: Logging into the application

- **Actor**: Developer
- **Goal in context**: To log into the application using AWS keys, so that the developer does not need to re-enter security credentials on subsequent browser requests.
- **Trigger**: Developer navigates to the application without having any cookies from the application set in the user browser.
- **Exceptions**:
  1. Developer submits the login form with some fields missing.
  2. Developer enters wrong keys.

In all cases the application should re-display the login form indicating the errors.

### 4.2.2 Use case: Viewing running applications

- **Actor**: Developer
- **Goal in context**: To display all the applications deployed by the developer using the framework including running servers and their system loads.
• **Precondition:** Developer should be logged on.

• **Trigger:** Developer clicks the view running applications link

• **Exceptions:**

  1. Developer does not have any running application, in which case a message should be displayed indicating that there are no running applications.

### 4.2.3 Use case: Deploying an application on AWS

• **Actor:** Developer

• **Goal in context:** Provision AWS infrastructure to host the application, deploy the application on the provisioned infrastructure and enable automatic scaling of the deployed application.

• **Preconditions:** Developer should be logged on.

• **Trigger:** The deploy an application link is clicked.

• **Scenario:**

  1. Developer chooses the type of application to be deployed.
  2. Developer supplies an application identifier.
  3. The developer’s application files are supplied.
  4. The application is deployed and the operation’s feedback should be displayed.

• **Exceptions:**

  1. The application identifier supplied already exists.
  2. The application identifier does not consist of alphanumeric characters only.
  3. Application files are not provided in the correct format.
  4. Any of the above missing.

Users should be prompted to re-enter the information in the event of any of the above. Appropriate error messages should be displayed.
4.2.4 Use case: Rebooting running servers

• **Actor:** Developer

• **Goal in context:** To reboot any of the running servers.

• **Preconditions:** Developer should be logged on.

• **Trigger:** The reboot a server link is clicked.

• **Scenario:**

1. The application displays all running applications and the servers belonging to the applications.
2. The user chooses the server to reboot.
3. The server is rebooted and the feedback of the action performed is displayed.

• **Exceptions:**

1. User does not have any application running.

An appropriate error message should be displayed.

4.2.5 Use case: Terminating a running application

• **Actor:** Developer

• **Goal in context:** To terminate a running application.

• **Precondition:** Developer should be logged on.

• **Trigger:** Developer clicks the terminate an application link

• **Exceptions:**

1. Developer does not have any running application, a message should be displayed indicating that there are no running applications.

4.3 Non-functional requirements

Non-functional requirements are those requirements that describe properties or qualities that a system must have [24]. The requirements also specify the constraints under which a software system must operate. In this section we describe the non-functional requirements of an application deployed by the framework and the framework itself.
Chapter 4 — Requirements of the framework

4.3.1 Framework application

In this section we describe the non-functional requirements of the framework application.

Easy to use and learn: The web interface should be easy to use and developers should easily learn how to use the interface. Guidance on possible actions should be offered throughout a developer experience.

Easy navigation: Developers should easily be able to navigate from one web page to another. Each web page should be reachable from all the other web pages.

Developer Security: Communication between developers’ browsers and the framework application should be secure to safeguard their sensitive information like AWS credentials. Also to minimise risks associated with identity theft, developer credentials should not be stored in browsers in any form.

Input validation: All submitted input should be validated before being used. Appropriate error messages should be displayed should there be any input errors.

Feedback: Feedback on all performed actions should be displayed to developers. Also if some processes take longer than normal to execute, developers should be given an indication of the progress of the execution.

4.3.2 Deployed application

The deployed application should be based upon the architecture described in 3.4.1. We choose this architecture, since it enables us to fully understand how scalability works.

Response time: An application deployed using the framework should have average response times of less than a minute to minimise user waiting time. After suffering degradation in response times owing to increased loads, the automatic scaling system should quickly automatically scale and recover the application.
Chapter 5

System design and implementation

In this Chapter we describe how the software framework is designed to meet the requirements documented in chapter 4. We start by describing the design of an application that is deployed by the framework before describing the framework application’s design.

To implement the framework we choose as the Python programming language. Compared to other programming languages such as Java and C++ (static typed), Python is a high level language, which means many low level details are handled automatically such that one can write the same C++ or Java code in fewer lines [34, 38]. This means Python code is shorter and faster to write and is easier to understand. However using scripting languages such as Python results in slower execution speeds, unlike in system languages such as C++ where compiled code is optimised [34].

5.1 Deployed application

The framework is designed to automatically set up an application based on the architecture described in section 3.4.1. Fig. 3.2 shows this architecture. The architecture consist of an interface, master and a group of worker slaves. In this section we describe in detail the modules contained in these servers and how they are designed to execute the functions of the server.

5.1.1 Master server

The master server component consists of various subcomponents that work together to automatically scale the deployed application. Fig 5.1 shows the components of the server. The Boto package component is used for AWS access as explained in section 5.2.1.
Figure 5.1: The components of the master server
Figure 5.2: Interaction diagram of the master server components.
aws_access.py

The (aws_access) Python script contains a class (aws_access) that encapsulates various methods and data used to send requests to the Boto package that implements the web services interface to Amazon Web Services. Fig 5.2 shows how the aws_access class interacts with the other software components of the server.

main.py

The main.py script contains Python code that executes the main tasks of the server shown in fig 5.3. The script instantiates an aws_access object and uses the object to get the current number of SQS messages in the SQS queue, instantiate EC2 servers, terminate EC2 servers, get the number of pending EC2 servers that are still booting up and the number of running EC2 servers.

cpuinfo.py

The cpuinfo.py script serves HTTP requests for obtaining the CPU load of the server as explained later in section 5.2.1.

appname.py

The appname.py script serves HTTP requests for getting the identifier of the application under which the server is running. This is explained in section 5.2.1.

get_user_data.py

The get_user_data.py script is run automatically on server boot up and contains Python program logic that downloads the user-data sent by the framework application during the instantiation of the server. The downloaded user-data consists of server configuration data. Various configuration parameters are retrieved from the downloaded data and saved into a configuration script that can be accessible by the other application files. The following is a list of the parameters retrieved from the user-data:

- **AWS keys**: Identifies a particular developer when communicating with AWS.

- **application_identifier**: The application identifier identifies each AWS resource allocated to a deployed application such as:
  
  - Interface EC2 server.
  - Slave servers.
  - Master server.
Chapter 5 — System design and implementation

- Request SQS queue.
- SimpleDB domain.

- **slave_ami**: This is the AMI id of the slave server. It is used to instantiate more slave servers when automatically scaling the system.

- **threads**: The number of concurrent request processing threads to run on each of the slave servers. This number of threads is specified by a developer during the deployment of the application. Together with the AWS keys and application_identifier, the master server sends this as configuration data to instantiated slave EC2 servers when automatically scaling the system.

- **min_ec2_instances**: Developers should specify this number. The master server should keep this number of slave servers running even if there are no user requests in the request SQS queue.

- **threshold_requests_per_running_slave**: We use this threshold value to implement the automatic scaling strategy and it should be supplied by developers. The parameter is further described in section 5.1.1.

**keys.py**

The `get_user_data.py` Python script saves the server configuration data in the `keys.py` Python script. This script provides the data described above to other scripts in the server.

