NIMBO CLOUD COMPUTING


A Thesis
Presented to the
Faculty of
San Diego State University


In Partial Fulfillment
of the Requirements for the Degree
Master of Science
in
Computer Science


by
Rajpal Paryani
Summer 2011
SAN DIEGO STATE UNIVERSITY

The Undersigned Faculty Committee Approves the

Thesis of Rajpal Paryani:

Nimbo Cloud Computing

__________________________
Joseph Lewis, Chair
Department of Computer Science

__________________________
Carl Eckberg
Department of Computer Science

__________________________
Michael O’Sullivan
Department of Mathematics and Statistics

6/16/11
Approval Date
DEDICATION

I would like to dedicate this thesis to my family, who has always been there to support me in every situation, irrespective of it being good or bad. Very special thanks to my father Dr. Dharampal Paryani who has always motivated me to progress ahead in life and always have a situation with plenty of options in front of you.
ABSTRACT OF THE THESIS

Nimbo Cloud Computing
by
Rajpal Paryani
Master of Science in Computer Science
San Diego State University, 2011

A software engineer working on a software project needs to have a development and test environment that is setup on systems that are “production like”. To setup a development and/or a test environment is a time consuming process. Developers use either their workstations or virtualization products to help in that effort. Any solid software project uses tools to help track issues, revise code via a revision control system and collaboratively work on the developer’s documentation. By using Amazon EC2 and expanding a bug tracking system it is conceivable to automate the process of creating the development and test environments on demand. This project will automate this process from bug report to development environment via click of a button. This push button enhancement to an existing issue tracking system will spawn an environment on demand. By reducing the environment setup time for each developer a company stands to save significant amounts of money. Additionally the ability to get systems on demand via outsourcing the IT services to a cloud provider such as Amazon’s EC2 saves a company IT setup and administration costs.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT ............................................. v</td>
</tr>
<tr>
<td>LIST OF FIGURES ..................................... vii</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS ................................... viii</td>
</tr>
</tbody>
</table>

## CHAPTER

1. **INTRODUCTION** ........................................... 1
2. **BACKGROUND AND LITERATURE** ........................ 7
   - Chef Overview ........................................... 7
   - Amazon EC2 Overview ................................... 11
3. **METHODOLOGY AND DESIGN** ............................ 19
4. **RESULTS** .................................................. 37
5. **CONCLUSION AND FUTURE RECOMMENDATIONS** ...... 44

## REFERENCES ........................................... 48

## APPENDIX

A. **USE CASES** ............................................. 49
B. **ACRONYMS** ............................................ 52
LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>Anatomy of chef run, from the chef opscode website</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2.2</td>
<td>Cloud computing diagram.</td>
<td>12</td>
</tr>
</tbody>
</table>
ACKNOWLEDGEMENTS

Throughout my life, I have been blessed to have wise people around me. They have come in form of parents, teachers, senior colleagues, friends and even people younger than me. Each one has taught me a valuable lesson and made me feel that there’s so much knowledge all around you, and you just need to open all your eyes and let all of that sink in. Even with friends who have tried to misguide me in some stages of my life, I have been blessed to come out stronger, learn from that experience and make sure that I don’t repeat that mistake again. To all those people in my life, I would like to say Thank you.
CHAPTER 1

INTRODUCTION

In software development industry setting up the right environment with respect to the project takes a lot of time of developers. It might take a day or couple of weeks or a month to set up a production like environment. My study involves implementing such a system which reduces this work of setting up an environment and gives an automated solution by enhancing a bug tracking system and spawning an image on just a click of button. A bug tracking system is a software application that is designed to help quality assurance and programmers keep track of reported software bugs in their work.

Many bug-tracking systems, such as those used by most open source software projects, allow users to enter bug reports directly. Other systems are used only internally in a company or organization doing software development. Typically bug tracking systems are integrated with other software project management applications. Having a bug tracking system is extremely valuable in software development, and they are used extensively by companies developing software products. Consistent use of a bug or issue tracking system is considered one of the “hallmarks of a good software team.

Most companies use JIRA [1] as their bug tracking system because of the 150+ features it provides. The JIRA UI is accessible from all the major browsers, for e.g. Internet Explorer, Firefox, Safari, and Chrome. You can also manage issues from different Integrated Development Environments (IDE’s) like Eclipse, IntelliJ, Visual Studio, Netbeans, JDeveloper, Zen Studio etc... by use of plugins for each of the environments. There are mobile applications available for iOS, Android, Windows and Blackberry OS so that people can access and update issues even from the most remote places. We can even define Email Notification schemes for project and personal levels so that each user can get the notifications depending upon their access to the system. The administrators also have access to define individuals to have “Watch Access” to different issues, so that any change to that issue would trigger an email notification to that user. Email subscriptions can be very useful to track even the most subtle changes in the JIRA system. RSS (Really Simple Syndication)

[1] JIRA is a trademark of Atlassian Software Technologies Pty Ltd.
becomes one of the most important features as the user can get the updates about the changes to the issues and JIRA system in real time using their favorite RSS reader, even without them logging into the JIRA web interface. One can even configure a different announcement banner for each page in the JIRA system so that useful information is displayed on top of every page. Each of these features is useful at different stages of the project.

In my project, I have used JIRA as a bug and issue tracking system for enhancement. JIRA provides issue tracking and project tracking for software development teams to improve code quality and the speed of development. JIRA makes bug tracking personalized and painless, so a team focuses energy on what matters most: great code. JIRA has different projects for different modules of the product, and each project can be further subdivided into a category of related issues.

This project will automate this process from bug report to development environment with a click of a button. This push button enhancement to an existing issue tracking system will spawn an environment on demand. To setup a development and/or a test environment is a time consuming process. The benefits of studying and implementing this product is that by reducing the environment setup time for each developer a company stands to save significant amounts of money. Additionally the ability to get systems on demand via outsourcing the IT services to a cloud provider such as Amazon’s EC2 saves a company IT setup and administration costs.

Cloud computing is one of the most important technologies of this time. The internet has evolved over the last few decades, and cloud computing is the technology for the current generation. It has the ability to provide the storage and computing power for proper functioning of an entire business at its different stages. There are different models of Cloud Computing: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Infrastructure as a Service (IaaS) layer gives different users the option to use storage and computing resources in their business solutions. Platform as a Service (PaaS) layer offers a kind of template service to the developers, so that they can use this template to build their own custom applications on an already existing infrastructure of computers. They can even provide developer tools which could be used to build data access, database or other billing services. Software as a Service (SaaS) layer, there is an external service provider that hosts the software on their server so that we don’t need to buy the hardware, install and
manage the software. All you have to do is connect and use it. SaaS Examples include
customer relationship management as a service.

Cloud Computing offers a lot of advantages. The cloud is elastic and resizable, which
means that we can shrink or grow the cloud and its associated services easily and within a
short period of time. This feature of elasticity enables scalability of the cloud, which means
that the cloud can be easily scaled upward in case of a rising demand and shrunk downwards
in case of the number of consumers using the system decrease in a certain period of time.
One thing that has to be planned in advance is the application should be programmed in such
a way that the resizing mentioned in the above steps does not affect the performance of the
application, and the change in size of the infrastructure is transparent to the users using that
application over a period of time. The users that want to use the services of the cloud have
the provision of getting the required hardware and software resources without a pre-requisite
of learning a lot of concepts about that Cloud system, so that they can allocate those required
resources within a short period of time. Once they are done with those resources, they should
have the option of deallocating those resources quickly, so that they do not get charged for
the time after which the resources are given back to the cloud provider. Cloud service
providers should have a standard set of APIs for using the cloud service through certain code
in the customer application, so the customer application and cloud service system can talk to
each other without any problems. Users of the cloud should be billed only for the amount of
time that they are using them. The pay-as-you-go model means usage is metered and avoids
any discrepancies between the customer and the cloud service provider.

Amazon EC2 is one of the most popular Infrastructure as a Service (IaaS) services.
EC2 gives the user an ability to rent servers and storage by the hour in a pay-as-you-go way.
It saves a lot of companies millions of dollars and is even fundamentally changing the way IT
organizations operate and support the business. Amazon EC2’s simple Web based
management interface allows users to view and control the computing capacity used with a
minimal amount of initial learning. It provides the customer complete control over their
computing resources and lets them program applications on Amazon’s excellent computing
environment. Amazon EC2 reduces the time required to obtain and boot new server instances
within a timeframe of couple of minutes, allowing the user to quickly upscale or downscale
capacity as their computing requirements change.
Amazon EC2 presents a true virtual computing environment, allowing us to use web service interfaces to launch instances with a variety of operating systems, load them with our custom application environment, manage network access permissions, and run a particular image using as many or few systems as desired. To use Amazon EC2, we simply select a pre-configured, template based image to get up and running immediately, or create an Amazon Machine Image (AMI) containing applications, libraries, data, and associated configuration settings. We can even configure security and network access on your Amazon EC2 instance. We can choose which instance type(s) and operating system the user wants, then start, terminate, and monitor as many instances of the AMI as needed, using the web service APIs or the variety of management tools provided. We can also determine whether we want the images to run in multiple locations, utilize static IP endpoints, or attach persistent block storage to each of those instances. And as mentioned above, we pay only for the resources that are actually consumed, like instance-hours or data transfer.

In addition to the Web interface, EC2 also provides a lot of Development tools to the user by which they can control all the above mentioned actions of the image as they are controlled by the Web based interfaces. We are using the Amazon API and AMI tools to control and generate the images (machines) respectively.

It takes a tremendous amount of time for developers to setup the environment to fix a bug in an application. Depending upon the complexity of application, it may take a few days to several weeks for a developer to replicate a production environment. Very few solutions exist today that help reduce that time. With the help of Amazon EC2 you can in theory set up instantaneous infrastructure through a bug tracking system (JIRA).

