CLOUD MERGE:

HETEROGENEOUS CLOUD APPLICATION MIGRATION USING PLATFORM AS A SERVICE

by

MAYANK JAIN

Presented to the Faculty of the Graduate School of

The University of Texas at Arlington in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

THE UNIVERSITY OF TEXAS AT ARLINGTON

December 2016

ACKNOWLEDGEMENTS

Firstly, I would thank my advisor Prof. David Levine for his valuable guidance and support, and his tireless guidance, dedication to his students and maintaining new trend in the research areas has inspired me a lot without which this thesis would not have been possible.

I would also like to thank the other members of my advisory committee Dr Ramez Elmasri and Dr. Farhad Kamangar for being part of my thesis committee and offering insightful comments. I appreciate all members of cloud research team, Shraddha Jain and Samvaran Kashyap for their support during my research work.

Finally, I am grateful to my family, my father Mr. Sanjay Kumar Jain, my mother Mrs. Anjula Jain for their support, patience, and encouragement during my graduate journey.

November 15, 2016

ABSTRACT

IMPLEMENTATION HETEROGENEOUS CLOUD APPLICATION MIGRATION USING PLATFORM AS A SERVICE

Mayank Jain, M.S

The University of Texas at Arlington, 2016

Supervising Professor: David Levine

With the evolution of cloud service providers offering numerous services such as SaaS, IaaS, PaaS, options for enterprises to choose the best set of services under optimal costs have also increased. The migration of web applications across these heterogeneous platforms comes with ample of options to choose from, providing users the flexibility to choose the best options suiting their requirements. This process of migration must be automated to ensure the security, performance and availability, keeping the cost to be optimal while moving the application from one platform to another. A multi-tier web application will have many dependencies such as the Application Environment, Data Storage and Platform Configurations which may or may not be supported by each of the cloud providers.

Through this research, an automated cloud-based framework to migrate single or multi-tier web applications across heterogeneous cloud platforms is presented. Heroku and AWS (Amazon Web Services) cloud platforms are used as examples in this paper. The proposed framework can be extended to support more cloud providers in future such as Microsoft Azure, IBM Bluemix, Openstack etc. Observations on various configurations required by a web application to run on Heroku and AWS cloud platforms have been presented and discussed. This research will show how, using these configurations, a generic web application can be developed which can seamlessly work across multiple cloud service platforms.

Finally, this paper shows the different experiments conducted on the migrated applications, considering the factors such as scalability, availability, elasticity and data migration. Application performance was tested on both the AWS and Heroku platforms, measuring the application creation, deployment, database creation, migration and mapping times.

TABLE OF CONTENTS

LIST OF ILLUSTRATIONS

LIST OF ACRONYMS

AWS – Amazon Web Services

CLI – Command Line Interface

Db - Database

EB – Elastic Beanstalk

EC2 – Elastic Cloud Compute

GCS – Google Cloud Services

IaaS – Infrastructure as a Service

PaaS – Platform as a Service

QOS – Quality of Service

RDS – Relational Database Service

SaaS – Software as a Service

SLA – Service Level Agreement

S3 – Simple Storage Service.

CHAPTER 1

INTRODUCTION

1.1. Introduction and Background

The cloud service platforms helps us to reduce the time to market the applications. They offer on-demand scalability at very low cost for the enterprises. With so many evolving services capturing a huge market share, cloud computing has emerged as a hot research area. In order to take the full advantage of the features provided by different cloud vendors, applications should be deployed across multiple cloud platforms depending on what feature is required from which vendor.

There are “n” number of cloud providers providing similar services. Each server has provider specific SLA, cost, availability and latency. The cost of the cloud services frequently changes over a period of time (which can be a day, month or year). An efficient cost monitoring system needs to be designed to get an up to date cost of each cloud service. In the scenario of typical web-based architecture in cloud services each application will be needing a combination of one or more of the following services.

- Compute
- Storage
- Database
- Network

Resource provisioning in the cloud providers can be either on-premise infrastructure called private cloud or on a public cloud (such as Amazon AWS, Google cloud, Rackspace, etc.,) sometimes the solution can be distributed across the private infrastructure and public cloud called as hybrid cloud. Hybrid cloud deployment has its pros and cons. Patterson's paper [2] discusses some of the major problems faced by the cloud-based IT provisioning. Among the given problems "Vendor lock-in" is one of the most predominant one.

Tech Target [14] states "Vendor lock-in is a situation in which a customer using a product or service cannot easily transition to a competitor's product or service. Vendor lock-in is usually the result of proprietary technologies that are incompatible with those of competitors. However, it can also be caused by inefficient processes or contract constraints, among other things."

Example: Consider a consumer who is maintaining an application on Google app engine which provides an abstraction layer over the Google infrastructure. A user will be provided a code deployment tool (Google GCS client) to run their application on app engine. However, to deploy the application, users must package their application adhering to standard format (i.e., file structure and YAML) imposed by Google. Amazon also has a similar service named as "elastic beanstalk". If a user wants to migrate the application from the google app engine to Amazon elastic beanstalk and vice versa periodically for cost optimization or various other factors such as service unavailability and data unavailability. A user need to come up with their own tools and strategy for the transition which needs substantial efforts and expenses.

In this process of migrating the application, the user also needs to decide whether to port the services bound to the application from source cloud to the destination cloud. There might also be a scenario where the services bound might not be found at the destination.

In this thesis, an automated cloud-based framework to migrate single or multi-tier web applications across heterogeneous cloud platforms is presented. This research discusses the migration of applications between different cloud providers, Heroku and AWS as examples in this research. Observations on various configurations required by a web application to run on Heroku and AWS cloud platforms have been discussed. Then we show how using these configurations we can develop a generic web application which can seamlessly work on both the cloud platforms.

Further we show how to attach and migrate different components to an application such as database, middleware and environment configurations. Finally, we show the different experiments conducted on the migrated application considering the factors such as scalability, availability, elasticity and data migration. Application performance was tested on both the platforms measuring the application creation, deployment, database creation, migration and mapping times. Also application performance was analyzed by stress testing the web application with 100, 1000, 10000 and 100000 DML operations.

1.2. Motivation Behind the Thesis

In a traditional web application model, the underlying hardware should be able to handle the peak load on the application. There are times when the traffic on the web application may be uncertain and the unexpected variations may result in underutilization of expensive resources. Therefore, provisioning for peak workloads leads to unused [3] computing cycles when the traffic is low on the application. With the evolution of cloud computing, cloud services offered for hosting the applications are elastic and matured for handling the on-demand traffic during the peak hours and automatically scaling down when the traffic is low. This causes enormous cost savings for the users, saving any upfront cost on the hardware resources.