**Automatic scaling of the system**

When automatically scaling the system, the master server should balance between instantiating enough EC2 servers to serve user requests whilst at the same time making sure that computing resources are not wasted by terminating some EC2 servers that are lying idle. First, there is need to find a way of measuring system load. Secondly we need to define an effective criteria for adding more or removing some EC2 servers based upon the system load and the criteria should balance the need to maintain enough and the need not to waste EC2 resources.

**Measuring system load**: There are several metrics that can be collected and used to measure the system load such as:

- Number of incoming requests.

- Average response time of the system.

- CPU or memory utilisation on running slave servers.
Chapter 5 — System design and implementation

In the master server, to measure system load, we determine the number of incoming requests to the system. Amazon SQS provides us with a simple way to determine the number of incoming requests.

**Criteria for instantiating more or terminating some slave EC2 servers:** When determining when to instantiate or terminate slave EC2 servers we look at the current number of request messages in the request queue and compare this number to the current number of running slave servers. We express this comparison as the number of requests per each running slave server \( \text{requests per running slave} \). We define a threshold value for the \text{requests per running slave} at which if exceeded more slave servers should be instantiated and likewise, if the current \text{requests per running slave} is less than the threshold, some slave servers should be terminated. This threshold value should differ from application to application as different applications have different processing and resource needs. For example, this ratio would probably need to be 1 for a high processing application that would require a high number of CPU cycles and amount of memory when processing requests, whilst this value would probably need to be much more for applications with short bursts of processing. Developers should use trial and error playing around with this value until they find a value that best suits their applications.

**Instantiating slave servers:** The \text{instantiate_slave} operation of the \text{aws_access} class instantiates new slave server instances using the slave AMI id from the server configuration data saved in the \text{keys.py} Python script. When instantiating the slave servers, like on the framework application, server configuration data should be sent to the servers. The configuration data consists of AWS keys, threads and application identifier parame-
ters from the keys.py and is sent as user-data to the instantiated slaves.

5.1.2 Interface server

Fig 5.4 shows the various components contained in the interface server. The cpuinfo.py, keys.py and appname.py Python scripts and the Boto component in this server serve the same purposes as in the master server.

if_server.py

Every time the developer presentation logic receives a user request two HTTP connections are created: one to the SQS queue service to send requests to the request SQS queue and another to the SimpleDB service for receiving processed responses from the slave servers. If the interface server is busy, it frequently creates and disconnects HTTP connections when communicating with AWS. As specified by TCP, after HTTP connections close, they remain in the TIME_WAIT state for some time to ensure that delayed packets arrive on the correct connection [20]. HTTP connections incur memory overheads from TCP and this can adversely affect the responsiveness of a server since the memory requirements are directly proportional to the number of connections. The frequent creation and closing of a large number HTTP connections results in a large number of connections in the TIME_WAIT state depleting memory resources, eventually causing the server to be unresponsive.

To avoid problems associated with opening too many HTTP connections, we create an HTTP connection for each AWS service that we need to connect to once, and reuse these connections for all requests that need to be processed. We implement a Python HTTP server (if_server) that listens for requests from the developer presentation logic. The
server creates single instances of HTTP connections to the SQS and SimpleDB services and these connections are reused for all requests received from the presentation logic.

The if_server.py script contains Python code that implements the if_server. The server receives user requests sent from the developer presentation logic before passing them to the request SQS queue. For each request sent to the request queue, the server monitors for the corresponding response in the SimpleDB database, before passing back the response to the client (developer presentation logic) that initiated the request.

**Communicating requests and responses:** Messages stored in an SQS queue are strings of up to 8kB by default (this size can be changed by developers). To pass on requests to the slave servers using the request SQS queue, a string consisting of a request id that uniquely identifies that particular request and a command (representing the user request) that invokes the business logic on the slave servers is constructed.

After sending the request message to the request queue, the if_server monitors for the processed response in the SimpleDB database. During the deployment of an application, a SimpleDB domain (database table) with a name corresponding to the application identifier of the application is created. The slave servers use this domain for communicating responses to the interface server. As previously discussed in section 2.3.4 data is stored as items with each item identifiable by an item name in SimpleDB domains. A slave server retrieves the request id and request from a request message, processes the request and saves the processed response in the responses domain as an item with the retrieved request id as the item’s name. The if_server logic monitors for a request’s response from the responses domain by looking for a response item with a name corresponding to the request id it assigned to the request.

On the slave servers, the developer business logic is invoked by running the command received from the message read from request queue. Section 5.1.3 describes the command expected from the interface server.

**send_to_if_server.py**

This script contains a Python function send(request) that interfaces developer presentation logic to the HTTP server implemented by the if_server.py Python script described above. The presentation logic calls the send() function in this script passing a user requests as an argument and the function sends the request to the if_server before waiting for the response. The function returns the response received from the if_server to the calling presentation logic.
Chapter 5 — System design and implementation

Figure 5.5: The components hosted by the slave server.

Developer presentation logic

The presentation logic of a developer application described in section 3.4.1, receives a browser request and formats the request so that it can be send to the request queue for processing by the slave servers. The logic uses the `send(request)` function of the `send_to_if_server.py` Python script to send the request to and receive the response from the `if_server`

`get_user_data.py`

Like in the master server, the `get_user_data.py` Python script is run automatically during boot up and it downloads the server configuration data sent by the framework application during the instantiation of the server. The configuration data for this server consists of the following parameters :

- **AWS keys**: Needed when communicating with AWS.
- **application_identifier**: used to identify various AWS resources.

The script also downloads and unzips developer presentation logic zip file uploaded to the S3 application files bucket by the framework application.

5.1.3 Slave server

The server consist of several software components as shown in fig 5.5. The `cpuinfo.py`, `appname.py` and `keys.py` Python scripts function the same way as the same scripts in the interface and master servers.
main.py

This script contains program logic that performs the various functions of the server. The logic reads a message from the request queue and instantiates an object of the slave class and uses the object to process the request and send the processed response to a response queue where it will be read by the interface server. The main.py Python script runs automatically on server boot up and continues to run until the server is terminated.

Threads: In the script we make use of threads to separate the reading of messages from the request queue and the processing of a received request. There is a main thread that reads a message from the request queue and creates another thread responsible for processing the request read from the queue. For each request read, a separate thread is created for processing the request. This allows the main program on the slave server to continue picking up messages from the request queue even if the other threads are still running. This might be helpful to some applications with less CPU processing involved, as having more concurrent processing threads means that a slave server will process more requests per given time. However, for high processing applications, having more concurrent processing threads further increases the CPU processing time, sometimes grinding the system to a halt. This means the optimum number of concurrent processing threads on the slave servers differs from application to application. We let developers define this value when deploying their applications on the framework application. This value is send to the server with the other server configuration data as user-data during the instantiation of the server in the framework application.

slave.py

The slave.py Python script contains a Python class (slave) that encapsulates data and operations used to process requests read from the request queue and send responses to the responses queue. Processing of requests is done by invoking the developer business logic. Fig 5.6 shows how the slave class interacts with the other software components hosted in this server.

get_user_data.py

Like the other servers, the slave server contains a get_user_data.py Python script for downloading configuration data sent during instantiation, before retrieving various parameters from the downloaded data and saving it into a configuration file accessible by all other application files. The following parameters are retrieved from the data:

- **AWS keys**: Needed when communicating with AWS.
- **application_identifier**: Used to identify various AWS resources
Figure 5.6: Interaction diagram of the slave server.