For each application, the developers usually need a machine for the old platform on which the bug exists, and a separate machine to install the new environment, which has the bug fix. In some situations, the QA (Quality Assurance) environment could need a separate machine for all the testing. If the total number of applications increases to say 40, then this leads to a maintenance nightmare, and the whole solution would be really expensive. The solution will consist of extending an issue tracking system to capture the configuration of a specific environment through a plugin and automatic installation and configuration of a system through templating of a system.
For automatic installation of softwares, we use an open source Ruby based system integration framework called Chef. Chef is built to bring the benefits of configuration management to an entire infrastructure of different kinds of machines with different operating systems. With Chef, the user can manage their entire fleet of servers by writing code, not by running commands, but by using a concept called Cookbooks. We can also integrate Chef tightly with our applications, databases, LDAP directories, and more via a concept called Libraries. We can also easily configure applications that require knowledge about the entire infrastructure by using the Search feature of the Chef framework. As chef is an open source project, it has a huge advantage over its other competing products because there are thousands of people working together to make chef a better product with its every iteration.

In this project, when the user clicks on the “Launch Nimbo” button on the issue screen, the Java based server program creates a machine on the Amazon EC2 service and then waits for a specified interval (120 seconds specified in the property file now). The newly created machine has a Java client program as one of the startup processes, so when the machine gets started, this program waits for an input on port 5000. The server program now sends the value of issue key and the username to the client program running at Port 5000. After the client program gets this input, it acts as a soap client and sends a soap query to the soap service running on the JIRA server machine to fetch all the Nimbo related attributes (role, SVN repo link) for the issue key received in the previous step. SOAP and the other acronyms are referenced in Appendix B.

The standard SOAP plugin provided by JIRA is limited in functionality such that it only provides a certain set of attributes associated with the issue to the soap client. In this project, we have provided many extra “Nimbo” specific attributes like Image Id, Role, SVN Repo Link and Apt Repo Link to a specific issue. To fetch all those attributes, we had to extend the standard RPC based SOAP library to add different functions to return all the above mentioned Nimbo Attributes. We made changes to existing SOAP library, added our functions to return those values and then republished SOAP library in the form of jar file that was put in the library directory of JIRA. The Soap client authenticates itself with the SOAP server using the value of the admin username and password of the JIRA system, and once the
authentication if successful, the soap server provides all the Nimbo specific attributes to the soap client.

The client program then sends the mail string message to the soap server program running on the JIRA server, so that server program can send a notification email to user that clicked the “Launch Nimbo” button in the first step. After this the client program uses the value of “svn repo link” attribute obtained in the previous step to check out the cookbooks associated with that role by using the svn client. Once the cookbooks are checked out in the home directory of the “ubuntu” user, the client program then calls the Chef Standalone client program “chef-solo” to install the softwares specified in the checked cookbooks from the previous step. The user now has the option of connecting to the newly generated machine by using the SSH protocol and start using this machine for his work of reproducing the bug mentioned on the issue screen. The above process ensures that the user has the ability to generate a lot of machines in a very short time and concentrate all his efforts in finding the solution to the bug in the issue screen.

In short, the preceding three paragraphs describe the primary research objectives of this thesis: configuration of machines using the Chef framework, the extension of the SOAP server functionality to support the system’s capabilities and finally the mechanism of communication, in the form of an email, with the developer. While none of these components, nor the numerous supporting pieces of software, have been described yet in enough detail to apprehend the extent, capability and complexity of the framework, the remainder of this thesis is devoted to provided both the necessary overview and the relevant details.
CHAPTER 2

BACKGROUND AND LITERATURE

CHEF OVERVIEW

Chef is a Ruby Based open source system integration framework which can help us automate actions on a huge repository of machines by using a specified set of programs. We can write a set of programs and then include those programs in the boot procedure of a skeleton image of machine. Whenever a machine gets instantiated from the skeleton, those programs would run and install a certain set of softwares required to give us different kind of machines.

Chef uses different components which need to be configured to make the chef skeleton using which the softwares can be installed. One of the most important concepts is the Chef Node. A Node is a host on which the Chef Client is installed. Its primary features are attributes and their run list. A node is the entity for which recipes and roles are made. Whenever we work with a machine which needs some things installed, that machine is referred as the Chef Node. Each Node has roles and its associated attributes. Attributes are structures of nested key-value pairs which describe the property and its associated value. Node and role attributes are commonly used as inputs for resource attributes. In case of a situation when you want different versions of software on different kind of systems (production, testing etc...) you can define role attributes to be different for those kinds of systems and hence have the same program run to produce different versions of the same software. The attributes are set in the attribute files which make sure that each attribute has a valid value (and not any jumbled incorrect value), this also makes sure that the errors are avoided when the program that reads these attribute value does not timeout or error out and gives a finished product.

Another important concept of Chef is the role. Using role, you can group similar features of simple nodes and define different roles, which would have one fixed part of configuration which is common amongst all nodes and the other variable parts in which
certain attributes which are different across different systems define the different roles that the system might have.

A cookbook is the collection of all the parts of the chef framework. It has recipes, resources definition, attributes, library, cookbook file and template files etc... All of these subsystems are explained subsequently. All of these subsystems with metadata of cookbook are used to install softwares and maintain the system (node). Different collections of all these subsystems with their attribute values decide the kind of cookbook that it would become and it also defines the software that would be installed on the system.

Recipes are the files where all the resources are written. Chef is run in a two step process: In the first step, called the compilation step, Chef evaluates the recipe files, building up a list of the resources. In the next step, Chef executes the desired action for each resource on the provider for that resource. Any arbitrary code in a recipe will be run during the compilation step, not the execution step. There are other steps by which we can defer the execution phase of a Chef run.

A particular piece of software might have some dependencies, all of which are mentioned in the metadata file of the Cookbooks. These dependencies could often be another cookbook, which would act as pre-requisite software, without which the current installation cannot proceed. Dependency tracking is the most visible part of metadata, but metadata also can contain information about authorship, licensing, a description of the cookbook, what platforms the cookbook is expected to work on, and whether or not a cookbook plays nice with other cookbooks. Maintaining accurate dependency information is absolutely essential, since nodes may not get all of the cookbooks they need if this information is incomplete, and this would lead to the failure of the whole installation procedure.

A Resource is a cross platform abstraction of the repository which would be ultimately used to install the software for that particular type of operating system. Each system supports a different kind of repository to install softwares, for e.g. Apt is the preferred repository for Debian based architectures and Yum repository is more preferred in Red-Hat based Linux distributions. Chef's resources are mostly just containers for data, with some basic validation functionality. Resources are declared in Recipes and Definitions. They are the basic unit of work in Chef. Resources can be further divided into Resource Attributes and Actions. Resource attributes are the pieces of data that resources contain. In the case of
managing a package, this might be the name of the package we want to install, the version to be installed, or options to pass to the package manager. Actions are usually the description of what we want Chef to do with the resource. We can give actions while the package is being installed, upgraded to the latest version or being removed. Actions are usually specific to the resource, but all resources support the nothing action, which does what its name suggests i.e. nothing.

The provider is the platform-specific implementation of the thing a resource abstracts. On Red Hat or Centos, a package resource will use the yum package provider to get the package installed, but on Debian and Ubuntu, the apt package provider will be used. Providers contain most of the intelligence: they're responsible for making Chef idempotent by checking if an action needs to be taken and issuing the commands to the system to take that action. In the case of package providers, they first check if the desired version of a package is installed and run the yum, apt-get, or other package manager commands to install or upgrade as needed. When working with Chef, we normally don't need to worry about providers. For those occasions, Chef provides "shortcut" resources which will always use the desired provider. The dpkg_package and rpm_package resources allow you to install packages directly from the filesystem, using providers specific to these package managers, for e.g. Providers take action on Resources. A given Node decides what Provider should be used for a Resource by default, or a Resource can specify a provider explicitly.

The Search functionality in Chef allows us to query arbitrary data about our infrastructure. The output of the search then decides the direction that our program would take. These searches are used mostly in which we use conditional expression check statements to control the direction of our program.

Figure 2.1 shows the diagrammatic representation of all the steps taken by Chef from the initiation stage to the final stage in which the software gets installed on the system -:

In our case , as we are directly using the chef-solo program, it does not require a Chef Server to run and give the Chef-Client all the node attributes, instead all those details are mentioned in the attribute files present in the cookbook, and the location of those files are given in the command which is used to invoke chef-solo.
Figure 2.1. Anatomy of chef run, from the chef opscode website.
AMAZON EC2 OVERVIEW

Cloud Computing is one of the most important cornerstones of the Computer industry today. The Cloud provides the power of Computing infrastructure, applications, business processes and personal collaboration to the user. Each of these can be provided as a service to the user at any time and magnitude as he needs it.

When the user requires a large amount of computing power to work on his task efficiently, he can either choose the computers with the same or different hardware and software characteristics and then combine their power to satisfy his requirement. But for getting that power, he will have to make significant initial investments in the form of server hardware and software programs and Operating systems, gather that investment should be combined with the cost of technical personnel to combine the power of different machines into one big entity. With time, they would also have to take the electricity and other bills into account, all of them when combined cause a big problem in terms of money required achieving his objective.

Cloud computing is actually a collection of services, with each service catering to a special need. Figure 2.2 shows all the cloud services.

Software as a Service (SaaS) is used to refer to the applications which are programmed in the cloud and can be accessed by different users at the same time. These applications use different kind of resources (Storage and different type of hardware) and combine all of them to make one single application which is very robust and usually gives high-availability due to its non-reliance on one piece of hardware, making it averse to a single point of failure.

One of the major players using the SaaS model is Salesforce.com. It is one of the leading products which offers different kind of services, but is best known for its Customer Relationship Management (CRM) applications. The CRM application is highly-available and lives in the cloud, so any user with an internet connected device like a cell phone or a laptop can access this product using a web browser. As this application if based in the cloud and does not live on a single machine and storage equipment, its downtime is really close to nothing, which makes it ideal to the enterprise kind of requirements.