The main cloud services can be classified in the following layers:

1) Software as a Service (SaaS)

2) Platform as a Service(PaaS)

3) Infrastructure as a Service (IaaS)

The advent in cloud services also bring along some extra costs and risks for the users. The major issues faced by users with cloud providers are:

Vendor Lock-In, Service not Available, Data Lock-In, Services/ Platforms not supported by the provider. Other factors may include:

- Elasticity and resource availability:
- Vendor Lock-In [13] is one of the major obstacles for wider cloud adoption. In current cloud status, customers are often locked to a specific cloud vendor product or service, and an easy transition to a competitor

does not exist. Lack of interoperability and portability spans the complete cloud stack, embracing data, applications and infrastructure. Development of a cloud market that considers utilizing resources from multiple providers in a transparent, interoperable, and architecture independent manner can help the cloud users to overcome the existing vendor lock-in fears and develop a cloud market in which freedom of choice prevails.

- Distribution across geographies for reducing latency, address legal constraints and enable high availability [13].
- The multi-layered nature of clouds brings concerns for users in regards to regulatory context. Existing worldwide established providers address this issue, by offering diverse regions with limited level of automation among these. This mechanism it is also offered to support high availability. Beyond these, increased automation among diverse cloud offerings in different geographies can satisfy increasing demands for user businesses to act at a global scale, fulfilling specific applicable [13] regulations, automating high availability across clouds while addressing needs spread service consumers.

1.3. Goals of Thesis

Through this paper we first present different challenges a user faces with the cloud vendors while hosting their multi-tier web applications.

After that we will show the limitations of keeping the application spread just over one cloud provider. In the Rest of the paper we will discuss our work on development of an automated framework, which can migrate an application from the local environment to AWS/ Heroku cloud or can also port the application across multiple clouds. We will discuss approaches to develop a generic framework supported by different cloud providers and the configurations required for porting the application.

After that we will present our experimental results for migrating an application from one provider to another with the performance analysis and comparison of the cloud providers being used in terms of compute, cost, query analysis, deployment times and efficiency. Finally, we will conclude with the observation results and summary along with proposal for future works.

1.4. Organization of the Thesis

This thesis starts with an introduction and background, motivation and goals of the thesis. Chapter 2 gives an overview of the related works. Chapter 3 defines the problem statement. Chapter 4 discusses about the preliminary analysis on AWS and Heroku cloud platforms. The manual implementation of single and multi-tier application deployments on Heroku and AWS clouds. Chapter 5 gives details of the Cloud Merge

Framework and its API architecture. Chapter 6 gives the implementation details of Cloud Merge and various deployment strategies.

Chapter 7 details about the various experiments done on deployment, tear down, and DML operations such as SELECT, INSERT, UPDATE on the applications migrated to AWS and Heroku clouds using Cloud Merge framework. Chapter 8 gives the summary of the experiments and concludes this thesis. Chapter 9 outlines some of the future works that can be extended over this thesis.

CHAPTER 2

RELATED WORK

There are several initiatives [1] and standards that target services deployed on the cloud, and aim at guaranteeing properties such as Quality of Service. These initiatives use deferent approaches, with the consequence that software developers either have to develop special APIs or programming models to code their applications, or to model them using project-specific domain languages.

This paper by Sea Clouds [1] discusses an adaptive and efficient approach for homogenizing the management of the cloud applications over multiple cloud providers. They proposed an approach for achieving “Agility After Deployment” [1] by tackling the problem from the service orchestration perspective. With the open source framework for web application management across multiple cloud providers, SeaCloud proposed standardization for PaaS monitoring services. This helped us in our research to consider various metrics while migrating the applications such as Quality of Service, high availability and cost optimization considerations across multiple cloud providers while migrating the application.

The TOSCA paper [2] proposed at UC Berkley discuss about the top obstacles faced by users while deploying the applications across cloud platforms. The problems discussed included the metrics such as Availability of Service, Data Lock-In, Data Confidentiality and Audibility, Data transfer problems such as data bottlenecks, parameters such as performance unpredictability over the cloud platforms and even software licensing issues with cloud vendors.

Cloud Genius [3] paper discusses about the multi-component web services being introduced and defined by the web service community which is discussed in CAFÉ [4] and TOSCA [5], which sets standards for many cloud computing research works. Cloud Genius presented a hybrid approach that combines multi-criteria decision-making technique with evolutionary optimization techniques for helping the application engineers with the selection of best service mix at IaaS layer and enabling migration of applications clusters distributed across the clouds.

Inter-cloud Challenges, Expectations and Issues Cluster paper [6] discusses an approach to create a critical mass of projects addressing the topic of multi-cloud and inter-cloud so to share experiences, collaborate on approaches and discuss challenges for adoption and future research.

Right Scale [7] blog discusses about managing multiple clouds and different approaches to follow for Cloud Management, Migration and Deployments. The blog discusses about the factors that may be considered while migrating to any cloud providers such as Operating system versions, SSL terminations, Licensing, Database I\O requirements etc. Their solution provides an implementation for multiple cloud management, which offers a self-service cloud portal leveraging a multi-cloud framework which can manage which application of user should be migrated to cloud.

Door Dash [8] proposes an approach to migrate an application from Heroku to AWS using Dockers. The initial application deployment was on Heroku cloud but they had to migrate the platform for their application as they scaled up on the user base. The major reasons for switching to AWS from Heroku were performance of Heroku dynos which were performing poorly even after lot of tuning and required much more computation as compared to equivalent AWS EC2 instance compute. Other factors included cost efficiency in which Heroku dynos were very expensive as compared to AWS instance. [8] For roughly the same price as a Heroku “2x” dyno with 1GB RAM, they could have rented an Amazon c3.large EC2 instance with 3.75GB RAM. Other issues faced by them included reliability and control on the application which was more flexible on AWS. These observations by Door Dash helped us to work on strategy for our application to move across the clouds.

CHAPTER 3

PROBLEM STATEMENT

3.1. Introduction

The problem this thesis addresses is autonomous migration of multi-tire web applications across the heterogeneous cloud platforms. The cloud providers included in the implementation as examples are AWS and Heroku. The need arises with the frequent downtime and blackouts from cloud vendors, it is often the loss of users hosting the applications on such platforms.

In these situations, users, might want to port the application to some other cloud provider to avoid the downtime. Also, this thesis will address the problems of Vendor lock-in and Data lock-in as discussed in Patterson's paper [2].

Data lock-in [2] is a situation where customers are not able to extract the data out of a cloud provider easily. The situation may be understood as a web application hosted on Heroku cloud with Postgres database. Until some time back Postgres was not supported by AWS and thus Heroku users were not supported with migrations from Postgres DB on Heroku to Postgres DB on AWS. In this case, it may be a beneficial situation for cloud vendor but extracting data out of one service provider to other becomes a tedious job and prevents user from migrating the application. These types of customers may also be vulnerable to more expenses on cloud infrastructure rather than having a cost-effective solution using cloud services.