- **threads**: The number of concurrent threads to run on the slave server.

Like on the interface server, the script also downloads and unzips the developer business logic zip file from the S3 application bucket.

**Developer business logic**

The developer business logic, previously described in section 3.4.1 is the part of a developer application that does most of the application’s processing. This logic is invoked from the `main.py` Python script using an object of the slave class. From the slave class operation that invokes this logic, we assume that the logic is contained in a programming script that is callable from the command line. The script accepts command line arguments. For example a `primes.py` Python script containing an application’s business logic for calculating the prime factors of a number is deployed on the slave server. The following code shows the command expected from the interface server and how the business logic is called from slave class operation:

```python
command = "python primes.py 16888"
fout = os.popen("command");
response = fout.read()
```

From above, 16888 is the number whose prime factors are calculated.
5.2 Framework application

In this part of the chapter we describe the design of the framework application responsible for setting up the architecture of developer applications described in the previous sections.

5.2.1 Architecture

To simplify the design of the framework application, it is divided into a number of functional components that work together to achieve the requirements of the system. The different components are hosted by an EC2 server. Fig 5.7 shows the various functional components of the system.

The controller component interfaces the other components to developer browsers. The component passes requests to other components based on the request received from a browser. The boto package interfaces the rest of the application to AWS.

Boto component

On the AWS platform, computing resources are exposed as services through a web services interface [2, 42]. AWS provides command line and web based tools that implement the web services interface and the tools can be used to communicate with the services on the platform. For communication with AWS from inside programming code, there are various API packages for different programming languages provided. Boto is a Python API package consisting of several Python API classes that implement the web services
interface [10]. Boto consists of Python API classes that can be used to access the following AWS services:

- SimpleDB
- Simple Queue Service (SQS)
- Elastic Cloud Compute (EC2)
- Simple Storage Service (S3)

Boto is downloaded and used without any modification.

**FrameworkApp component**

To re-factor common recurring activities concerned with the provisioning of AWS resources, deployment of applications on the provisioned resources and management of deployed applications and their resources, a Python object-oriented class that interfaces with the Boto API is implemented. This makes the programming task easier as the code from this component is re-used.

The frameworkApp component consists of this class. This class encapsulates data members and class operations that are used to communicate with Boto classes during deployment, termination and management of AWS services. Fig 5.8 shows the class and how it relates to other components. The class’s data members are described below.

- **secret_key** and **access_key**: Passed on to methods in the Boto classes used to create connections to services on AWS. The keys are used to identify and validate a particular developer as a sender of a request.

- **SQS_conn**: A connection object used to pass requests to SQS.

- **EC2_conn**: A connection object for communicating with the EC2 service.

- **S3_conn**: A connection object for S3 communication.

- **SimpleDB_conn**: A connection object for communicating with the SimpleDB service.

- **app_identifier**: To identify an application deployed by the framework each deployed application should be allocated an application identifier. This identifier is associated with all the resources running under a particular application and enables us to identify the resources. When deploying an application, a developer should supply an identifier that is unique within the AWS account. This identifier should consist of alphanumeric characters as it is used to name the application’s AWS
resources such as SQS queues, SimpleDB domains and S3 buckets. Amazon Web Services specifies that SQS queue and S3 bucket names should consist of alphanumeric characters only [11, 7].

In the following sections we describe the class’s operations:

**run_server(ami_id, keypair):** The class function instantiates a new EC2 server instance using the first argument as the Amazon machine image with which to launch the server. As previous discussed in section 2.3.1, an AMI contains a server’s software configurations such as operating system, application software and associated settings that are used to boot up an EC2 server.

To control SSH access into instantiated EC2 servers, AWS uses RSA keypairs [6]. Developers can generate keypairs using a web interface provided by AWS. The generated keypair can be downloaded and be passed on to an EC2 server that is being instantiated. The same keypair is then used to SSH into the instantiated server. The `run_server()` class function takes the keypair as a second argument.

The EC2 service allows developers to send a server a start-up script or configuration data during the instantiation of the server [6]. This data is called **user-data** and is
uploaded to an AWS web server where an instantiated server can fetch it. All instantiated
EC2 servers can download their user-data from the following URL:

http://169.254.169.254/latest/user-data

The run_server() operation sends data that contains AWS credentials (for communicat-
ing with AWS), an application identifier (later on used to associate the server with the
application it belongs to) and other server-specific information as user-data to instan-
tiated servers. The instantiated servers later on fetch this data and save it in a server
configuration script that is accessible by other application files.

upload_files_to_S3(file_object, bucket_key): This operation transfers application files
to the appropriate S3 bucket, where they are collected by instantiated slave and interface
EC2 servers. In order to deploy a developer’s application files, the files should be separated
into two folders. The first folder contains the application’s presentation logic files and the
other folder contains the business logic files that need to be scaled. These two parts of an
application are described in section 3.4.1. The two folders should be compressed before
being uploaded as zip files. After the application files are received from a developer, they
are transferred to an S3 bucket where the relevant servers (interface or slave) collect the
files. S3 bucket names should be unique across the whole of the S3 domain, as described
in section 2.3.3. In order to minimize the risk of using a bucket name that already exists,
the following bucket naming convention is used:

bucket_name = application_identifier+ \n            hashlib.sha224(access_key_id).hexdigest()

Both of the two compressed folders are uploaded to the same bucket but with different
bucket key names. The same key names are used to download the application files from
S3 on the interface and slave servers.

create_request_queue(): This class function creates the SQS queue used for the trans-
fer of requests from the interface server to the slave servers. In order to identify SQS
queues, the queues should be allocated names. Amazon SQS names should be unique for
the AWS credentials used. To differentiate between different SQS queues for the same
developer, the queues are named using the following naming convention:

queue_name=’request_queue’+application_identifier

create_domain(): The class function creates the SimpleDB domain used to communi-
cate request responses from the slave servers to the interface server.
**stop_server(server_dns_name):** The operation terminates the EC2 server with the specified DNS name.

**reboot_server(server_dns_name):** The operation reboots the server with the specified DNS name.

**list_running_applications():** This class method is used to get a list of all applications running under the developer’s AWS keys, deployed using the framework application. Also, all the EC2 servers running under each application are listed. The following Python code populates a list with all the EC2 servers running under a particular developer’s AWS keys.

```python
EC2_instances=[]
reservations= EC2conn.get_all_instances()
for r in reservations:
    for i in r:
        EC2_instances.append[i]
```

To help identify the application under which an EC2 server is running, we configure a web server on all the servers of the application. On the document root of the server there is a web script that writes out the application identifier of the application under which the server is running as an HTTP response. Below we show the Python code that does this. The application identifier is contained in a Python configuration script (`keys.py`).

```python
req.write(keys.application_identifier)
```

The **list_running_applications()** class operation sends an HTTP request to the web page to get the application identifier of the server.

**terminate_app(application_identifier):** The terminate operation terminates all AWS resources running under the application identified by the application identifier passed on in the function parameter. The function uses the **list_running_applications()** operation described above to obtain a list of all the EC2 servers running under the application identifier, before terminating the servers. As described in section 5.2.1, the request SQS queue, SimpleDB domain and S3 buckets of an application are named in a format that uses the application identifier under which they are running. This makes it easy to terminate the resources, as we simply name the SQS queue, domain and S3 buckets belonging to the application using the prescribed format, before terminating them.
Figure 5.9: *The program flow for the controller module.*

**Controller component**

The controller component receives all requests to the application and coordinates all the functions of the framework application. The activity diagram in fig 5.9 shows the flow of activities in the controller component.