A lot of SaaS applications are based on a multi-tenant architecture. This means that a single version of application is installed on multiple machines on the same level, this concept
is usually called horizontal scaling. A load balancer is then used which takes up all the incoming request, calculates the load on each machine in the architecture using an efficient algorithm, and then sends these requests to different machines in the same level providing the user with highly scalable and robust application experience. This is one of the examples of the various different kinds of architectures that the SaaS service provides. We can keep on adding horizontal and vertical layers of machines, and keep on programming our application to incorporate those changes and thus increasing the reliability of application.

Infrastructure as a Service (IaaS) refers to the different kind of machines which are provided as a service to the user. He/She is provided with options of choosing between different types of virtualized computers, each of which has different resources (in form of storage, memory, processing power etc). Depending upon the requirement of the user, he can then allocate a certain amount of resources and then get that machine at any time and with affordable cost.

Figure 2.2. Cloud computing diagram.
One of the most common analogies of IaaS can be defined in the area of public utilities where each user is not given the option to generate their electricity or arrange for their own water. Instead, the user is given the contact of agencies which do this on a global scale and then each user can lease it from those companies. In computing, this can be explained by the situation that each user does not allocate his personal computers, but instead contacts the big Infrastructure provider like Amazon so that they can allocate their own machines by making use of the Amazon EC2 service. They can run a single command to allocate the resources for a machine and generate an instance of a machine by using an Image Id. Once that command is run, they can use defend service like SSH, Remote Desktop, VNC to connect to those machines and start using them for their desired purposes.

Many people say that IaaS concept was brought in by Amazon when they launched their EC2 service way back in 2006. They provided a simple way of renting out really powerful machines, so that different individuals/companies can use these machines for a really low total cost of ownership and can begin their work as soon as possible rather than worry about the acquisition and maintenance (Storage area/electricity) of different machines. This idea is challenged by some people with the concept that even before Amazon started their EC2 operations, there were different ISP’s which gave out their machines for renting rather than ownership, the only difference was that most of these ISP’s gave out physical machines which were a little difficult to manage rather than the virtual machines that Amazon offered.

Amazon EC2 and other services related to it offer various advantages over the traditional buying of machines. The price of ownership of these machines can sometimes be significantly more than option of renting those machines. For E.g. -> If we consider an Ubuntu Linux machine that has a 2.4 GHz Quad Core Processor with 8GB of memory on Amazon EC2, the price of renting that machine (considering the east coast US zone) comes out to $3600 per year. This price is significantly less than a situation when an individual buys the machine for a particular amount (say $2500) and then combines it with different kind of expenses related to networking, Internet and IT services of people who provide the server management in form of software and hardware support.

In addition to the price, the speed at which different machines can be deployed is one of the major advantages enjoyed by the virtualized machine architecture companies like
Amazon. Once an image is created on Amazon EC2, the files associated with a bare bones architecture can be stored in a different Amazon Service like Amazon S3 (Amazon Simple Storage Service), and an Image ID can then be generated by using different commands. This single image ID can then be used to deploy multiple machines by using a single command, and each of those machines will be started in less than few minutes. The user can then configure different machines according to a load-balanced or any other type of architecture, and enjoy all the benefits of a highly scalable and highly available application which is then ultimately provided as a SaaS application which we discussed above.

Another advantage of IaaS service providers is that once the machine (or a set of machines) is deployed, the user does not need to look into day to day activities of monitoring the bandwidth that is accessible by these machines. All this will be done by the IaaS service provider. If the user still thinks this is a problem, he can call one single number for technical support for that service provider and can address all his complaints in one centralized place. Also the user is not concerned about the security of the machines by different thefts/ natural disasters, the IaaS service provider can arrange for all those requirements by making sure that the physical machines which run those virtual machines of a user are secure and resistant to different kind of security problems like thefts/ natural disasters.

As all the data of the virtual machines is actually present on the physical machines of the IaaS provider, the availability of that data is one of the major responsibilities of that provider. These IaaS providers have published Service Level Agreements (SLA’s) with their consumers which usually promise the availability percentage of the machine more than 99.9 percent (in case of Amazon EC2) for different characteristics like Memory, Processing power and bandwidth. Such kind of SLA level roughly translates to a downtime of the machines not exceeding seven minute in an entire year which is more than enough for a lot of enterprise level requirements.

Most of the users of the IaaS architectures machines are charged a usage based pricing, which means that they will only be charged for time that their machines are active in the IaaS service. This price is pretty low in most cases (10 Cents/ per hour in a particular zone in Amazon EC2 service). This can be really beneficial in a situation where a user just needs to deploy/test his application on a really powerful machine and then does not need that machine after his testing complete. If a physical machine was used for that testing, the user
would spend unnecessary money upfront, and then waste his money as the machine would no longer be used for anything else.

Amazon is one of the biggest players in the IaaS market today with thousands of users/companies using its services on a daily basis to satisfy their infrastructure requirements. Joyent is another really big companies which provides service to a lot of successful companies. Facebook and LinkedIn are two of the biggest companies which have a lot of Web applications on the Joyent virtual machines. The other smaller players when compared to the previous companies are GoGrid and FlexiScale, these companies provide different cloud services with different type of business architectures. Users have an option to choose any of the above companies depending upon the situation where a particular company is most suited to their requirements.

In addition to the public cloud services, there is a company called Eucalyptus which provides an open source alternative to Amazon EC2. It is based on the same Xen hypervisor that Amazon uses for its EC2 service. As a result, a lot of commands are similar in Amazon and Eucalyptus and a user which has used one system can start using another system with little or no training at all.

Platform as a Service (PaaS) is the combination of the machine resources (that IaaS provides) with a custom software stack which together gives us a platform for a particular software service that various users can use. PaaS providers provide the abstraction of the work layer in which a user/corporation needs to buy the physical hardware and the required software to begin the development of a product. The users of the PaaS service can concentrate on building the applications and programming their logic into the best and most efficient way possible. The PaaS providers provide all the facilities and tools to support the complete life cycle of a product manufacturing - from the design stage leading to development (programming of the actual application), QA/testing of the working and efficiency of the developed application, deployment stage i.e. hosting of application on their servers. Once all these stages are complete, they even support and provide the tools needed for the users to do various follow up activities to the application like database design change, collaboration tools like chat clients and web conferencing tools, for checking the scalability and state analysis of the application on a day to day basis. Tools are even provided for working with several versions of the code in the initial programming stage as well as
subsequent branches of the various modules of the project so that code can be branched properly and once the bugs reported are fixed, the branched code can be properly merged without any collisions and finally giving us the ability to release the different versions of the product. You can even go back in history to old versions of the code and find out if any modules need to be re-designed for the product to behave in a more efficient way. All the above mentioned functions are provided by using a web front-end or any other system for the various services so that the developers/testers of the application can access all the functions efficiently on different pages and can then collaborate to give out the best product possible.

There are different kinds of PaaS providers which may or may not provide all the functions mentioned above and give out different services to its users. Each users/Companies services might be different and the suite of services might depend upon the services availed by the companies which would ultimately decide the amount each company would be charged and the frequencies of the bills to that company.

The user of the PaaS can either use a strict set of programming language or tools (databases, servers and kind of operating systems) forming a closed system. PaaS providers also have the ability to use the Open Platform service in which the developer has the freedom to use their own set of programming languages and tools as long as the platform service provider can support all the different kind of systems and then integrate the code from the different systems to form different modules of the product which can then be integrated into a single application so that when the application user navigates from one screen to another and also from one module to another, the user does not notice any difference in the experience of using the application and all the things get integrated into a single seamless experience.

There are some PaaS providers which give us only the services of maintaining the security and the making sure that the application is always available for the use by the user. These providers are not concerned about the development/testing of the product and keep the onus of the efficiency or speed of the application on the developers who wrote the system. These providers increase the availability of the application either by using some resources like load balancers which could be either in hardware or software form. They also make sure that the application is secure by closing out all the possible points of entries rather than the authentic ports on which the application runs. They might even provide the facility of doing
some kind of penetration testing for the application initially for the company which has designed the application and provide them the output in form of detailed reports of all the vulnerabilities faces by the application and suggestions of how to fix those vulnerabilities. Regular audits can be followed in certain intervals of time (monthly/yearly etc.) which make sure that the application is safeguarded by all the latest attacking mechanisms being developed. All these mechanisms are to make sure that the application in question is not susceptible to various advancements in the areas of breaking the code by intentional or unintentional ways.

There are various stand Alone development PaaS environments which are not concerned with the licensing dependencies on the application in question. Technical resources or tools for the application are not provided and the development of the application is solely dependent upon its designers, programmers and testers. All these providers do is to provide various collaboration services along with a general development environment which the developers use to make a useful and efficient application.

Some vendors of the PaaS also allow existing PaaS based applications to be customized according to some special requirements. Some additional add-ons can be added which extend the existing functionalities by providing some extra interfaces. For e.g. -> A macro facility which was added in the subsequent versions of excel were added by just adding some features to the old code version base.

In addition to providing the development environment to the developers for development of applications and providing hardware and software resources, the PaaS vendors also provide additional web service extensions like Simple Object Access Protocol (SOAP)[2] and Representational State Transfer (REST)[3] and other RPC related access methods which the developers can use to access different resources. When all of these communication methods are combined with database access and other service access, it creates a set of Webservices called Mashups. The access to each of these web interfaces can be restricted by the network level, by allowing connections either in the private Local area network or by allowing internet connections from outside. If the outside access if allowed, the vendors can the application companies can still restrict the access by allowing internet connections only from a certain set of IPS. All these security procedures are essential for
enterprise level security and making sure that the bad guys are not allowed to access the applications.