Vendor lock-in [14] is a situation in which cloud platform users are made dependent on cloud providers services such that the interfaces of one cloud provider are not supported by another cloud provider. In such cases customers are not able to use the cloud services of other vendors without substantial costs on redesign of their cloud applications. Often cloud platforms provide RESTful API services in the form of storage or compute and application services such as business analytics. With the advent of so many services, the applications and inbound functionalities becomes tightly coupled with the cloud provider API's and may become functionally dependent on the cloud provider's framework leading to a Vendor lock-in.

The benefits that the customer gain having so many RESTful API's are that these services offer advanced features to enterprises such as auto-scaling and auto provisioning. Often cloud provider offer data storage services which are native to the cloud vendor. These services may offer better performance for the application since using the native services removes the abstraction layers and remove the burden of translating the platform specific calls for the native cloud provider. Also, native services offered by cloud vendors include default security groups for governing and managing the hosted cloud application. Implementation of these services often requires the design of the hosted application to be modified as per the supported frameworks for native cloud vendor which may not be supported by other cloud service providers. Thus, despite of having so many benefits of with native cloud services, the risk of locking the application to a cloud provider always exists [14].

Through this paper we will address the above-stated problems with a proposed implementation of a generic model for migrating web applications and its dependencies from one cloud provider to another. We propose an approach in which user can migrate the applications from local environment to a cloud environment or from one cloud provider to other. User may even choose to have heterogeneous cloud deployments keeping the application hosted on one cloud and move just the database to another cloud provider. In this approach, we develop a generic web application supported by both of our test bed cloud environments Heroku and AWS. We will also see the performance analysis, deployment times and query analysis on the migrated applications on both Heroku and AWS cloud platforms

CHAPTER 4

PRELIMINARY ANALYSIS

4.1. Introduction

This chapter describes various milestones of this thesis. The initial phase of the thesis involved the study of the operational and functional details of multiple cloud providers. This comprised of the period to learn about various cloud platforms. The cloud providers chosen for our analyses were AWS and Heroku. The reason for choosing these cloud platforms was that they came with good documentation support. They are more stable in terms of the services we were considering for our analysis and both cloud platforms supported interoperability of the web application services across the cloud platforms.

4.2. POC Single Tier Web Application

A cloud web application is an interface that runs on a environment hosted on cloud platform. Cloud applications usually are a mix of features of a desktop application and a web application. This means that cloud-based application can be fast in terms of responsiveness as provided by desktop applications, running on local machines and the portability of web applications which can be managed from remote machines. Thus, cloud applications gives full control to the user to manage and update the files from any location over the web, saving the storage space on the user’s computer.

Considering the above-mentioned features, we choose Flask as our web application development framework for this thesis. [15] Flask is a micro web framework written in python and based on the Werkzeug toolkit and Jinja2 template engine. It is BSD licensed. The Flask framework is very popular among python web application developers as web frameworks can be very quickly designed and implemented using Flask. It’s easier to learn where developers don’t need to follow any MVC or MTV architectures. Another benefit that Flask offers is that its API-driven model and user can very easily integrate external extensions such as Flask-sqlalchemy to interact with the database layer. When compared to Django framework templates, Jinja2 templates offer faster responses than Django templates.

Using the Flask framework and Jinja2 templates, a simple single tier python web application with request/ response model is developed. There is no middle tier or database layer involved in the application. The primary purpose of developing this web application is to understand the manual deployment process on AWS Elastic Beanstalk. Further in this chapter we will see different features Elastic Beanstalk offers and different configurations required for deployment of application on AWS cloud. After successfully understanding the EBS deployment of the application, we used the same application, without changing anything in the code structure and analyse the possibilities and configurations required for deploying same application on Heroku cloud.

4.2.1. Single Tier Web Application Manual Implementation on AWS.

Preliminary Implementation is carried out by developing a basic single tier python web application using Flask framework. We tried deploying this application on AWS cloud using the manual command line interface tools. This is done to understand the deployment process of a python flask single tier application on AWS using Elastic Bean Stalk service.

AWS Elastic Beanstalk [16] is an easy-to-use service for deploying and scaling web applications and services developed with Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker on familiar servers such as Apache, Nginx, Passenger, and IIS. The deployment of code is very simple using this service. User can simply upload the application code, and Elastic Beanstalk will automatically handle environment configurations such as deployment, capacity provisioning, load balancing, application health monitoring and auto-scaling. The beauty of using Elastic Beanstalk is that user has the full control over the deployed resources which can be accessed very easily.

With Elastic Beanstalk, we can simply create an application version number for applications. The source code can be stored anywhere on S3 storage on AWS or on remote GitHub repository. After creating the application version, we launch an environment to deploy the application. The Elastic Beanstalk automatically manages the environment creation and handling configurations such as load balancers, auto-scaling. This means that during the peak loads on the application, additional EC2 instances will be automatically added to the environment and terminated when not required. Any application running on an Elastic Beanstalk instance will have a URL or web address, through which it can be accessed using a web browser.

The EB CLI is the command line tool implemented using Python using the python SDK for AWS. This tool uses the boto client for python to interact with AWS. For deploying a python application on Elastic Beanstalk there are some prerequisites for development environment.

- Python 2.7 or 3.4 should be installed.
- The pip utility matching our python version should be installed. This utility helps to setup and install all the dependent python libraries that will be required by the web application.
- The virtual environment package should be installed. This package helps us in developing and testing our application by replicating it in Elastic Beanstalk environment. It maps all the dependent libraries and automatically installs them in the AWS EBS environment.
- The AWS CLI package should also be installed. This package helps in the configurations of the application files making them compatible with the AWS environment.

- To Access the running application on AWS environment we also need ssh client to be installed in development environment. This utility is helpful when we want to access any application log files or want to debug any errors during application deployment.