The controller is implemented in a Python script which consists of Python functions that implement the activities shown in the flow diagram in fig 5.9. Fig 5.10 shows the various functions and variables of this component.

**Login processing:** Developers are logged into the application using the AWS keys (access key and secret access key) issued when they sign up to use AWS. On the application we validate AWS credentials through AWS, since only AWS can determine the validity of these credentials. To validate the AWS keys, we send a request to AWS using the credentials. AWS rejects requests sent with invalid access/secret key combinations. Therefore, we do not persistently store developers’ credentials, but validate the credentials with AWS every time a developer logs in. To login developers, the application displays a login form where users can enter in and submit the AWS keys. Fig 5.11 shows a screen shot of the login page showing the login form. The login function of the controller processes the following activities of the controller program flow shown in fig 5.9.
Figure 5.10: A UML component diagram showing the controller’s data and functions.

Figure 5.11: A screen shot of the login page.
Chapter 5 — System design and implementation

- Determining if there is login form data in the request.
- Displaying the login form if there is no login form data in the request.
- Validate developer submitted login form data if the data is present in the request.
- Redisplaying the login form and errors if there are any.
- Logging in the developer if developer input is valid.

The program logic verifies that there are no missing entries on the submitted login form. The form should be submitted with both the AWS keys filled in. If there are no missing inputs, the logic further verifies that the entered AWS keys are valid. This is done by trying to send a request to the AWS SQS service, to get a list of all the developer’s running SQS queues. If the AWS keys are wrong, AWS will return an error message indicating such, otherwise the request goes through.

After the login form data input is validated, the user is logged in by setting a non-persistent cookie on the developer browser. The cookie contains the session id of the developer and is retrieved on subsequent requests, thus maintaining the user’s login status. The session id and the corresponding AWS keys are temporarily stored in a database and on each subsequent request during a session, the session id is retrieved from the cookie and used to look-up the AWS keys. The non-persistent cookie is deleted when the browser is closed, and a javascript script sends a request to the server to delete the entry with the corresponding session id in the AWS keys database.

**Determining login status:** The controller accepts the request received from the browser as a parameter. It looks into the received request for a cookie containing the session id of the current session. If the cookie is set, the application further verifies if the session id in the cookie exists in the AWS keys database. If it does, then the user is logged-in, otherwise the user is not logged in. The AWS keys are looked-up in the database using the session id and used when sending requests to AWS.

**Passing requests to other components** The controller differentiates requests for different functions of the application using a query string parameter in the URL. This parameter specifies the function that is being requested. For example, the URL of a request for the **deploy_an_application** page ends with **?function=deploy**. The controller determines the value of the query string parameter **function** in the received request and uses this to determine the function to call for processing the request, the called function processes the request before returning the results to the controller. The code in the example below shows this.
function=req.form[‘function’]
if function=None:
    display_default_page()
if function==’view’:
    content=view.view(req)
if function==’deploy’:
    content=deploy.deploy(req)

Figure 5.12: The program flow for the **deploy** module.

The controller passes on requests to different component functions using **call** and **return** commands, whereby a main program calls a subprogram, passing on arguments to it. The subprogram then processes the received request before passing control back to the main program. In this case, the controller is the main program, which calls other subprograms (other framework components), passing the request to be processed as a parameter. The called component processes the request before returning the processed information content to the controller, which displays it to the browser.

**Deploy component**

The deploy component processes all requests concerned with the deployment of an application. Fig 5.12 shows the program flow of the deploy component and fig 5.13 shows a screen shot of the application deployment web page.
Figure 5.13: A screen shot of the application deployment web page.
Verifying user input: To deploy an application a developer should supply the following on the application deployment form:

- Application identifier consisting of alphanumeric characters.
- Application presentation logic zip file.
- Application business logic zip file.
- The RSA keypair which is sent to a server that is being instantiated. The keypair enables a developer to SSH into an instantiated server.
- The number of concurrent threads to run on each running slave server.
- The minimum number of slave servers to run even if the system is idle.
- The number of requests in the request queue per running EC2 server. If the current requests per running EC2 server is more than this limit, more EC2 servers should be instantiated and, likewise, if the current requests per running EC2 server is less than this limit, some EC2 servers should be terminated.

The program logic verifies that:

- There are no missing fields when the deploy form is submitted.
- The application identifier does not contain non alphanumeric characters.
- The presentation logic file is a zip file.
- The business logic file is a zip file.
- The number of concurrent threads to run on the slave servers is an integer.
- The minimum number of slave servers to run even if the system is idle is an integer.
- The number of requests per running EC2 server is a float or integer.

If the submitted form data is not valid, the deploy form should be redisplayed indicating the errors that have been made.

Provisioning of AWS resources and deployment of an application: To deploy an application a frameworkApp object is instantiated. This object is used to instantiate the EC2 servers (run_server()), transfer application files to S3 where they are collected by the slave and interface servers (upload_file_to_S3()), create the SimpleDB domain for communicating responses from slave servers (create_domain()) and create the SQS queue used for communicating requests from the interface server to the slave servers.
Figure 5.14: The program flow for the **terminate** module.

(create_queue()). After the provisioning of the AWS resources and the transfer of application files to S3, the DNS names of the instantiated EC2 servers should be displayed. These DNS names identify each of the EC2 server instantiated under the application.

**Terminate**

The terminate component handles all activities concerned with the termination of an application and all its associated AWS resources. Fig 5.14 shows the flow of the program logic in this functional component. There are two execution paths possible and the component determines the execution path to follow by looking for the application identifier parameter in the received request URL. If this parameter is found, then all the AWS resources of the application identified by this parameter are terminated. However, if this parameter is not found, then the program logic gets a list of all of a user’s running applications and displays the application identifiers as links that can be clicked to terminate the application in question. The example below shows an example of such a link.

```html
<a href='/terminate?application_identifier=primes'>Primes</a>
```

Like the deploy component, the terminate component uses a **frameworkApp** class object to send requests to AWS.

**Reboot**

The program logic in the reboot component works in a similar manner to the **terminate** component. To determine the execution path to take, it looks for the **ec2_dns_name** parameter in the received URL. If the parameter is present then it reboots the server with
the identified DNS name. If the parameter is not present, the program logic gets a list of all of the user’s running applications and the EC2 servers running under the applications and displays the servers’ DNS names as links which can be clicked to reboot the EC2 server in question. Fig 5.15 shows the flow of the program logic. Like the terminate component, the component uses an object of the frameworkApp class to communicate with AWS.

**Viewing running applications**

The program logic in this component obtains a list of a user’s applications and the EC2 servers running under the applications together with average CPU loads of the EC2 servers over the last 5, 10 and 15 minutes. Each EC2 server is displayed under the application identifier of the application it is running under. Fig 5.17 shows a screen shot of the view_running_applications web page.

The component uses the frameworkApp component to communicate with AWS. Fig 5.16 shows the flow of the program logic in this component.