PaaS also provides the creation of interfaces by using different internet technologies like HTML and JavaScript. Using these technologies, efficient designs can be conceptualized in form of web based interfaces that the users can access and interact with the application. As all the designs are web based, a lot of users can access this interface concurrently and the application developers can extend the application by making sure that the application is averse to failover of machines. If one of the machines hosting the web based interface goes down, there would be some other machine which automatically takes the old machines place. All this substitution should be transparent to the user and he should not experience any disruption of the application that he is interacting with.

Despite all the advantages of the PaaS mentioned above, it also suffers from some disadvantages. One of the major disadvantages is the lack of interoperability between different PaaS vendors. As mentioned before, each of those PaaS vendors can be different; they can vary from the kind of support for different programming languages, and the different tools it gives for collaboration between the developers/tests of the application. If the company which made the application using one PaaS provider tries to switch to some other cloud provider, they might not be able to move their application code from one cloud to another as the other cloud provider might not support the old programming languages and its development environment. In some cases, a switch might be possible, but could be really expensive for the company to migrate all the features of the application. In addition, if the company that owns the application has to maintain a transparent switch of the PaaS vendors, it would have to synchronize all the data in real time between the two PaaS vendors, which in some situation might be impossible, so the planning of the move has to be really carefully planned.
CHAPTER 3

METHODOLOGY AND DESIGN

The motivation of our project stemmed from the need to get production ready systems for the developers who are working on certain issues (bugs) of a project, so that they do not waste any time in getting all the components required to start working on an issue. With this system we automate all these actions by integrating functionality with a popular bug tracking system (JIRA), so that a developer can get a machine with all the softwares installed by clicking a single button. The requirements for all the actions done after clicking the “Launch Nimbo” button are mentioned in Appendix A.

Once that button is clicked on the issue screen of the JIRA bug tracking system, the JIRA Server calls a Java based server wrapper program which instantiates an image on a cloud based virtual platform, which in our case is Amazon EC2. The image started has a pre-configured skeleton created by using the requirements defined by an administrator of the company. The administrator has to give out basic requirements such as the OS required on the image, any special softwares that are specific to their business and would be required on every machine that their company requires to be automatically instantiated. During this stage, a NIMBO representative gathers all the requirements from the administrator of the client company so that he can create the bare bones skeleton of all those machines on Amazon EC2.

The creation of the skeleton is a complex process that involves understanding core concepts of Amazon Web Services[4] such as Amazon EC2 Virtualization Cloud Platform, Amazon S3 (Amazon Simple Storage Service) buckets in which the skeleton the image is stored. An Amazon bundle is a collection of files of the image with the manifest file defining each file required to ultimately create the image. The manifest file of the skeleton machine has all the details about the ancestor Amazon Machine Identifiers (AMI) images through which the skeleton image is created, the list of encrypted files in the S3 bucket which when combined together would make one single skeleton image, and the signature of the user that created the image stored in an encrypted format.
For the creation process, we have to use the Amazon API and AMI developer tools which give us commands required at various steps of the image creation process. The API tools are even used to control various aspects of the images. We can use “ec2-create-instances” command to create various instances of images by using a single AMI Id. The “ec2-describe-instances” command is used to display all the details of the images like hostname, instance id, zone etc … created by a single account. The manifest file in the S3 bucket is used by the “ec2-register” command to issue an AMI id. All these developer tools become really useful when all the image manipulation is to be done from the programs that can integrate these command line programs to do a certain set of actions. Amazon even provides a Web based interface called AWS Management Console to control all the above mentioned Amazon Services and any user with valid credentials can login and control various properties of all the services.

During the Skeleton image creation process, we first take a sample image which would be customized to create the final skeleton used to create multiple instances of the image. In our case, we took the sample image 10.04 Lucid Lynx Ubuntu image that the Ubuntu company Cannonical made for Amazon EC2 service. This image is a special image and runs a special kernel based on the Xen hypervisor used by all the Amazon Machines on the EC2 service. Once we start up that image by using our EC2 key and “ec2-create-instances”, we can connect to that image by using the hostname value and the Ssh client. On that sample image, the NIMBO programmer also puts a special Java based Client Program in the startup programs of that OS, so that this client program calls the server to fetch all the information required to decide various parameters to be set in the client which would ultimately decide the kind of softwares to be automatically installed on the client machine. Once the client program is put and the image is customized to suit the needs of the role defined by the administrator, the programmer also includes different kind of client programs required at various stages of the deployment of the machine. Each of these client programs has a specific purpose and is used at different stages of the installation. For e.g. -> “SVN” client is used to check out different files required for installation of softwares on the machine. Java Client programs are used to run the client java wrapper program as well as satisfy the needs of other java programs used at different stages of the installation process. The Java program is the daemon which listens at port 5000 for the client and is used for
communication with the server program. In the installation of softwares, we are using an open source ruby based framework called CHEF, so all the ruby based pre-requisites required for chef need to be installed on the skeleton machine. This is done by first installing the enterprise edition of Ruby by downloading the package from the Phusion Website, and then installing other dependencies of the Chef framework like ruby-developer framework, Open SSL libraries of ruby (libopenssl-ruby), Document Generator for Ruby Source (rdoc), Ruby Index Database (ri), Interactive Ruby Shell (irb), Build Essential Package that lists/install a lot of useful programs for building and compiling ruby based script and programs, GNU wget package which is used to download of files from the Web using HTTP, HTTPS, FTP etc protocols, SSL Certificate validation Package (ssl-cert). All the above mentioned dependencies in our case were installed by using the default package manager for Debian based architectures called APT. Our installation was done on an Ubuntu machine, but the same concept can be applied to all the Unix machines based on the Debian Architecture. After the dependencies are installed, we then install the rubygems package by using the repository. At the end we then build the chef based gems (chef, ohai) by using rubygems package and all the above mentioned dependencies. The gems give us all the chef programs in our skeleton machine which would use the cookbooks checked out by the client program and then installing the programs by using the chef framework. After the Nimbo based softwares are installed, we need to transfer the private key and certificate associated with our Amazon EC2 on that instance, which would be helpful in the bundling of the image to the custom state we want.

Once all the softwares required for the basic image are installed, we need to setup the Amazon AMI tools on this sample machine. For the setup, we need to extract the AMI and API tools at a particular place in the file system, preferably in a hidden directory under the /mnt directory. This makes sure that the Amazon API tools do not get bundled in the new image created. After the extraction of those AMI and API tools, we need to setup environment variables which would give the location of the private key and the certificates copied in the previous step, also setup the location of EC2_HOME and PATH variables to the directories in which the AMI tools are extracted.
We now use the following command to convert this image into a bundle -:

`ec2-bundle-vol -d <directory containing the bundles AMI parts> -k <private key associated with client’s Amazon account> -c <private key associated with client’s Amazon account> -u <Client’s Amazon Web Services Account Number> -r <architecture of the machine, either 32 or 64 bit> -p <name of the image which would be uploaded to bundle>

For e.g. -> ec2-bundle-vol -d /mnt -k /mnt/pk-HKZKYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem -c /mnt/cert-HKZKYKTAIG2ECMXYIBH3HXV4ZBZQ55CLO.pem -u 495219933132 -r i386 -p sampleimage

The above mentioned command puts the different files of a bundle under the directory mentioned after the “-d” switch, which in the above mentioned example is “/mnt”. The “-p” and “-c” switches are used to point the location of the private keys and certificate files associated with the client’s Amazon EC2 account. The “-u” option specifies the unique client User ID provided for the Amazon AWS services. We also give the architecture information, which is 32 bit in the above case (i386) by using the “-r” switch and the name of the manifest and other files by using the “-p” option.

Once we have the bundle inside the “/mnt” directory, we need to upload the same bundle to the Amazon S3 Storage. So, we first need to make a bucket on the S3 service by using the web based AWS Management Console, and then use the following command to upload the bundle from the local “/mnt” directory to the S3 bucket -:

`ec2-upload-bundle -b <your-s3-bucket> -m /mnt/image.manifest.xml -a <aws-access-key-id> -s <aws-secret-access-key>

For e.g. -> ec2-upload-bundle -b nimbobucket -m /mnt/nimboimage.manifest.xml -a 111sssaassd23 -s 232323sdhhsa321`
Once the bundle is unloaded to the Amazon S3 bucket successfully, we have to change the EC2_HOME environment variable to point the location of the Amazon API tools and then use the following command to register the bucket files to a corresponding AMI -:

```
ec2-register <your-s3-bucket>/<Manifest file name>
```

For E.g. -> ec2-register nimbobucket/image.manifest.xml

The above mentioned command assigns AMI's (Amazon Machine Identifiers) to that role. This AMI is then mapped to the role in the Nimbo system, so that the same skeleton is used to generate several machines with same configuration. Each of those machines would be created by the Server side Java wrapper program running on the same machine as the JIRA server.

In our project, we can create machines with different types of Operating Systems, but to finish this project within a specified time limit, we have currently added support to only generate machines that have their primary OS as Ubuntu. Ubuntu is a free Linux based Operating system based on the Debian Based Architecture and has the widest support for the different kind of hardware on which the Operating system can be installed. It is one of the best choices when one needs to install the Linux based operating system on a particular hardware configuration. Its robustness, lightweight architecture make it a really good candidate for installations on even the slowest of machines, for e.g. - A machine with a single core 1 GHZ Intel Processor that has just 256 MB of memory is good enough for installing the latest version of Ubuntu OS (10.04 Lucid Lynx). We can even install the server version of Ubuntu 10.04 on really powerful machines that have 8 cores of processors and more than 100GB of RAM, and these machines can be used to service a lot of users that connect to it simultaneously. Our criteria of choosing Ubuntu for our project were its easy availability of all the major versions of the operating system on the Amazon EC2 platform. There is a project from Euclyptus[5] called Ubuntu Enterprise Cloud (UEC) which makes the images of Ubuntu that are customized to run on Amazon EC2 and similar architectures. Eucalyptus is also based on the same architecture as Amazon Web Service and uses almost the same API provided by Amazon to do different actions with cloud based images. In addition
to that, Ubuntu uses one of the best package managers available for the Linux platform called APT (Advanced Packaging Tool) which has a vast collection of softwares which can help the operating system perform a different variety of roles, for e.g. - A fully fledged enterprise web server with Apache installed that can service millions of users simultaneously, a database server with enterprise version of mysql server installed etc..