EB CLI provides the command line interface to interact with the Amazon Web Services. The initial steps is to initiate the Elastic Beanstalk application in development directory. The following commands as described by AWS CLI documentation were executed [17]:

- On the command prompt of web application directory, *eb init* command was run.

```
~/eb $ eb init
Select a default region
1) us-east-1 : US East (N. Virginia)
2) us-west-1 : US West (N. California)
3) us-west-2 : US West (Oregon)
4) eu-west-1 : EU (Ireland)
5) eu-central-1 : EU (Frankfurt)
6) ap-south-1 : Asia Pacific (Mumbai)
7) ap-southeast-1 : Asia Pacific (Singapore)
8) ap-southeast-2 : Asia Pacific (Sydney)
9) ap-northeast-1 : Asia Pacific (Tokyo)
10) ap-northeast-2 : Asia Pacific (Seoul)
11) sa-east-1 : South America (Sao Paulo)
12) cn-north-1 : China (Beijing)
13) us-east-2 : US East (Columbus)
(default is 3): 3
```

- EBS CLI would ask for AWS security credentials which would be generated while aws account creation. These can also be found in AWS Identity and Access Management console.

```
You have not yet set up your credentials or your credentials are incorrect
You must provide your credentials.
(aws-access-id): AKIAJOUAASEXAMPLE
(aws-secret-key): 5ZRIrtTM4ciIAvd4EXAMPLEDtm+PiPSzpoK
```

- Now EB CLI would ask you for options such as creating a new application or choosing an existing one. As mentioned earlier, an EB application will be a combination of various resources such as application versions (a new version will be created every time when the application is deployed), an environment to deploy the application and associated resources. User will also be prompted to enter the language for the platform in which web application has been developed. In our case, we developed the application using python. User will also be required to enter/generate the key pair for accessing the application. [17] The CLI registers the new key pair with Amazon EC2 and stores the private key locally in a folder named .ssh in the user directory.

```
Select an application to use
1) [ Create new Application ]
(default is 1): 1
```

```
Enter Application Name
(default is "eb"): eb
Application eb has been created.
```

With the above mentioned setups through AWS CLI, the *.elastic beanstalk* directory will be created in the web application root directory with *config.yml* file in it. This file will contain all the configuration information about the EBS application and environment just created in previous steps. The file data will be in JSON format containing the key, value pair of different configuration options such as application_name, environment, default_platform, default_ec2_keyname etc. A requirements.txt file will also be created while creating the AWS environment. This file contains all the dependencies required by python code to run the web application in AWS environment. The EB CLI deployment uses these files to parse all the details about the web application configuration and deploy on Elastic Beanstalk.

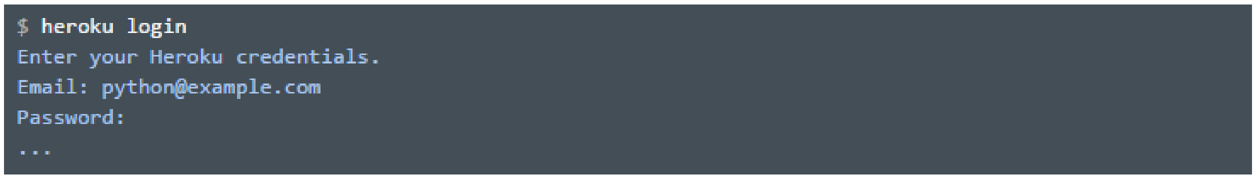
4.2.2 Single Tier Web Application Manual Implementation on Heroku.

Preliminary implementation is carried out by developing a basic single tier python web application using Flask framework on Heroku cloud. We tried deploying this application on Heroku cloud using the manual command line interface tools. This was done to understand the deployment process of a python flask single tier application on Heroku.

The initial steps to proceed ahead and deploy any application on Heroku

- Create an account on Heroku.
- Python 2.7 or higher should be installed on local machine/ development environment.
- The pip utility matching our python version should be installed. This utility helps to setup and install all the dependent python libraries that will be required by the web application.

- The virtual environment package should be installed. This package helps us in developing and testing our application by replicating it in Heroku environment. It maps all the dependent libraries and automatically installs them in the Heroku environment.
- Also, we will need the Heroku Command Line Interface (CLI) for interacting with Heroku environment from our development environment. This tools helps in managing the web application, provision and Add-On to the application, also users can view logs of the application in case of any troubleshooting is required while deploying the application or in case of any exceptions.
- Heroku CLI also helps to run and test the web application locally.
- After installation of CLI, user must log in to Heroku account using the command line. This will authenticate the user to manage their Heroku as well as git account from the CLI.



```
$ heroku login
Enter your Heroku credentials.
Email: python@example.com
Password:
...
```

For proceeding ahead, will use the same application we used in the section 4.2.1 to understand the portability and compatibility on Heroku cloud. For creating the application on Heroku we will follow the below steps:

- With the *Heroku create* command, a new application will be created in Heroku environment. With this Heroku creates a git repository (Heroku) and will be associated with the project.

```
$ heroku create
Creating lit-bastion-5032 in organization heroku... done, stack is cedar-14
http://lit-bastion-5032.herokuapp.com/ | https://git.heroku.com/lit-bastion-5032.git
Git remote heroku added
```

- To deploy the code the application created in above step, push the code to the Heroku git branch created in the above step.

```
$ git push heroku yourbranch:master
```

With the last step, the code will be fetched from the GitHub repository, dependencies will be installed in Heroku environment and application will be deployed in Heroku environment.

In the above two sections we analyzed a single tier flask application and tried to deploy on both AWS and Heroku cloud. When the application hosted on AWS was tried to be deployed on Heroku, the configuration changes required to make the application compatible to Heroku environment without making any changes to the application code were analyzed which will help us later during this research to automate the application migration from one cloud environment to another. In the next sections, we will see and analyze the manual deployment of multi-tier web applications in the similar way as in above two sections.

4.3 Multi-Tier Web Application.

A multi-tier web application consists of various components such as Presentation layer, Business logic layer, and a Database layer. Each of the component independently performs various actions and handles user request and responses. The figure 4.3 below shows a typical architecture of a multi-tier web application.

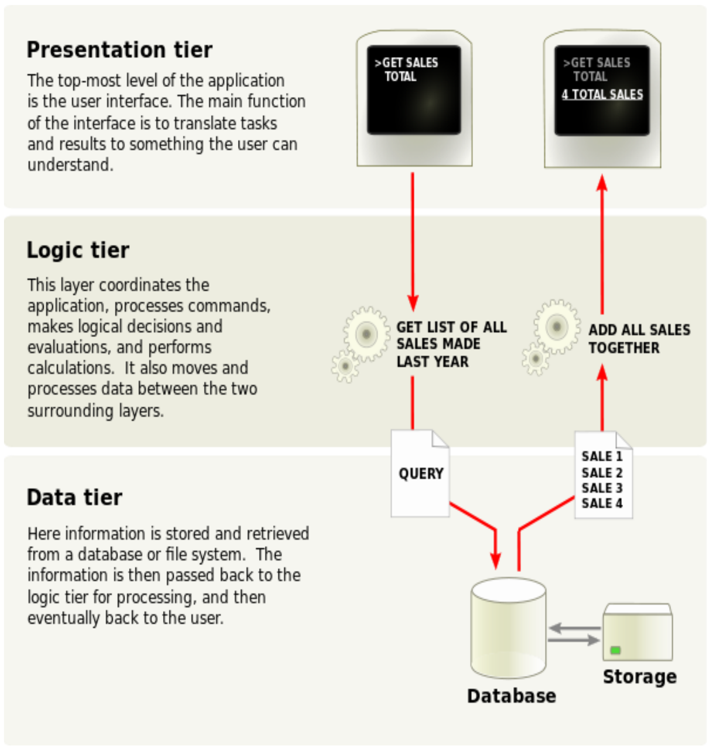

Presentation tier
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.
>GET SALES TOTAL
>GET SALES TOTAL
4 TOTAL SALES
Logic tier
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.
GET LIST OF ALL SALES MADE LAST YEAR
ADD ALL SALES TOGETHER
QUERY
SALE 1
SALE 2
SALE 3
SALE 4
Data tier
Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.
Database
Storage

Figure 4.3 Architecture Diagram of a Multi-Tier Web Application

For this research, we created a multi-tier web application with a presentation layer to handle user requests, a Flask framework to handle the business logic and a MySQL database for storing the user data. The implementation and architecture details for both AWS and Heroku clouds are discussed in the coming sections.