**Obtaining the CPU loads of the servers:** As previously described in section 5.2.1, a web server is configured on each instantiated EC2 server. In order to obtain the CPU load of a server, on the document root of the server there is a web page script that obtains the CPU load of the server in the last 5, 10 and 15 minutes before writing out this as an HTTP response. An HTTP request is sent to the script on the server to obtain this information.
Figure 5.16: The program flow for the `view_running_applications` module.
Figure 5.17: A screen shot of the view_running_applications web page.
User Interface

The application is accessible through a web interface. The web pages are designed such that the application is easy to use, learn, and navigate.

Layout  All the web pages of the application uses a similar page layout, offering a consistent feel throughout a user experience. The diagram in fig 5.18 shows the basic layout of all web pages of the application.

The menu bar appears on all the web pages. The items displayed in the menu bar differ depending on whether one is logged in or not. If a user is not logged in, only two items are displayed in the menu, as shown on the web page displayed in section 5.2.1 fig 5.11. The first item represents a link to the about page, where a brief description of what the application is about is given. However, if a user logged in, the menu includes items with links to all the other functions of the application. As with the menu bar, the masthead appears on all the web pages of the application.

HTML information displayed in the main area of the web page differs according to the requested page. The controller component described in section 5.2.1 displays informational content generated by the other function components of the application in this section.

Navigation:  Fig 5.19 shows the hypermedia structure of the application. A user of the application should first be logged on before accessing the functions of the system. After logging in, all the web pages of the application can be reached from any page using the hypertext links displayed in the menu. Also users can logout of the application from any page. This offers users flexibility in navigation.
Figure 5.19: The web page navigation structure of the application.
Chapter 6

Testing

In this chapter we perform tests to verify that the framework and the deployed application work correctly according to the requirements documented in chapter 4. For both the framework and deployed applications, we first start by testing the low level functions of the system, gradually testing functions at more higher levels until we test the whole integrated system. For the deployed application we conduct experiments to determine the scalability performance of the system.

6.1 Framework application

The framework application consists of the controller and several other components as described in section 5.2.1. Fig. 5.7 shows the framework application’s architecture.

6.1.1 Controller

The controller module contains code that performs the following functions:

- Login and logout processing.
- Displays the default page.
- Parses the path in the received request to determine which model component to send the request to, before passing on the request to the determined model component.

Login and Logout processing

Objective: The objective of this experiment is to determine if the login and logout processing functions of the framework application work correctly.
Protocol: During the login and logout processes, the user interacts with the browser which in turn communicates with the controller component of the application. The controller contains the program logic that processes the login status of a user. The interaction diagram in fig 6.1 shows the basic processes performed to validate the correctness of these functions.

Results: The application rejects missing inputs and incorrect AWS keys when logging on. On submitting the login form with the correct AWS keys a cookie containing the current session id is saved on the browser and the user logged in. After logging in, links to various functions are displayed and the login status is maintained on subsequent requests. The browser cookie is deleted and the login page is displayed when the logout link is clicked.

Conclusion: The login and logout functions of the controller module work properly.

Figure 6.1: Testing protocol for the login process.
Chapter 6 — Testing

Passing requests to model components

Objective: The objective of the experiment is to validate that received requests are passed on to appropriate function components.

Protocol: The controller uses the path in the browser request to determine the function component to pass on the request. To carry out the tests we first log into the application. We then click the various links displayed to navigate to the different web pages of the application. The UML interaction diagram in fig 6.2 shows the steps followed to test the correctness of this function of the controller module.

Result: All the requested web pages are appropriately displayed as requested.

Conclusion: We conclude that the controller appropriately passes request to different function components. Also from this we conclude the navigation links in the web application work properly.

6.1.2 Deploying an application

The tests in this section verify that the framework correctly provisions AWS infrastructure and deploys applications on the infrastructure. To carry out the tests, we log into the application before navigating to the deploy an application web page. During deployment, the framework performs the following subfunctions:

- Gets user input.
- Verifies user input.
- Transfers application zip files to S3.
- Instantiates interface, master and slave EC2 servers.

Getting user input

Objective: The objective of this experiment is to verify that the data entered on the deploy an application form by a user is captured and transferred to the web server correctly.

Protocol: In order to verify that this function of the deploy module works correctly, we display back to the browser the text data entered on a deploy an application input form. We also verify that application zip files are correctly uploaded by saving the files on the server and checking if we can see the saved files. The interaction diagram in fig 6.3 shows the steps followed when carrying out the experiment.
Figure 6.2: Testing protocol for passing requests to various model functions.

Figure 6.3: Testing protocol for the \texttt{get\_user\_input} function of the \texttt{deploy} module.
Chapter 6 — Testing

Results: Captured text user input is displayed correctly. The unzipped application files can be seen and opened from the location where they are saved.

Conclusion: We conclude that this function works correctly.

Verifying user input

Objective: Before the captured form data is used, it should be verified and correct error messages should be displayed should any of the input not be of the expected type and format. This function should test the following:

- All the input is entered before the form is submitted.
- The application identifier consists only of alphanumeric characters.
- The number of concurrent threads to run on the slave servers is an integer.
- The minimum number of slave servers to run even if the system is idle is an integer.
- The number of requests per running EC2 server is a float or integer.
- Uploaded application files are zip files.

The objective of the tests is to verify that the function correctly validates user input before the application uses the data.

Protocol: The interaction diagrams in figures 6.4 and 6.5 show the protocol followed during the experiment.

Results: All invalid user inputs are correctly picked up and appropriate error messages are displayed.

Conclusion: From the above results we conclude that the application correctly validates user input when deploying an application.

Uploading application zip files to S3

Objective: The objective of the test is to validate that the framework application correctly uploads application zip files to the application files S3 buckets during the deployment of an application. S3Fox is a FireFox browser extension that uploads or downloads files to and from S3. It displays users’ buckets as folders, with each folder containing uploaded objects identifiable by a key. Each bucket folder is identifiable by the bucket name assigned when the bucket is created. We use S3Fox to view application zip files
Figure 6.4: Protocol followed when testing the `validate_user_input` function of the `deploy` module.
Figure 6.5: Continuation of the protocol followed when testing the `validate_user_input` function of the deploy module.
uploaded to S3. As previously mentioned in section, 5.2.1, application buckets are named using the naming convention below:

\[
\text{bucket}\_\text{name} = \text{application}\_\text{identifier} + \backslash \\text{hashlib.sha224(}\text{access}\_\text{key_id}).\text{hexdigest()}\]

**Protocol:**

1. Navigate to the deploy an application page and fill in the displayed form correctly before submitting it.

2. When the framework application displays the DNS names of instantiated servers, verify that the application zip files are uploaded to the application files bucket using S3Fox.

**Results:** The application files can be viewed from the application files bucket using S3Fox.

**Conclusion:** From the experiment, we conclude that the application properly uploads application zip files to S3.

**Instantiating EC2 server instances**

**Objective:** In this section we test if the framework properly instantiates all the EC2 servers required by an application during the deployment of an application.

**Protocol:**

1. Navigate to the deploy an application page and fill in the displayed form correctly before submitting it.

2. Verify if the servers are correctly instantiated by opening an SSH session to each server.

**Result:** We are able to open an SSH session to each of the instantiated servers using the DNS names supplied by the framework.

**Conclusion:** From above we conclude that the framework can properly instantiate all the required EC2 servers.
6.1.3 Viewing running applications

**Objective:** The objective of the experiments done in this section is to verify that all of a developer’s running applications and the EC2 servers running under the applications, deployed using the framework are properly displayed when the view running applications link is clicked.