All the above mentioned roles are automatically installed with the help of Chef framework that converts the ruby based cookbooks into the simple apt based commands that can install different kind of softwares. The open source nature of both the Chef and APT projects ensures that a large variety of softwares are available for this operating system, and the bugs in one version of softwares can solved by a large amount of developers that contribute to the availability of the open source projects. The open source nature of these projects and its associated nature of all the softwares also makes it possible for different companies to customize the softwares according to their individual needs.

In our project, we have developed a Client side Java Wrapper Program that are included with the other normal startup scripts of that OS. We put our Java Program and its properties file in the “/usr/local/Nimbo” directory, and the startup script under the “/etc/init.d” directory. In UNIX, we have different runlevel in which the operating system can boot, these runlevel would decide the kind of programs that could be started in that runlevel, and with some programs in the stopped state, and we would get a system with different states. In Ubuntu and other flavors of Linux, the runlevels are usually used to distinguish between a GUI and a command line environment. The GUI level can be used for displaying a desktop style OS, in which the user can perform all the functions by using different kind of GUI tools instead of using the hard way of using the command line to change different kind of settings. The GUI tools can also make navigating and accessing different files easier and are really helpful to people who don’t want just want a simple system for their daily tasks like browsing the internet, listening to music etc. The advances users can boot in another runlevel, which would be all command line based, and could have various server kind of services started in that runlevel. For E.g. - An advanced user such as an administrator of the operating system and a specific service can use the command line to change specific properties of that service, in case if an apache web server, the administrator can use a text based editor like VIM to edit various configuration files of apache, and also
restart services from the command line to affect the functioning of the Web server. In case of Amazon EC2 and our project, all the images are command line based, so the user can use the command line interface to navigate to the different parts of the filesystem and do his actions on the machine.

We have put our startup script “NimboClient” in the same startup directory “etc/init.d”, and then used the “update-rc.d NimboClient defaults” command to make this startup script run in all the runlevel of Ubuntu, as a result, this script would run on all the configuration of the system and is available to all the users. This startup script is loaded on a new machine on the Amazon EC2 platform and listens on port 5000 for all the incoming requests from the JIRA server.

The main program in the Ubuntu client is the StartClient.Java. This program uses the “Properties” class in java to load and save the properties related to different attributes from a file, which is “clientSettings.properties”. This file also stores the values of attributes that are received from the JIRA server in the same file. When the “Launch Nimbo” button is clicked from the JIRA server, the server side wrapper program starts the image on Amazon EC2 using a particular image id which is picked up from the Project properties. Once the command to start the image is given, the server side program waits for some specific time (120 seconds in our case, but can be changed by changing the server properties file), and then queries the Amazon EC2 service for the hostname of the newly generated machine. Once it has the value of that hostname, it sends the values if issue key and the user name value that initiated the action to the client script running on port 5000 of that hostname. The client program which was in passive mode until now i.e. just listening on port 5000, it logs those properties in the “clientSettings.properties” file and then starts it’s processing. The properties file is also used in case there is any problem in the communication between the server and client, and the correct values do not reach the client side. The client then picks up these default values and does the configuration using these values. This program also has a Logger class which is user to log all the actions of the client script. Once the Client script gets the properties of the issue key and username in the above case, it logs those values in the log file (JiraClientLog.txt), which in our case is located in “/tmp”.

When the Client Program receives the values of issue key and username, it then sends a soap query to the JIRA server which has a fixed address. In our case, the JIRA server runs
on the hostname “raj.nimbo.us” and port 8080. For the soap query to be sent from the client script, we have to enable the soap related libraries in our project. We have extended the existing SOAP library which is originally based on the Apache AXIS libraries; it is provided with the standard JIRA distribution. We have created a extension to that RPC SOAP library [6] so that it is able to give all the details associated with our Nimbo Project.

The standard SOAP library is limited in the functionality; it only provides a certain set of attributes associated with an issue. For providing all the Nimbo specific attributes, we have to extend the standard RPC SOAP library of JIRA. We insert additional functions that establish an ODBC connection to JIRA database and provide all the NIMBO specific attributes to the soapclient. The soapclient authenticates its connection with the soap server by using the username and password of the user that has admin access to the JIRA bug tracking system and can query the issues to return specific attributes associated with those. The requirements for the soap service extension are given in Appendix A.

Once the developer clicks the “Launch Nimbo” button on the issue screen, the server program creates the machine for that role (mapped to the project that developer is working on) on Amazon EC2 by using the AMI created by the NIMBO programmer initially. While the machine is being started, the client program on the machine creates a process that listens on port 5000 on the client. The server and client program are interactive programs that send information between each other to decide various parameters and proceed with the other actions depending upon the information passed. After the machine is started and a hostname is assigned to that machine, the server program first sends the “issue key” information to the client on port 5000. The client program acts like a SOAP client and sends the SOAP requests to the server using the issue key to find out the information about the issue on which the “Launch Nimbo” button was clicked. A SOAP server running on the server side is used to provide all the “Nimbo” based information like role, svn repo link, apt-repo link etc .. to the client program. Once the client has all this information, it determines the role of the machine by using the “svn repo link” information received and checks out all the cookbooks associated to the role by using the svn client already installed in the bare bones skeleton of the image.

A machine that gets started by the server side wrapper program is template kind of machine, and acts as a starting point for the final machine that is required. In every case, the
role of the machine decides the kind of softwares that would be installed on the machine. Currently, we support three kinds of roles, the first of being a plain Apache Web server which has the support of all the build in standard modules. Each of these modules is used for a specific purpose, “mod_auth_basic” supports the basic authentication feature of apache, “core” refers to the core http server features that have to available at all times, “prefork” module implements a non-threaded, pre-forking web server architecture etc.

The second kind of role is PHP server which is an extension of the above mentioned apache web server, so that it has specialized php modules installed. These modules give it the capability of handling server side php pages which act as a calculator of web pages, each user is displayed a different web page which is decided according to the values entered by the user on the previous page and also its personal preferences. The Third kind of role is of a database server that uses the enterprise version of mysql installed on it. It uses some general values for the database, the user and password values that can be agreed between the administrator and the Nimbo programmer so that the programmer can design those values depending upon the personal preferences of the administrator from the client side.

Each of those roles have cookbooks that are placed on specific places on the svn storage area and can be checked out by using a single command that is obtained by the “svn link repo” from the Nimbo related attributes of the project stored on the JIRA server. Once the cookbooks are checked out, the Chef framework takes over. We use a part of Chef framework called “chef-solo”, this is a standalone component of the framework that does not need a chef server to process instance related information and install softwares. Instead we give all this input in two files, the first one if the config file that has two variables – the “file_cache_path” and “cookbook_path” that give the location of the file cache directory and the cookbook directory respectively to be used by chef-solo. The paths could be at any place which is accessible by the user running the chef-solo command, but the important thing in this case is that the path should have absolute path values and the “cookbook_path” should be a subdirectory of the “file_cache” path. This command is usually run by using root privileges, either run by root directory or by using a “sudo” command, which translates the local user permissions to be elevated to the root level as long as the user is in the wheel group and also the part of the sudo file and is allowed to run these kind of commands.
The second file is the json file which gives the attributes and their values associated with the cookbooks that are supposed to be installed to the server.

Here’s the example of “solo.rb” from the mysql cookbook :-

```ruby
file_cache_path  "/tmp/chef-solo"
cookbook_path  "/home/ubuntu/mysqlchef/cookbooks/"
log_level  :info
log_location  STDOUT
ssl_verify_mode  :verify_none

In the above file, the cache files of chef would be stored in the “/tmp/chef-solo” directory. The path "/home/ubuntu/mysqlchef/cookbooks/" would be searched for cookbooks and all its associated file in its subdirectories too. The log_level module would determine the amount of log detail that would be give out in case the log switch “-l” is used.

log_location gives out the location where the log would be dumped out, it can be either the screen (STDOUT) or it can be also redirected to any specific file on the filesystem.

“ssl_verify_mode” gives the capability to verify the client side ssl certificate through which the cookbook is called, which in this case is set to none because we don’t check for authenticity of the client that called the cookbook.

An Example of the “dna.json” file from the mysql cookbook :-