4.3.1 Multi-Tier Web Application Manual Implementation on AWS EB.

A basic multi-tier python Web application was developed using Flask framework. We tried deploying this application on AWS cloud using the manual command line interface tools. This was done to understand the deployment process of a python flask multi-tier application on AWS using Elastic Bean Stalk service as described in the previous sections 4.2.1 and 4.2.2.

The web application was designed using a request/response model with a data storage layer using RDS service on AWS. The purpose of the application was to accept data from the user, process through an application layer and after business logic processing store/retrieve the data from the data store.

Amazon Relational Database Service (Amazon RDS) [19] is a web service that makes it easier to set up, operate, and scale a relational database in the cloud. It provides cost-efficient, resizable capacity for an industry-standard relational database and manages common database administration task. With the RDS services, user is easily able to manage the backups, failure detections and recovery of databases. User can manage automated data backups either at scheduled intervals or user can manually take the snapshot of the database. The process for restoring a database on AWS is efficient and reliable. RDS provides a wide variety of databases that can be used with our applications such as MySQL, Maria DB, PostgreSQL, Oracle, Microsoft SQL Server, and the new, MySQL-compatible Amazon Aurora DB engine [19]. Some of the benefits of using amazon RDS services includes and easy to administer model which can be controlled through AWS management console or AWS RDS command line tools. These services also come with wide variety of API call support to manage the services on Amazon cloud. Database can be scaled up very easily often without a downtime. The security is very efficiently handled by RDS which [20] lets you run your database instances in Amazon Virtual Private Cloud (Amazon VPC), which enables you to isolate your database instances and to connect to your existing IT infrastructure through an industry-standard encrypted IPsec VPN. Many Amazon RDS engine types offer encryption at rest and encryption in transit.

A data dump of the existing database was kept in a S3 storage bucket on AWS cloud. “Amazon Simple Storage Service (Amazon S3), provides developers and IT teams with secure, durable, highly-scalable cloud storage. Amazon S3 is easy to use object storage, with a simple web service interface to store and retrieve any amount of data from anywhere on the web” [23].

With the above configurations, ready, we used the AWS CLI as mentioned in section 4.2.1 to deploy the web application and its dependencies on the AWS EB

(Elastic Beanstalk) platform. Once the application was deployed successfully, we mapped the MySQL instance created using AWS RDS to this application. After this application was thoroughly tested by posting the transactions and verifying in data store on RDS.

4.3.2 Multi-Tier Web Application Manual Implementation on Heroku.

After the successful deployment on AWS EB environment, same web application was used and deployed on Heroku cloud. The deployment on Heroku was not straight forward. The necessary changes were made in configuration files to make the same code compatible to Heroku environment. Different cloud providers use different standards for deploying the applications. On Heroku, applications are deployed as containers. Also unlike AWS which has RDS service for creating databases instance, Heroku does it with the help of “Add-Ons” which are [24] “Tools and services for developing, extending, and operating your app”.

A Clear DB Heroku add-on was created and mapped to the web application. The data dump was imported and schema was created. Once the database was ready and mapped, we tested the application by posting some transactions and verified that the data is properly stored in the DB instance.

In the above two sections we saw how to create and deploy a multi-tier web application on Heroku and AWS clouds. Both the cloud platforms had different configurations and just with little standardization in the configuration, we could deploy the same application on both the cloud platforms. In the next chapter, we will see the architecture for our proposed POC application “Cloud Merge” for managing the migration autonomously from one cloud platform to another.

CHAPTER 5

FRAMEWORK DESCRIPTION

5.1 Introduction

In this chapter, we will discuss various components of proposed application Cloud Merge. The application tier has different components such as AWS cloud, Heroku cloud, a web client for user to interact with our heterogeneous cloud solution Cloud Merge. The application infrastructure also uses a GitHub private repository to centrally maintain the code to be deployed on both the test cloud platforms. The more detailed description on each component will be explained in coming sections of this chapter. Below figure 5.1 illustrates the Cloud Merge framework.