**Protocol:** We first log into the application before carrying out the following tests:

1. Click the view running applications link, without any applications running.
2. Verify that the application indicates that there are no running applications.
3. Deploy an application on AWS using the framework.
4. Click the view running application link.
5. Verify that the framework application list the running application and all the running EC2 servers belonging to the application.
6. Deploy another application using the framework.
7. Click the view running application link.
8. Verify that the framework displays both applications with all their running EC2 servers.

**Result:** When the view running applications link is clicked without any applications running, the framework indicates that there are no running applications. The framework also list all the running applications and their EC2 servers.

**Conclusion:** From the above we conclusively validate that the view running applications function works properly and correctly.

6.1.4 Terminating an application

**Objective:** The objective of the tests conducted is to verify that the application can properly and correctly terminate EC2 servers and delete S3 buckets and SQS queues belonging to an application that is terminated.
**Protocol:** To run the tests, we first log into the application. The function should first list all applications belonging to a developer as links when the terminate an application link is clicked. When a listed link is clicked, the application with the displayed name should be terminated. ElasticFox, another FireFox extension that is used to instantiate, terminate and display the status of EC2 servers, is used to determine that the function correctly terminates EC2 servers.

To determine if SQS queues are terminated we develop a Python test script that gets and displays the names of SQS queues running under the AWS keys used. We identify SQS queues belonging to the application by looking at the queue names of the displayed queues. As previously stated in section 5.2.1, request SQS queues are named using the following format:

```
queue_name = 'request_queue' + application_identifier
```

To verify that all S3 buckets belonging to the application have been deleted, we use S3Fox as previously described in section 6.1.2.

**Testing procedure:**

1. Deploy several applications using the framework.
2. Click the terminate an application link.
3. Verify that all the running applications are displayed as links.
4. Identify an application to terminate and note the application identifier.
5. Run the Python test script and verify that SQS queues belonging to the application are listed.
6. Click on the link belonging to the identified application to terminate the application.
7. Verify that the application and its EC2 servers are no longer listed using the view running applications function.
8. Verify that the terminated application’s EC2 servers are no longer listed in Elastic-Fox.
9. Run the Python test script to verify that there are no SQS queues belonging to the application.
10. Use the S3Fox to verify that there are no S3 buckets belonging to the application.

**Results:** After terminating an application, there are no EC2 servers running under the application and all the application’s S3 buckets and SQS queues are deleted.
Chapter 6 — Testing

Conclusion: From the above experiments, we conclude that the terminate running applications function works correctly.

6.1.5 Rebooting a running server

Objective: The objective of the tests carried out in this section is to verify that the framework application can properly reboot an EC2 server.

Protocol: To determine that the framework can properly reboot EC2 servers we develop a Python test script that determines and displays the status of an EC2 server.

1. Click the reboot an EC2 server link.

2. Verify that all the running applications and the DNS names of the EC2 servers running under the applications are listed.

3. Click on the DNS name of the EC2 server to be rebooted.

4. Verify that the server is rebooting, by querying the status of the server using the Python test script.

Result: On clicking the reboot a running server link, all the running applications and the DNS names of the EC2 servers running under the applications are listed. The EC2 server with the clicked DNS name is rebooted.

Conclusion: It is concluded that the reboot a running EC2 server function works correctly and properly.

6.2 The deployed application

The deployed application logic is distributed between the interface, slave and master servers. When the EC2 servers are instantiated, they automatically run a Python script get_user_data.py that gets application configuration data (user-data) sent by the framework application during deployment. In this section we first start by verifying that each server gets its application configuration data, before we test the functions of each server.

6.2.1 Getting the application configuration file

Objective: The objective of this experiment is to verify that all the instantiated servers properly get their configuration data from AWS, sent by the framework application dur-
Figure 6.6: Protocol followed when testing the get_user_data.py Python script.

The Python script get_user_data.py properly gets configuration data sent during instantiation and that they get their application files from S3. On each server, the get_user_data.py script does the following:

- Gets the configuration data and saves it in a Python script where the data is accessed by other application files.
- For the slave and interface servers, downloads the servers’ application zip files from the S3 buckets.
- Unzip the zip files into the the slave and interface EC2 servers.

Protocol: For testing purposes, firstly, the Python script get_user_data.py is not run automatically on boot up, but is run manually from an ssh session after the servers are instantiated. Functions listed above are incrementally added to the get_user_data.py script, testing each function before proceeding to the next. The interaction diagram in fig 6.6 shows the steps followed when testing the functions of this script.

Discussion: The Python script get_user_data.py properly gets configuration data sent during instantiation and saves the configuration data into a configuration file on each server. Also on the slave and interface servers the script can properly download the application zip files from the S3 application files bucket and unzip the files.
Conclusion: All the functions of the `get_user_data.py` Python script work correctly.

### 6.2.2 Interface server

The interface server hosts the developer presentation logic and a Python class (interface) in addition to the `get_user_data.py` described in section 6.2.1. The presentation logic uses the interface class operations to perform the following functions:

- Formats a request and sends it to an SQS queue.
- Polls the response queue for the response from the processed request.
- Transfers data to S3.
- Downloads data from S3.

**Objective:** The objective of the test performed in this section is to validate that the functions listed above are performed correctly.

**Protocol:** To test the interface server functions, an SSH session to the server is opened and a test Python script that contains code that instantiates an interface class object is developed and saved on the server. The instantiated object is used to send messages to and get messages from an SQS queue and send and download data from S3. The interaction diagram in fig 6.7 shows the steps followed to validate the correct working of these functions.

**Result:** We can send and receive messages to and from SQS queues and send and receive files and strings to and from S3 using the interface class operations.

**Conclusion:** We conclude that the interface server performs its functions properly.

### 6.2.3 Slave server

Together with a Python script for downloading server configuration data, the slave server hosts a Python script (`main.py`), Python class (slave) and the developer business application logic. The `main.py` Python script uses a slave class object to perform the various functions of the server. The slave class contains class operations that performs the following functions:

- Reads a message containing a user request from the request SQS queue.
- Retrieves the request parameters from the message and invokes the business logic of a developer application using the command retrieved from the message.
**Figure 6.7:** Protocol followed when testing the functions of the interface server.
Chapter 6 — Testing

- Receives responses from the business logic processing.
- Writes the responses to the responses SQS queue.

**Objective:** The objective of the experiments run in this section is to verify that the slave server functions execute correctly.

**Protocol:** We first test the functions of the slave class before testing the `main.py` script. To carry out the tests we develop a simple web application and deploy the business processing logic of the application on the slave server (`primes.py`). The web application generates the prime numbers of a given number. We develop a Python test script that test the various functions of the slave class as shown in 6.8. The test script sends a message that represents a request to be processed to the request queue. We then run the `main.py` script from the command line and verify that it can correctly read and process a request sent to the request queue using the slave class operations, before sending the processed response to a response queue. The processed response should be read from the response queue using a test script.

**Results:** The slave class operations to read requests from the request queue, process the requests and send the processed responses to the response queues work correctly. Also, the class operation that invokes the business logic script from inside its code works correctly. The `main.py` script correctly uses the slave class operations to perform the functions of the server.