```json
{
  "mysql": {
    "server_root_password": "root",
    "bind_address": "127.0.0.1"
  },
  "ruby_enterprise": {
    "version": "1.8.7-2010.01",
    "install_path": "/usr/local",
    "ruby_bin": "/usr/local/bin/ruby",
    "gems_dir": "/usr/local/lib/ruby/gems/1.8",
```
}
"url": "http://rubyforge.org/frs/download.php/68719/ruby-enterprise-1.8.7-2010.01.tar.gz",

"passenger_enterprise": {
    "version": "2.2.9",
    "root_path": "/usr/local/lib/ruby/gems/1.8/gems/passenger-2.2.9"
}

"gems": [
    { "name": "rake" },
    { "name": "rails" },
    { "name": "bundler" },
    { "name": "paperclip" },
    { "name": "chronic" },
    { "name": "coderay" },
    { "name": "mysql" },
    { "name": "newrelic_rpm" },
    { "name": "rack" },
    { "name": "rack-cache" },
    { "name": "rakismet" },
    { "name": "RedCloth" },
    { "name": "ruby-openid" },
    { "name": "will_paginate" },
    { "name": "cucumber" },
    { "name": "rspec" },
    { "name": "rspec-rails" }
],

"apps": [
    {
        "name": "rajapp",
        "username": "raj",
        "git_branch": "akitaonrails",
        "server": "ubuntu.local"
The above is separated into various parts. The file part gives the root password of mysql, the binding address. We can even mention other users that use mysql server along with the their MD5 encrypted password and the comment which describes the information like the User’s Name etc ... We have to even give out the details of the gems that would be
used for the installation of mysql on the current server. All the recipes that would be used by
the cookbooks are to be mentioned in the order in which they would be used while the
installation. In addition to that, we give other generic attributes like ruby version, the
recipes in the cookbook, the run list (which decides the order and the conditional execution
of the recipes, all these attributes are generic to all the cookbooks and can be added/deleted
according the requirements of that particular type of role.

The following command is used by chef-solo for installation of softwares -:

chef-solo -c ~/solo.rb -j ~/node.json –l debug

Once the above command is run, it picks up the run_list from the json file and checks
for the version of the operating system on which the command is run (which is Ubuntu 10.04
in our case) by using the “ohai” rubygems. It then loads the chef framework related attributes
from the cookbook files under the “definitions” subdirectories. Usually the attributes are
given by to the system by using knife by the chef server, but as we are using the “chef-solo”
standalone module, all the information has to be provided in the cookbooks. It combines
those attributes and then searches for the recipes to be run. Chef can either directly picks up
the recipes from the json file, or you can even specify the main recipe name mentioned in the
cookbook, which in turn calls all the other recipes. In case of apache, we have mentioned the
main chef recipe called “apache2” which in turn calls all the other recipes like “mod_status”,
“mod_alias”, “mod_authn”, “mod_authz” etc. All these recipes are used to install the
different modules supported by the apache webserver and are to be installed on this server.
Once all the recipes to be used are loaded by chef, it then searches for corresponding
cookbooks associated with those recipes. Before the installation begins, it checks for the
current version of the software installed on the machine to determine if the install is just an
upgrade install or a fresh install, and accordingly call the appropriate the apt command. As
this is a first install of a fresh image (server), it determines that the current version is “nill”
and it then determines the later version available on the apt repo. Once that is determined, it
then installs the apache software by using the command which would be otherwise used by
the user to install apache on the Ubuntu machine by apt. It uses the command “apt-get –q –y
install apache2=2.2.14-5ubuntu8.2” which would install apache in the quiet mode and also
answer yes to all the questions asked by this command to the user. These questions usually are the confirmation that a certain amount of disk space would be used by the install command and also if some changes to server for the startup is acceptable. As the installation procedure is done without the involvement of the user, we usually answer yes to all the questions. The apt command then fetches all the packages from their apt repositories and then installs every package sequentially. Once all the packages are installed, the chef framework checks to see if the apt command used to install the packages ran successfully and without any errors by checking the return value and making sure it is 0, which indicates a successfully execution. After the confirmation, chef then starts all the services associated with the installed software and makes sure all them install successfully by again checking the return values of run of all commands used to start, stop, and status of the service.

Once all the services are checked, the chef then creates all the necessary configuration files required for the normal functioning of that software. In this case, it puts the apache configuration file, “apache2.conf” in “/etc/apache2”. It then puts the file used by all the modules of apache in their respective path and checks that the functioning of the of each of these modules by giving it a sample input and then looking at the output of that module and making sure it is the same as expected. All the files put in different path above by chef are already present in the cookbook path as templates, some things are changed in each file depending upon the platform of the operating system, and then put into their respective places. To be extra sure, chef even calculates of the original template and the new file and compares the checksum to make sure that the files were not changed by some malicious user/process. The chef framework even sets the permissions of different directories by changing the owner as the default user under which the service will run, this rules out the possibility of compromising the security of the entire system in case a bug in that service is exploited by a malicious user. Once all the steps are complete, the chef framework then starts the associated services and that software is ready to be used by the user.

All the above mentioned error checking mechanisms makes chef a very robust and enterprise class framework that be used to manage a really large amount of systems and can scale the infrastructure of an enterprise from a small install to a large fleet of server within a very short span of time using the automatic installation, and ruling out any human error which is found so normally in installation of large service like a web server on big servers.
Before the cookbook checkout section begins, we send out a mail to the user that clicked the “Launch Nimbo” button, as the machine is already running. For doing this, we first compose all the content that is supposed to be mailed to the user, that includes the hostname of the generated image, and a lot of other client specific details. These details are first obtained by running various commands using the exec() system call provided by Java, and then using this output to make the final value of the mail content. Once this is done, we call the soap server’s sendMail() function to actually send out the mail.

The Client side program also includes a Logger program “LoggerUtils.java” which logs every action that is being performed by the client in a file under “/tmp” called the “JiraClientLog.txt”. It starts by first trying to start the client program on port 5000, if for some reason the client program cannot be started; it logs an error and exits the program. While we are getting the key and the issue id from the server, it logs that information into the file. The mail sending part with the username value is recorded by the log file. The checking out of cookbooks, the installation of the software using the chef framework and the entire output of chef is even recorded the logger file. The logger program makes sure that in case something does not happen according to the above mentioned sequence, we can always the log file to go back and find out the current state of the machine. It narrows down the scope of the error so that we can go back to the machine and perform the actions which would make sure that the problem does not happen again.

On the Jira server side, we get a stock SOAP server based on the RPC protocol (based on the Apache AXIS service architecture) which handles all the soap requests. Before using the SOAP server, we have to make sure that we change the settings to “Allow remote API Calls” and also enable the soap server module which are disabled by default in the new JIRA server install. Once this is done, we have to first fetch the Wsdl[7] file of the soap server on the JIRA server side and store it in one of the local files on the client side. This wsdl file[8] gives the soapclient information about the different paths that have to be queried for the soap server to take up requests from the soap client and then give it relevant outputs. We use this wsdl file along with value of the username, password which has required permissions to query the projects on the JIRA server side.. We also have to supply the identifier which would give us information about the issue. This identifier can be either the direct issue key, or project or the artifacts and version Id’s associated with the project. The SOAP server then
can give out various kinds of information about the issues associated with a certain project to which the user has access. We can use this SOAP server for getting some information about the issue by using the issue key, but as we have made another table in the JIRA server, which has information about the NIMBO aspects of the issue like role name, image id, svn-link repo, apt-user repo etc., we cannot use the standard SOAP server to satisfy all those queries. As a result, we had to make our own SOAP server that can take the input in form of an issue key and give us all the NIMBO related information. One of the best features of JIRA in this situation is that we do not have to start from the scratch and can instead use the “RPC Endpoint Plugin Module” in the JIRA server. Once that is activated, we just have to program the additional functionality that is required by our project and then place our JAR file in the JIRA libraries directory. As JIRA is open source, we can even download the code used by the existing SOAP server which acts as a very good starting point for writing our own soap server. We can use the existing classes used in the stock soap server and then write our own classes to extend the functionality of the SOAP server.

For our project we have a class called “SafirJiraSoapService” that extends the existing SOAP server class “JiraSoapService”. In this class, we have three functions: getProjectID, getNimboDetails, and sendMail. Each of these functions is implemented by using an interface called SafirJiraSoapServiceImpl (which extends the standard interface JiraSoapServiceImpl of the stock SOAP server library). This interface has the definitions of all the above functions, each of which will be explained below.

getProjectID() is a function that takes the issue key as the input and then gives us the project id associated with this issue. The project id is used as one of the keys in the NIMBO table that has all the values associated with NIMBO, so once we have this value, we can find out all the details in that table and pass it to the client script. We also have the logger program “LoggerUtils.Java” similar to above client side program that logs all the activities of the Soap server too, so that we can use this information to troubleshoot the step at which the program stopped the last time, and make sure that this does not happen again. In the getProjectId(), the logger logs the entering stage into this function, the issue key used as an input and also the exit stage out of this function.

getNimboDetails() is the main function which is used to get all the Nimbo related attributes by querying the PROJECTNIMBO table and giving out the results to the client side
program. For the return type, we user a special class called “NimboProjectBean” that has the variables for the Project ID, Role, Image ID, Apt repo location, Cookbook Repository location as projectId, role, imageId, aptRepo, cookbookRepo respectively. We have corresponding functions getProjectId(), getRole(), getImageId(), GetAptRepo(), getCookbookRepo() that return the values of Project ID, Role, Image ID, Apt repo location, Cookbook Repository location respectively. We also have the functions setProjectId(), setRole(), setImageId(), setAptRepo(), setCookbookRepo() to set the values initially for all the fields of the PROJECTNIMBO table. In the execution of this function, we first establish a ODBC connection by using the standard “ConnectionFactory” class provided by the standard SOAP server. Once the connection is made, we then use the Project ID from the above getProjectId() function and run the query “select * from PROJECTNIMBO where projectname='"+projectId+'"' to get all the above mentioned Nimbo values. Now we assign all the values obtained from this query to an object of the “NimboProjectBean” class, and this object is then passed to the client side program. In this function, the logger program logs the entering stage of the function, then the stage where the ODBC connection is made and tells us if the connection was successful or not. It then finally tells is about the exit stage of program.

The mail sending capability of the client program is actually implemented by using the sendMail() function of the Soap Server. It takes the input as the username used to send the mail and also the mail content that needs to be sent to the user. This sendMail() function uses the SMTP Mail send capability of JIRA to send a mail to any user when significant changes occur in the system. In this case that significant change is the launch of an image for that particular issue by clicking the “LAUNCH NIMBO” button.

In this case, we use the Gmail SMTP server on the JIRA server to send a mail to him. To set this up, we have to first get the certificates used by Gmail on port 465, and then import this certificate in JIRA by using the Keytool utility provided by the java sdk files. We now have to transfer the activation and mail jar files from the “WEB-INF/lib” in the jira install directory to the lib directory used by the tomcat server. Finally we need to add Gmail as a JNDI resource in the server configuration file, and give it all the details like the username, password, smtp server address etc for the user that would be used to send out the mail to user that clicked the “Launch Nimbo” button to start the image on Amazon EC2.
For programming of the project, we used an agile development model[9] in which the requirements were coded as soon as they were available, and we just kept on making changes if we hit a certain roadblock which was technically not possible to implement. We then did some design changes[10] to our existing design, and then coded that change into our product.

All the parts in the Ubuntu Client Program and the SOAP server Program work together as an interactive program so that at one time, a function will be waiting for input from some other function, and once it receives the input, it processes that input and then sends it output to another function which in turn is waiting for it. All these actions working together make the NIMBO product work.
CHAPTER 4

RESULTS

The purpose of making this Nimbo system was to design a very hands-off, automated system which would take no input from the user, and give it a ready to work machine so that the user (which in most case is a developer of the company using Nimbo) does not waste any time in getting the required hardware of the machine either by purchase or contacting it’s IT department. We also have think about the extra time that the IT department would take to allocate the budget to buy the machine. He then has to waste time in either installing all the software on the server machine or explaining all his requirements of software on that machine to the IT department and then wait for the time that the IT department takes to install the softwares on the machine. Sometimes, the communication from the developer can be misinterpreted, and the machine with installed software might not be the final product desired by the developer, so the same user has to wait for an additional time for all the desired corrections to be made to the machine. In some cases, the IT department cannot get to the point of servicing the request of the developer at the same time, due to the pending queue of issues from other users of the company, thus it could result to a delay of several days before the IT department can begin the work on the request of developer in question.

By using the Nimbo System, the developer skips through all those delays and gets a machine with the necessary processing power of the hardware and all the softwares pre-installed on the machine. Once he clicks the “Launch Nimbo” button, a command is sent to the Nimbo system which makes the new machine, installs all softwares on it, and once everything is finished, the user is sent an email with the instructions to connect to the machine using a certain service like ssh, Remote Desktop , vnc etc.

This system gives the developer more time to work on a particular issue, so that he can troubleshoot the problem in the product quicker, and give the resulting solution to the problem to this organization. Once he is done, he does not need the machine anymore, he tells the IT department that he is done with his work. Now all those resources are just being
wasted, the money spent for those resources is also effectively being wasted until the time these resources are reconfigured and then rerouted to another user.

In this Nimbo System, we have used the Logger utilities which give us the output of various stages of functions/methods inside a text file. The logger program (LoggerUtil.java) is a part of internal design of all components. For our project, we have two main systems running as an interactive system. They interact amongst each other at different times such that one function of first project waits for an input from other function in the second project, once the first function receives that input, it does its processing and then give its output to the another function of the first subsystem.

In our project, we have two text files, the first one is the log file from the client side Java Wrapper Program and is named “JiraClientlog.txt”, the second one is the SOAP Server Log on the main JIRA server and is named “SafirJiraPluginLog.txt”. The other block of text than would help us know if all the functions in both the systems were called and correctly executed is the email that gets sent to the user which pushed the “Launch Nimbo” button in the JIRA bug tracking system.

Once the “Launch Nimbo” button is clicked, if all the actions are successfully executed, we get the following SOAP Server log (from the JIRA server) :-