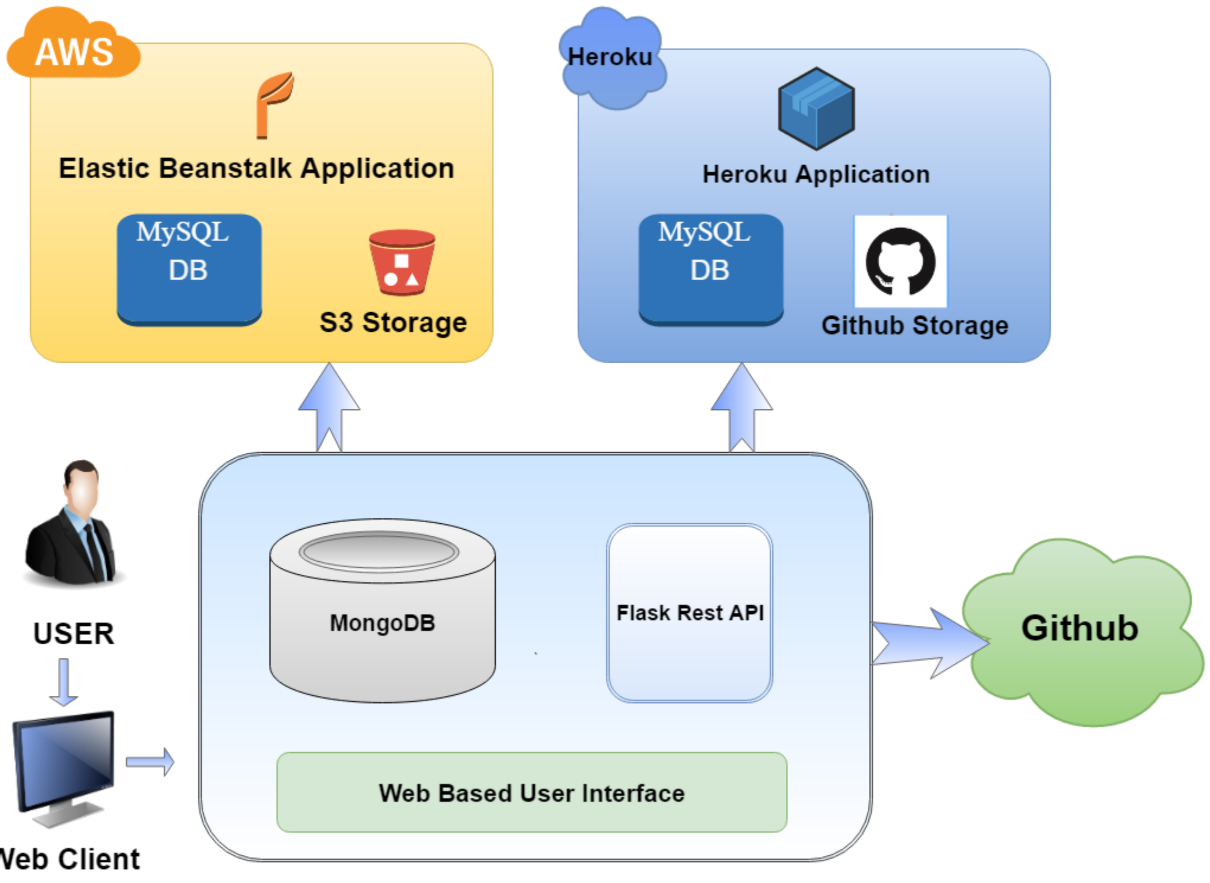


Figure 5.1 Cloud Merge Framework

5.2 Cloud Merge Web Architecture Description.

As shown in the above figure, the web application architecture of Cloud Merge application consists of several subcomponents such as a web interface to interact with the users. This comes with enriched options for users to choose between the cloud platforms such as AWS or Heroku to deploy the application. The interactive application allows user to communicate to AWS or Heroku cloud using the Rest API framework. The access key to authorize the users to gain access will be stored in a secured data store (Mongo DB) where all the keys for accessing AWS and Heroku cloud will be stored. We will discuss each component and its various features in detail as below.

The Web Interface for Cloud Merge is a multi-tier python application built using a flask framework. This application will enable users to interact with Heroku and AWS cloud and allow them to interact with cloud services such as Elastic Beanstalk, RDS, S3 storage buckets on AWS and Heroku Applications, and Add-on services on Heroku cloud. This application manages all the environment configurations for Heroku and AWS clouds through an interactive console and enables user to create new applications, databases on both the mentioned cloud platforms. Once the application is created on any of the cloud platform, user will be very easily able to manage their application and make modifications such as adding or dropping any resources, creating, dropping any database schema. The best feature and the purpose of the application will be to manage heterogenous clouds through the single console and user will also be able to move their application from one cloud to another very easily. The purpose of this paper was to study the various configurations required by any application to be inter-cloud compatible and to find a generic way so that same application can be run on any cloud platform without making an changes to the application code.

Features such as data migration from one cloud to another without having the need to worry about the downtime and managing separate resources for database and application migration, Cloud Merge will offer features to move application, database or both across multiple cloud.

Some of the business cases or scenarios that may be considered to better understand the approach:

1) An enterprise hosting a web application on AWS with data store on RDS SQL instance. With scenarios, such as AWS going down multiple times due to service not available, application may face a downtime which may be critical for transaction processing systems. In these cases, users, may want to port the application and its related services from one cloud provider to another
2) Enterprises use cloud services to host production, development and test environments. In these cases users, may want to host production environment on one cloud provider with auto scaling and load balancing but for test environment or dev environment, a small-scale infrastructure may work out. Instead of setting the environments separately user may deploy the same application across heterogeneous clouds with cheap resources or free tiers.
3) Increasing costs of running services on single cloud provider may motivate users to span application across multiple cloud providers.

To deal with the above problems we propose a solution Cloud Merge.

5.3 Cloud Merge Data Store Architecture.

Cloud platforms are accessable through REST API's using any programming language. The API's interact with the cloud platform using secret keys and private keys which authorize and authenticates the users. To ensure the security of the users, we store all the access and ssh keys of the user in a secure data store. The data store we choose for this purpose is Mongo DB. We want Mongo DB a data store because our data is unstructured, and we needed a database which could store and process the data with high availability. Other features that attracted us to use Mongo DB as our data store were :

- It is highly scalable and performance is good with unstructured data.
- In future, we may need to store different object types, and Mongo DB supports datatypes such as structured, semi-structured and polymorphic.
- The agile model of this database helps organizations to adapt changing requirements, scaling fast and reducing the marketing time for the application.
- It supports same functionalities as an RDBMS database and thus learning curve for developers is very less.
- Automatic data movement across different shards is called load balancing [21]. The balancer decides when to migrate the data and the destination

shard, so they are evenly distributed among all servers in the cluster. Each shard stores the data for a selected range of our collection according to a partition key.

- It offers secure authorization and authentication for users.

For all the above features Mongo DB serves as the best choice for our proof of concept application. To ensure the data security for users confidential data such as AWS and Heroku access keys and ssh keys, we are encrypting and storing keys using AES 256 bit encryption. The Block size used is 32. With this storing the keys became very secure and encryption process is very fast. Every time the keys are required, the web framework queries the database and decrypts the keys to access the AWS or Heroku cloud platform.

5.4 Cloud Merge Flask API Architecture.

REST API's are the communication interface between web client and backend services. In our application we are also using these API's to communicate and interact with AWS and Heroku cloud platforms. With this framework, we handled various operations such as GET, PUT, DELETE and POST. At some point in our application, we also used PATCH requests to update the application environment on AWS and Heroku cloud environments.

The framework of Cloud Merge is built using various libraries implementing several APIs. These APIs are framed to act as a middleware between the user interface and the backend. We are using Boto3 Python library to interact with AWS and Heroku.