**Conclusion:** We conclude that the slave server performs its functions properly and correctly.

### 6.2.4 Master server

**Objective:** The master server is responsible for automatically scaling the system. It does this by monitoring the average number of messages in the request SQS queue. The queue length gives an indication of how slave servers are coping with user requests. If the queue length is more than a certain preset threshold, over a certain period of time, then the master program logic launches more EC2 slave server instances to meet the user load.

The objective of the tests done in this section is to verify that the functions of this server are correctly executed. As with the other servers, the server contains the `get_user_data.py` Python script, a Python class (master) and a `main.py` Python script. The `main.py` script uses the various class operations of the master class to perform the functions of the server. The master class encapsulates operations responsible for:
Figure 6.8: Protocol followed when testing the functions of the slave server.
Monitoring the average number of messages in the request SQS queue.

Determining when to launch or terminate EC2 server instances.

Instantiating new EC2 server instances.

Getting the number of running EC2 server instances.

Getting the number of pending EC2 server instances.

Terminating running EC2 server instances.

**Protocol:** We first open an SSH session to the master server and create a test Python script that instantiate a master class object. The created class object is used to test various functions of the class by the invoking class operations. We change the code in this Python script to test the various functions of the master server. The diagram in fig 6.9 shows the basic steps followed when testing the functions of this server.

**Result:** The master class can correctly get the number of messages in SQS queues, instantiate new EC2 servers, determine number of running and pending EC2 servers and terminate EC2 servers.

### 6.2.5 Overall request processing system

To carry out the tests, we use the test web application used when testing the slave server functions. The web application’s presentation logic is separated from its business logic. The presentation logic of the program code fronts the application and is hosted by the interface server. The presentation program logic interfaces the web application to the users’ browsers. The business logic of the application is hosted by the slave server and it generates the prime factors of a number. The presentation logic provides a form where a user enters a number for prime factorisation. The number should be small enough such that the system does not take much time in processing the prime factors. This is because we are not yet testing the automatic scaling mechanism of the system that would enable the processing of large numbers.

**Objectives:** The objective of the experiments run in this section is to verify that the integrated interface and slave servers work together correctly and properly to serve user requests.
Figure 6.9: The interaction diagram for testing master server functions.
Protocol:

1. Test the business logic of the prime factors test application on a local machine and verify that it works correctly.

2. Deploy the prime factors application on AWS using the framework application.

3. Navigate to the prime factors web application on the interface server.

4. Submit a number small for prime factors generation.

5. Verify that the prime factors of the entered number are displayed.

Results:  When a number is entered on the web application’s interface, the prime factors of the number are correctly returned by the application.

Conclusion: The integrated interface and slave EC2 servers work correctly in serving user requests.

6.2.6 Automatic scaling of applications

Objective: The objective of the tests carried out in this section is to verify that the system is able to detect changes in user load and scale an application by accordingly instantiating more or terminating some slave servers. All instantiated slave servers should work together in serving user requests, thereby decreasing the user load. The system should be able to scale applications with different processing needs.

Protocol: To verify that deployed applications are automatically scaled effectively and properly, we develop two test applications with different processing needs. The first application calculates primes factors of a given number. The application’s logic is divided into presentation and business logic and deployed on the interface and slave servers respectively. This application’s business logic is processing intensive since it spends a lot of CPU cycles calculating the prime factors of a number, especially if the number is large.

We divide the calculation of the prime factors of the number into equal chunks from 2 up to the square root of the number. Each chunk consists of a range of numbers from which the application looks up for the prime factors of the number. We use trial and error to find an optimum size of each chunk. We start by using the application to calculate the prime factors of small numbers, moving on to bigger numbers. We note that for numbers less than a million, the application takes about 4 seconds to process requests, but for greater numbers, the response time jumps to 5 minutes gradually increasing until the system ceases to respond. From this observation, we estimate that each processing chunk
should be about a million in size. For example to find the prime factors of the number 10
000 000 000, only one processing chunk (1-100 000) is generated. We use the number 625
000 000 000 000 to test the application. This number produces 25 processing chunks from
2 to 25 000 000 (square root of 625 000 000 000 000). This means each request to calculate
the prime factors of the number 625 000 000 000 000 will produce 25 requests that are
sent to the request queue for processing on the slave servers. If there are 4 requests for
the same number, 100 requests will be send to the request queue.

The other test application is not as processing intensive as the prime factors generation
application. The application is a ticket booking system whereby users can book bus tickets
online. We only test one function of the application that accepts a reference number issued
to a user when booking a ticket. The function looks up the entered reference number in
the database and displays the details of the ticket. The database used for this application
is populated with about 1000 entries.

The ticket processing function is not processing intensive, since much of the work
involved revolves around looking up and retrieving data from a database and there is
little calculation involved. However, these type of applications can suffer from high peak
usage. For example a website (quite similar to our ticket application) selling Harry Potter
studio tour tickets crashed due to high traffic on its first day of inception and fans could
not book tickets [37]. As with the first application, the application logic is divided into
the presentation logic and business processing logic and deployed on the relevant servers.

We develop a Python script that sends a number of concurrent request to the applica-
tions. In the simulation script we start by sending concurrent requests to the application
at a lower rate for some time, before abruptly increasing the request rate. For the prime
factors generation application, we start with a request rate of 0.1 requests per second and
then increase the rate to 1 request per second before decreasing the request rate to 0.1
requests per second again. However for the ticket application, we start with a rate of 2
request per second, moving on to 4 requests per second and decreasing the request rate
to 2 requests per second again. The request rates are different for the two applications
because the ticket application takes less time to return responses, therefore the system
will start suffering from increased system load at higher request rates. The following steps
are followed when testing both applications:

1. Deploy the test application on AWS using the framework.

2. Run the Python script that simulates concurrent request to the application.

3. Plot graphs that indicate the response rates and number of running servers over the
time that the user load simulation script is run.
Chapter 6 — Testing

Figure 6.10: The graph showing results from tests run on the prime factors generation application.

Results: Fig 6.10 and fig 6.11 show the graphs plotted from the results from the tests that were run from the prime factors generation and ticket applications respectively.

Discussion: The graphs in fig 6.10 and fig 6.11 show that the system behaves as expected in both applications. For low request rates, the response time of the system is almost constant. As we increase the request rate, the system response time steeply rises, since it takes some time for new server instances to boot up and counter the increased load. From the graphs we see that as more processing slave servers are instantiated, the response time of the system decreases until it stabilises to a low constant again. When the system response time stabilises, the master server does not start up new slave servers, making sure that only enough slave servers are running. As the request rate is decreased, some slave servers are shut down so that there are no slave servers lying idle. The rate at which slave servers are shut down after decreased system load depends upon the cool down period specified in the master server. The cool down period is the time that should elapse after a decrease in load before the master server starts shutting down idle servers.
Figure 6.11: The graph showing results from tests run on the ticket application.

<table>
<thead>
<tr>
<th>Request Rate (req/s)</th>
<th>Recovery Time (minutes)</th>
<th>Worst Response Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>High</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 6.1: Table showing the recovery times of the ticket application under different request rates.
For both applications we choose a cool down period of 3 minutes. It takes about a minute for an instantiated EC2 server to boot up and a cool down period of 3 minutes allows all servers that are booting up to finish before the system starts to terminate some servers. This verifies that the master server is able to detect changes in the system load increasing or decreasing the number of processing slave servers, thereby automatically scaling the application.