```
DEBUG - Start the getProjectId method with parameters
DEBUG - issueKey : NMB-2
DEBUG - End the getNimboDetails method
DEBUG - Start the getNimboDetails method with parameters
DEBUG - Get the connection
DEBUG - connection successfully
DEBUG - End the getNimboDetails method
DEBUG - Start the sendMail()
DEBUG - create mail server
DEBUG - mail server created
DEBUG - get mail server
DEBUG - get mail server successfully
DEBUG - send mail successfully
```
The above mentioned log lists the various functions that were used while the SOAP server side program was called. As we can see in the above log, once the SOAP server log is called, the first function called is the getProjectId(), we see the control entering the function with a particular issue key which is displayed in the second line. Now the “IssueManager” class provided by the standard JIRA soap server functionality will take the issue key as the input and give out the corresponding value of Project Id. Once all this is finished, we see the third line where is exists that function.

Now, once we have the project ID, this is given as the input to another getNimboDetails(). This function first establishes a connection to the database and the table that stores the Nimbo related details, and then queries all the information associated with the project ID using a select query. The ofbiz also provides a ODBC connection which is then used to get the output. All the return values are stored in an object of “NimboProjectBean” which is then passed to the client side wrapper program listening on port 5000 for that client.

After all the processing done on the client end which would be described soon, the client sends a request to the SOAP server to send a mail with the value of the string that contains the details of the message to be sent. The control then passes to the sendMail() function which contacts the SMTP server to send out the mail. In this case, we are using a Gmail as the SMTP server to send the mail, which is configured on the JIRA server in advance. Once the connection is successfully made to the mail server we get the message “get mail server successfully”, and then the mail server sends to string passed by the client. Once that action is done, we get the final message in the log for “send mail successfully”.

We get the following log from the Client Side :-

DEBUG - Start the Client
DEBUG - Properties are
DEBUG - issueKey : NMB-1
DEBUG - username : safir
./mysqlinstalllog.txt, https://omnia.svn.codebasehq.com/nimbo-
svn/nimbo/trunk/cookbooks/apachechef/ apachechef/ -> sudo chef-solo -c
./apachechef/solo.rb -j ./apachechef/apache-dna.json -l debug >
./apachechefinstalllog.txt, https://omnia.svn.codebasehq.com/nimbo-
svn/nimbo/trunk/cookbooks/phpchef/ phpchef/ -> sudo chef-solo -c ./phpchef/solo.rb -j
./phpchef/phpdna.json -l debug > ./phpinstalllog.txt
DEBUG - TCPServer Waiting for client on port 5000
DEBUG - THE CLIENT /98.155.79.101:19540 IS CONNECTED
DEBUG - issueKey : NMB-2
DEBUG - username : raj
DEBUG - clienthostname : ec2-50-19-66-249.compute-1.amazonaws.com
DEBUG - keyname : i-6e6e6101
DEBUG - call getProjectid() with issueKey : NMB-2
DEBUG - return getProjectId() with projectId : 10000
DEBUG - call getNimboDetails() with projectId : 10000
DEBUG - return getNimboDetails() with NimboProjectBean : com.safir.jira.soap.beans.NimboProjectBean@526b3be6
DEBUG - call sendMail() with username : raj and mailcontent : Hi raj,
You have created a new machine with the required software installed as per the mysql. You
can connect to the machine using the following command -:
ssh -i i-6e6e6101 ubuntu@ec2-50-19-66-249.compute-1.amazonaws.com
For e.g. -:
ssh -i raj-ec2-keypair ubuntu@ec2-50-16-128-33.compute-1.amazonaws.com
For getting the key in the above mentioned command and any additional questions, please
contact your administrator. You can now connect to the machine, and start your work. You
can even install any other software which is not included in the role.

Thanks for using the Nimbo Cloud Service,

Admin
Nimbo Cloud Service.
nimbo.cloud@gmail.com
DEBUG - return sendMail() with sendmail : true
DEBUG - cookbook file :https://omnia.svn.codebasehq.com/nimbo-
svn/nimbo/trunk/cookbooks/mysqlchef/ mysqlchef/ --username nimbo --password nimbo123
DEBUG - *******************INPUT*******************
DEBUG - svn co https://omnia.svn.codebasehq.com/nimbo-
svn/nimbo/trunk/cookbooks/mysqlchef/ mysqlchef/ --username nimbo --password nimbo123
DEBUG - *******************START OUTPUT*******************

The client side program also logs the entrance and exit to every function in the logger file. The first line states that we have reached the client successfully”. We then set various attributes related to Nimbo like the issueKey, username, serviceUrl, and the command that needs to be executed on the client side. Once all these values are set, the client program then suspends its execution and then waits for an input by the server on port 5000. Once the client is called by the server side “Image Creation” program, the server side also sends the issue key and the username of the user that clicked the “Launch Nimbo” button to this client script. In the meantime, the client script runs the commands to get the hostname of the local machine by using “hostname” command and also the keyname associated with the image. The control is then transferred to the getProjectId() function on the client which uses the above mentioned value of the issue key and then in turn calls the getProjectId() on the Soap server program. Once the Soapserver gets the project Id associated with this Nimbo Project, it then calls the getNimboDetails() on the server side SOAP program which gives it all the Nimbo related attributes. After we get the input from the server side, we call the function sendMail() on the server side which sends the mail to the user using the Gmail smtp server setup on the JIRA server. Once we have all the Nimbo related attributes are given to the client, the client side program then first downloads the cookbooks associated with the Nimbo role, and then runs the chef-solo command which installs all the softwares related to that particular role on the client.

The client side program first composes the body of the email that will be sent to the user. It then calls the sendMail() function is server side SOAP program which uses the Gmail smtp server configured on the JIRA server side to send the following mail to user :-:
Hi raj,

You have created a new machine with the required software installed as per the mysql. You can connect to the machine using the following command :-:

```
ssh -i i-6e6e6101 ubuntu@ec2-50-19-66-249.compute-1.amazonaws.com
```

For e.g. :-