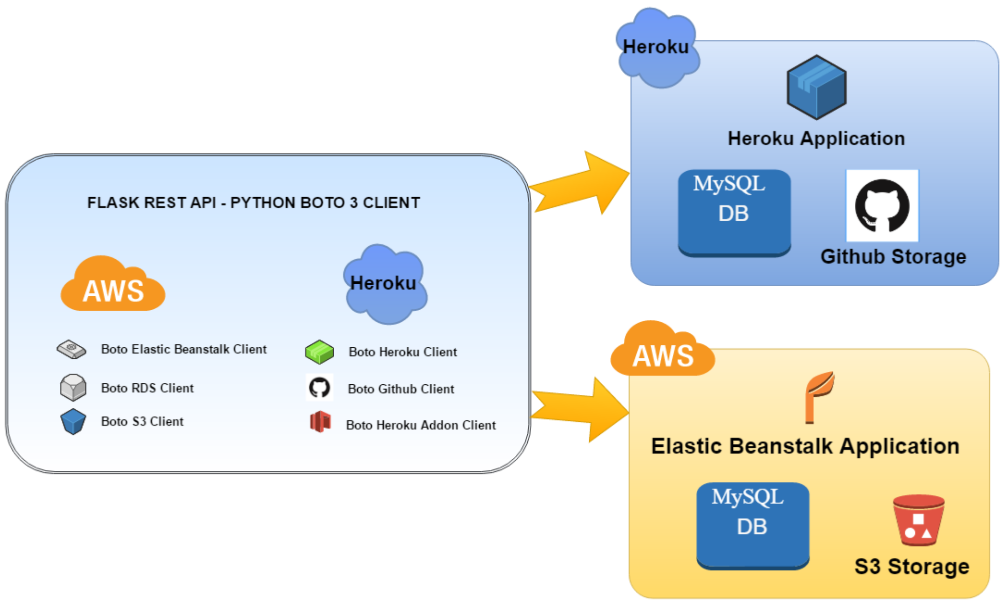


Figure 5.4 Cloud Merge Flask API Architecture

As shown in the above figure Flask Rest API framework consists of different clients to connect to several services on AWS and Heroku. For managing the AWS cloud through Cloud Merge, we are using Elastic Beanstalk Boto client. It gives ample of features to provision and orchestrate cloud applications. Some of the features offered by EBS client used in our application includes

- Create Application

- Create Application Version
- List EBS Applications
- Delete Application
- Create Environment
- Update Environment
- List EBS Environments
- Delete Environment

For storage, we used Boto S3 client, using which we were able to manage our application storage on AWS. When any application is deployed on AWS using EBS, internally the application is stored in an S3 bucket, and all subsequent updates and version are stored in the same bucket. Every time any update is made to the application, a new version of the application is created automatically. The S3 client services used in our application are as below:

- Create Bucket
- List Buckets
- Put Objects
- List Objects
- Delete Buckets
- Delete Objects

Also, later in this paper, we will discuss our migration approach. For migrating the database, we will be using AWS S3 storage to save the data dumps from the RDS instance. For managing the RDS instance through our application, we used Boto3 RDS client. Some of the features offered by RDS client used in our application are listed below:

- Create DB Instance
- List DB instance
- Delete DB Instance

Apart from the services mentioned above we wrote the logic to migrate a DB from data dump to AWS RDS instance and to migrate the DB from AWS RDS instance to Heroku cloud.

5.5 Cloud Merge User, Client and GitHub Repository Description.

In figure 5.1, the framework shows a user, a web client and the application framework interacting with the GitHub repository. This section will give the details and purpose of the mentioned components in the framework. A user will typically be a Cloud Administrator or an enterprise user who wants to manage the cloud infrastructure through our proposed application. The user will interact with the application using a web browser, which acts as a client between the application framework and user. The client gives interactive forms to the user to choose among different options to manage the application on AWS and Heroku cloud. With this client, the users will simply be able to migrate their application from one cloud to another. The user will also be able to create new data stores, migrate the data from the old data store to the new data store and deployes the application across multiple clouds, keeping the application on one cloud platform and database on the other cloud platform.

The GitHub repository in this architecture plays an important role in storing code of the web application to be deployed on AWS/ Heroku cloud platforms. Proposed heterogeneous cloud application will fetch the code from GitHub repository and deploy on the chosen platform, ranging from Heroku or AWS to any other supported cloud provider in future.

5.6 AWS Architecture Description.

The below figure illustrates the AWS architecture diagram of the proposed heterogeneous cloud application. The major components used in this framework are Elastic Beanstalk, RDS (My SQL database) and S3 storage.

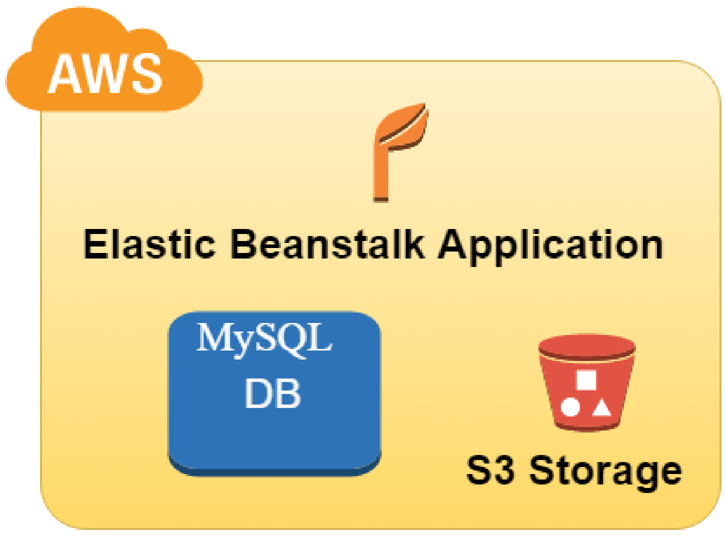


Figure 5.6 Cloud Merge AWS Architecture Description

The AWS EB service [16] is used to automatically resolve the dependencies and deploying the multi-tier web application on Amazon cloud. The application code is fetched from GitHub repository. The S3 Storage [23] is used to store the database dump.

While migrating, data dumps are accessed from these storage units and new database objects are created in AWS environment using the RDS [19] service. A MySQL instance is created and objects are compiled in new instance.

The workflow for migrating an application starts by building an application in AWS EB environment using the code from GitHub repository. Once done, an AWS EB environment is created to host the application. Now the database is created using RDS service. Once the MySQL instance is ready and end point information is available, compile all the objects stored as data dump, from S3 bucket, in the new instance. This architecture supports deployments from the local environment to AWS cloud and from Heroku cloud to the AWS cloud. Different implementation strategies (Homogenous and Heterogenous) will be dussed in coming sections.

5.7 Heroku Architecture Description.

The figure 5.7 illustrates the Heroku architecture diagram of Cloud Merge. The major components used are Heroku runtime for deploying the application, GitHub Storage to access the web application code and Clear DB (MySQL) instance as Add-on service for the data storage.