The responsiveness of the prime factors calculation application is affected more than in the ticket application even though the request rates are higher in the ticket application. This is because the ticket processing application is less processing intensive and its response time is less than that of the prime factors calculation application.

From the graphs, we see that it takes some time for the automatic scaling system to improve the responsiveness of an application after the user load increases. The responsiveness recovery time should be as short as possible so that the performance of the system is not significantly affected over time. We measure this time for both applications and discover that this time is 6 and 4 minutes for the prime factors and ticket application respectively. These times are reasonable considering that it takes about a minute for an EC2 server to start up.

We also determine how the recovery is affected when the request rate jump from a low rate to a high rate is increased on the ticket application. Table 6.1 shows the results. From the table we see that if the request rate jump is high (2-10), it takes more time for the system’s response time to stabilise again. This behavior is expected since with a high request rate, there are more requests to be processed, thereby taking more time for the requests to be processed by the slave servers. Also the worst response time of the application jumps from about 30 seconds to 120 seconds. An application’s recovery time and worst response time can be improved by instantiating a number of slave servers at the same time in proportion to the responsiveness of the application at that point in time.

Conclusion: From the above observations we conclude that the system set up by the framework can correctly automatically scale deployed applications.
Chapter 7

Conclusion

7.1 Objectives

In this thesis we described some of the major characteristics of cloud computing and the advantages of this type of computing over statically hosted solutions. We motivated how cloud computing can be leveraged to produce highly scalable applications.

We also presented a comparative study of two of the major cloud computing platforms, Google App Engine and Amazon Web Services. Insights from this comparison were used to build a proof-of-concept framework for the provisioning of resources, deployment of applications on the provisioned resources and automatic scaling of deployed applications on the AWS platform. Amazon Web Services is chosen since there is generally more work involved with up-front infrastructure development before hosting an application than on the Google App Engine platforms. Consequently the introduction of a provisioning and scalability framework can help developers to more easily harness the power and flexibility and power of AWS.

7.2 Results

In this section we give a brief summary of the findings of the comparative study of the Amazon Web Services (AWS) and Google App Engine (GAE) platforms. We also give conclusions about the developed framework.

7.2.1 Comparative study of the Amazon Web Services and Google App Engine platforms

The comparative study undertaken revealed that the Google App Engine platform is domain specific, targeting the hosting of web applications on its highly scalable infrastructure. The level of abstraction on this platform is high such that developers have
limited access to the server operating system. Developer applications are not aware of
the underlying server operating system and cannot write to the server file system.

GAE has recently introduced support for MySQL databases for easy migration to the
cloud, but originally it did not support any persistent database system other than its
proprietary database system, BigTable. Developers do not need to manage storage-and-
compute resources, on the platform as the platform does all the required management.
The platform automatically scales all deployed applications and the applications do not
need to be aware of the scaling mechanisms.

On GAE, applications can only be developed using either Java or Python. Hosting
applications on this platform is comparatively less flexible, as developed applications
have to conform to the patterns allowable on the platform. A wide range of potential
applications are ruled out from being hosted on this platform because of these restrictions.
However, if one wants to quickly develop and have a web application up and running, then
the platform is an attractive option. Also the platform offers free resource quotas, reducing
the startup hosting costs of small web applications to zero.

Amazon Web Services offers several services that can be used individually or integrated
to provide computing resources. This is in contrast to the Google App Engine platform
where compute and storage resources are offered as an integrated package.

The Amazon Elastic Compute Cloud (EC2) service offers virtual servers in the form of
its instance servers. An Amazon EC2 instance server resembles a physical machine in that
developers have access to entire software stack, from the kernel through to the operating
system. The level of abstraction on the servers is very low as compared to the servers on
the Google App Engine platform, to the extent that users have root access to the server.
EC2 servers can be customised to suit developer needs, and different operating systems
and application software can be installed on the servers. A wide range of both web and
non web applications can be hosted on EC2. However, the flexibility that comes with this
service, means that users have to spend time doing upfront infrastructure development
before hosting applications.

This thesis addressed the need to manage and administrate the EC2 server resources.
The platform does not automatically scale applications, but developers can use some
of the services provided to implement an automatically scaling system. The platform
offers a queue messaging system called Simple Queue Service (SQS), that can be used for
communication between components in a distributed computing system.

There is a wide range of options in terms of data storage, ranging from relational
databases such as SQL and MySQL to storage in the form of other services offered on the
platform such as SimpleDB, S3 and the Elastic Block Store.
Chapter 7 — Conclusion

7.2.2 Framework

This thesis described the development of a successful proof-of-concept framework. The system set up by the framework uses the EC2 and SQS services to implement scalability. The development of the framework afforded us a chance to get a better understanding of the scalability concept and how AWS services can be used by developers.

From the scalability tests conducted we have shown that applications set up by the framework are able to scale with increasing or decreasing user loads. The scalability mechanism can scale both high and low processing applications. We have shown that the responsiveness recovery time of applications after increased user load depends upon the type of application deployed and the amount of load. Applications with higher processing take more time to improve in responsiveness after increased user load. Also, applications under high request rates take more time to improve in response rates than those with lower request rates.

7.3 Limitations and further work

The framework serves as a simple proof-of-concept with limited but useful functionality, but can be developed further to provide a fully comprehensive tool that can be used to administrate automatically scaling applications on Amazon Web Services. The framework can be improved in numerous ways but we describe only a few possible improvements.

The system set up by the framework can be improved in terms of fault tolerance. For the slave servers, fault tolerance is naturally achieved, since if one slave server fails, the master server simply instantiates another if the load is still high. However the architecture consists of single master and interface servers. Fault tolerance can be improved by providing redundant instances of these servers.

Starting up new EC2 server instances takes time and this impacts on the recovery time of an application under heavy load. Stopped EC2 servers take less time in starting up than new server instances. However only Elastic Block Store backed server instances can be stopped without being terminated. Presently, the framework system terminates slave EC2 servers when no longer needed, since the EC2 servers used are not Elastic Block Store backed. The recovery time of an application of the framework can be shortened by stopping slave servers instead of terminating them and when there is need for more slave servers at some time, the stopped slave servers can be restarted. The system can be improved to use Elastic Block Store backed slave servers and improve the scalability of the system.

The master server keeps on starting new server instances as long as the number of request messages in the request queue keeps on growing such that the developer defined request per running server limit is exceeded. However if there is an unforeseen problem
in the developer business logic on the slave servers such that request messages are not processed, the number of request messages in the request queue keeps on increasing and the master server keeps on instantiating more slave servers that add to the costs while the servers are just sitting idle. The system can be improved by limiting the maximum number of slave servers that can be instantiated per given time, such that if the described problem occurs, the wastage on idle slave servers is limited.

The framework only supports Python-based web applications, but can be developed further to support applications developed in other languages such as C++, Java and PHP. Functions that allow developers to SSH into instantiated EC2 servers using the framework can also be added. Presently the framework uses small-sized servers for the interface, master and slave servers. The framework can be improved by adding a function whereby developers can choose the type of EC2 servers to be used by their applications, such as small, large or extra large server types.

7.4 Conclusion

As the popularity of distributed cloud-hosted applications grows, developers will increasingly need to manage dynamic provisioning and scalability of computing resources. The framework described in this thesis demonstrates how such complexity can be abstracted in a useful way.
Bibliography


http://en.wikipeida.org/platform_as_a_service.