```
ssh -i raj-ec2-keypair ubuntu@ec2-50-16-128-33.compute-1.amazonaws.com
```

For getting the key in the above mentioned command and any additional questions, please contact your administrator. You can now connect to the machine, and start your work. You can even install any other software which is not included in the role.

Thanks for using the Nimbo Cloud Service,

Admin
Nimbo Cloud Service.

nimbo.cloud@gmail.com

If we do not get that email within a 5-10 minutes of launching the image, then the user of the system (developer) will have to contact his administrator which has access to the JIRA Server log. The administrator can then go into the JIRA Server log to check the location of the problem. The administrator also has access to the Amazon EC2 Web Management console thorough which he can see if the image was generated or not. If the image was generated, the administrator already has the key to be used to login to the image,
he can login into the machine using ssh or some other protocol and then look into the “JiraClientlog.txt”.

The administrator can also try to launch the image again by pushing the “Launch Nimbo” button again, and if the previous problem persists, he can then notify the Nimbo programmer. The Nimbo programmer also uses the above mentioned log files and the email to start the troubleshooting of the problem, so that he can determine the cause of the problem and then perform remediation actions that would make sure that the problem does not occur again.
CHAPTER 5

CONCLUSION AND FUTURE RECOMMENDATIONS

Cloud computing is one of the most popular words in the computer industry today. Every company wants its resources to be available 24 hours a day, 7 days a week, and at the same time maintain the highest degree of security for that information. Cloud computing is one of the concepts, that helps them do that. The high availability of machines due to duplication of the machines across different geographic boundaries along with load balancing can ensure that information is highly available, and the faulty servers get replaced with the new ones, and this transaction is invisible to the user using the system. One of the major reasons of using Amazon EC2 in our project was that the Service level agreement of Amazon dictates that their services would be available 99.95% of the times; this translates to a downtime of less than 5 minutes in a calendar year of 365 days. This makes sure that the customer that uses our product would have the same amount of SLA, which is a huge advantage to us when compared to someone else with a similar product not on Amazon Web Services. Amazon even dictates that the security of information is of utmost importance to them, and hence they have ISO 27001 and multiple SAS 70 Type II audits, also they have made adequate provisions in the physical security and their architecture which makes it almost impossible for a hacker to infiltrate their system and steal some of that information. These security practices again give out product a major advantage and would be acceptable to companies with highest emphasis on security. But for some companies, their requirements dictate that all the systems and their information are limited to their internal networks are protected by firewalls and DMZ’s which allow only traffic bound with some rules. Some companies are very peculiar about sending their proprietary data to another company, even if it follows all the security practices mentioned above for Amazon. As our product is cloud centric, and if a company does not want information on Amazon, this would be one of the major deal breakers. In that case, we can convince the customer to use an internal cloud
technology called Eucalyptus which would enable them to make a cloud similar to Amazon inside their own infrastructure. The starting cost for all the hardware and software would be expensive in that option, but would provide peace of mind to the company and their stakeholders. The API and the architecture of Eucalyptus is almost similar to Amazon Web Services and all their commands also match, barring the initial letters, for e.g.- > To run an instance on Amazon EC2, we use the command “ec2-run-instances” and for Eucalyptus, we would use “euca-run-instances”. Currently our product only supports Amazon Web Services as the only cloud service provider, but in case the above option is required, we can customize our product to match the revised requirements with minimal changes to the code of the product and maintain all the feature set provided by it. This would be something which we could add to our product in the subsequent releases.

In this version of the product, we are just using Ubuntu as our primary Operating system, and are assuming that all the companies who would be interested in this product would have their softwares installed on the Ubuntu machine. One other reason of working with only one operating system is that it would give us power to use only a certain set of tools that could be used with Ubuntu and not worry about including the other alternatives to those tools, and increase the complexity of our system. Ubuntu is also a free operating system and that removes us from worrying about the licensing of operating systems for different systems generated by various developers using the same system at the same point of time, no license keys are required for activating individual copy of Ubuntu and thus the emphasis could be on completing the work assigned to the developer rather than worrying about legal issues of using an invalid license key.

Ubuntu can also be run on a wide variety of hardware and supports a large amount of processing power and memory to be used with it and pass on all processing power to the applications running on it, it does not use a large amount of operating system processing power and memory, and takes a backseat in usage of resources when compared to a server application that can be used by thousands of users at the same time, for e.g.- > in case of a Web server. Currently, we are using only one type of operating system as a base for all our roles, but as the time goes on, and we get more help in the development department, we can port our product to other operating systems like Red Hat Enterprise and Centos Linux (which are Red Hat based), as compared to Ubuntu (which is Debian based). Also once we get more
understanding of the licensing issues and scan all the differences that our product would have to go through, we would port it to Microsoft Windows operating systems too.

Once the user clicks the “Launch Nimbo” button, and the image is generated on Amazon EC2, a user is sent an email with the instructions for how to connect to that machine. Currently, As we are working with a command line Linux based operating system, we are content with using only protocol to connect to those machines i.e. ssh. Another advantage provided by Ssh is that the connection session between the user and the machine on Amazon EC2 is encrypted, so any hacker cannot intercept the important information by using a packet sniffer or something else, and even if he intercepts the information, it would be almost impossible for him to decrypt that information and make sense of it. As all the communication between the user is currently, command line based, using only ssh makes perfect sense as it provides a terminal based console to the user to login and interact with the system. As the time progresses, we will add the support for graphical based interaction with the generated images by using the X11 based libraries for the Unix machines, or by using Windows operating systems. In that case, we will evaluate the security of other protocols like VNC with our product, and then allow users to connect to those machines using this protocol and are primarily used in case of GUI (Graphical User interface) access for the users interacting with the system. We would even study the security and other requirements in case a Windows Remote Desktop service is to be used to connect to the instantiated machine on the Cloud, enable that service on the skeleton server machine and open the corresponding ports on the Cloud to allow connection to that service.

APT (Advanced Packaging Tool) is the only package manager being supported by our product with the Ubuntu type of machines at this time. The main reason for doing that is that APT is the most popular package manager in the market today for Ubuntu and has large amount of repositories spread amongst different geographical locations around the world. As a result, even if one of first repositories used to fetch packages goes down, we always have other package repositories to take its place, and this architecture gives us great advantage in making sure that the data is always highly available. We also have the control to modify the order in which the repositories would be contacted to fetch the packages required for installation by changing the order of repositories in the “/etc/apt.conf” file for the image. All these advantages makes APT an ideal candidate for the first package manager of choice. But
in some cases, where the customer does want us to use APT, we have to be prepared for that and make changes to the product so that it is not dependent on only APT to fetch and install all the softwares on the image.

For automatic installation of softwares, we are currently using Chef framework, and are currently checking out cookbooks from a particular Subversion repository, and finally installing those softwares using a standalone component of Chef called “chef-solo” which does the ruby cookbooks translation to the APT based commands on the Ubuntu machine. In case, the customer does not require APT, we have to be prepared in advance and look into alternate ways of installing softwares on our machine. We have other some other competing products like Chef in the market like Puppet or Engine Yard which could be used for automatic installation of softwares; we can always look into them in case a customer requires us to use these products instead.

When we were designing the product, we had thought about the option of providing all the customers a way to install their own custom softwares on the all the machines that would be generated on the cloud (either on Amazon or Eucalyptus), but that would require them to make their product as an APT based installation software. As we don’t have any customer right now, we have not included that option in the basic design of our product and only allowed the installation of all the basic softwares which could be used as a major service in different situations like using those machines either as a pure Web server (using Apache), database server (by using mysql), or a PHP server (by using PHP installer with Apache as the webserver). As we sell this product to a customer that is interested in installing their own custom software, we would have to make only some minor changes to the product and make sure all their requirements are satisfied.

Technology keeps on evolving, and there are new and interesting products available on the market every day. The saying that the only “Change” is the only thing constant in life holds true for all the industries, especially for technical industries in companies always search for new technologies to grab market attention and thus score a financial win on their competitors. We would also keep evolving with market change and keep on adding new features to this Nimbo product either as a request from specific customers or by looking out for alternatives for all the above mentioned technologies, and incorporating that into our product to make sure that our product is best in the market.
REFERENCES


APPENDIX A

USE CASES
Use Cases

A. Nimbo Push Button interface (Functional Requirements, Mockup, Design)
   a) **Use Case A-01**: Spawn an image via the click of a button on a Web Interface.
   b) **Use Case A-02**: Email Notification - IP address of machine with instructions.
   c) **Use Case A-03**: Email Notification - Access rights
   d) **Use Case A-04**: Email Notification - Instructions for additional Manual steps
   e) **Use Case A-05**: Email Notification - General image information, installed packages and applications

B. Communication through SOAP Protocol

Use Case Descriptions

A. Nimbo Push Button interface
   a) **Use Case A-01**: An engineer is looking through a bug record in an issue tracking system. He/she decides to work on this issue but needs to setup a “production like” system. The engineer does that by clicking a button on the bug report.
   b) **Use Case A-02**: An image has been started via the click of a button on a web interface. Once the image is fully configured and running the engineer will receive an email with the IP address and/or name of the machine. Clear set of steps on how to login with what utilities needs to be part of the email communication.
   c) **Use Case A-03**: An image has been started via the click of a button on a web interface. Once the image is fully configured and running the engineer will receive an email with the username/password combination and or randomly generated ssh key for password less authentication.
   d) **Use Case A-04**: The notification email needs to list the additional manual steps necessary required to fully configure the development and or test environment.
   e) **Use Case A-05**: The engineer will receive an email after the startup of the image. As part of this email communication the name of the image, Operating system, Version, Architecture, list of dependent installed packages and applications need to be listed.

B. Communication through SOAP Protocol
   a) **User case B-01**: Client Program on the EC2 Machine would act as a SOAP client and connect to the SOAP server on the JIRA server.
   b) **Use Case B-02**: Standard RPC Soap Service provided by JIRA needs to extended so that it provides Nimbo specific attributes.
c) **Use Case B-03:** Soapclient authenticates itself with the SOAP server by using the username and password of the user that has access to query the attributes associated with an issue.
APPENDIX B

ACRONYMS
## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SOAP</td>
<td>Simple Object Access Protocol</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
</tr>
<tr>
<td>EC2</td>
<td>Elastic Cloud Computing</td>
</tr>
<tr>
<td>S3</td>
<td>Simple Storage Service</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>AMI</td>
<td>Amazon Machine Identifier</td>
</tr>
<tr>
<td>SaaS</td>
<td>Software as a Service</td>
</tr>
<tr>
<td>PaaS</td>
<td>Platform as a Service</td>
</tr>
<tr>
<td>IaaS</td>
<td>Infrastructure as a Service</td>
</tr>
</tbody>
</table>