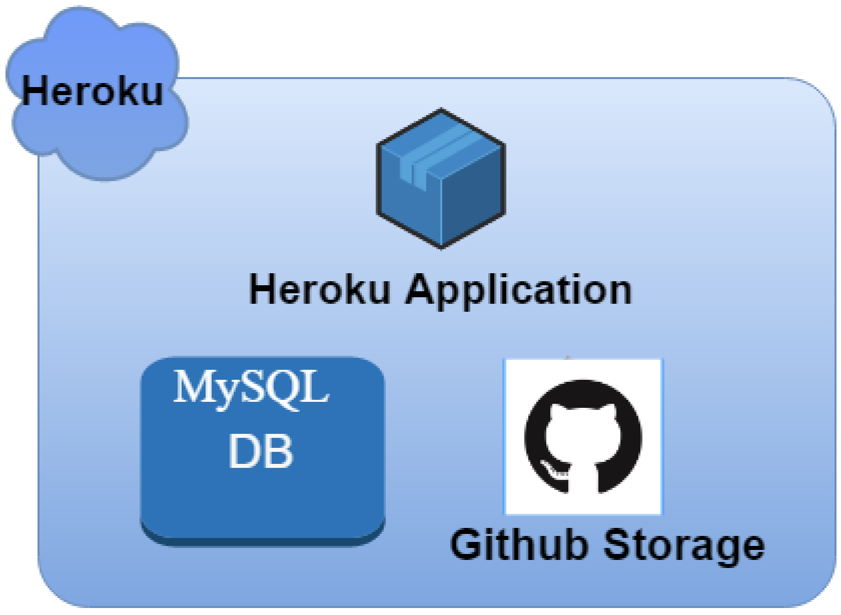


Figure 5.7 Cloud Merge Heroku Architecture Description

As seen in the above figure 5.7 the Heroku platform was used for deploying the code which was fetched from the same GitHub repository we used for deploying the AWS application in the previous section. Once the code was successfully deployed, we used the Heroku Add-on service to create a Clear DB instance mapped to the application. Once the endpoint information is available, the Cloud Merge framework would use the data dump from AWS S3 bucket to compile all the database objects and migrate all the data into the new instance.

The key for migrating the application here from one environment to another is coming up with a generic web application which is supported across both the cloud platforms and further can be extended to new cloud platforms in future. We standardized the environment variables used by the application and instead of using the platform-specific environment variables, injected those variables in cloud environments with standardized names. This way application could seamlessly access the endpoint information irrespective of what cloud platform it was deployed on. The next chapter will discuss the implementation details of the Cloud Merge framework on both cloud platforms and different deployment strategies that can be used to deploy a web application spread across multiple cloud platforms.

CHAPTER 6

FRAMEWORK IMPLEMENTATION

6.1. Introduction

In this chapter we will discuss about the implementation of the architecture discussed in chapter 5. The Cloud Merge framework can be either hosted on local environment or on public cloud. For our experiments we hosted the architecture on both local environment and on AWS EC2 instance. The flask application deployed on a Windows 2012 Server on AWS. A Mongo DB was installed locally on the server to store the secret keys to ensure the security of the data. The maintenance of keys is a one-time activity and can be read through configuration files from the project repository, but maintaining in files may expose the keys in case of any security breach so encrypting the keys using AES 256-bit encryption ensured the keys are safe in a data store.

Using the flask framework, our web application gives user the option to manage Heroku and AWS cloud applications and their resources. User can choose between ample of options from deployment strategies discussed in coming sections.

6.2. Homogenous AWS Deployment

In this deployment, application is hosted of AWS Elastic Beanstalk and database is also hosted on Amazon RDS.

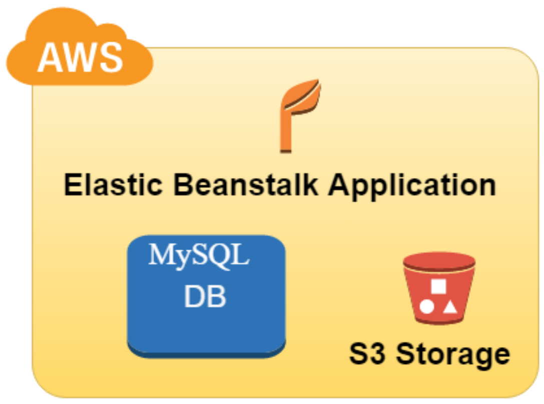


Figure 6.2 Homogenous AWS Application Deployment

6.3 Heterogenous AWS Deployment-Cross Cloud

In this deployment application is hosted in AWS Elastic Beanstalk environment and database is hosted on Heroku cloud as an add-on service. This way we will have a heterogeneous cloud deployment with application components spread across different cloud providers.

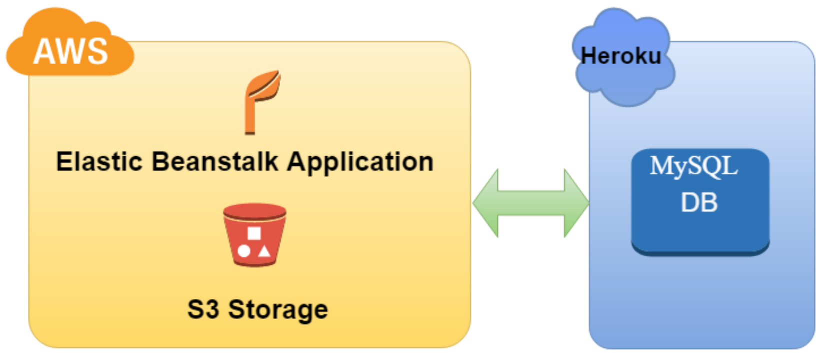


Figure 6.3 Heterogenous AWS Deployment-Cross Cloud

6.4. Homogenous Heroku Deployment

In this deployment, application is hosted on Heroku platform and database is also hosted on Heroku cloud using Clear DB instance as add-on service.

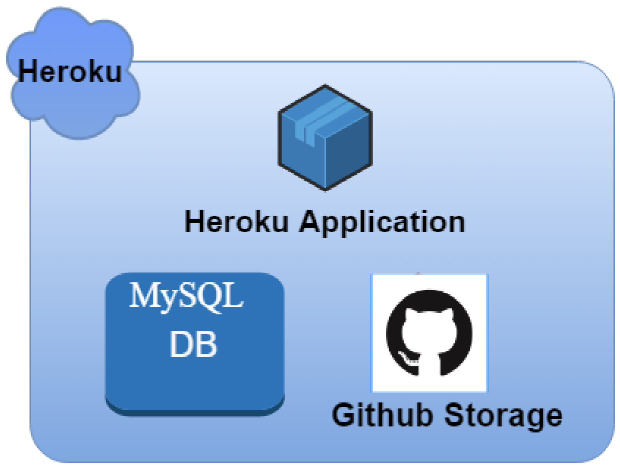


Figure 6.4 Homogenous Heroku Deployment

6.5 Heterogenous Heroku Deployment Cross Cloud

In this deployment application is hosted in Heroku environment and database is hosted on AWS cloud using RDS MySQL instance. This way we will have a heterogeneous cloud deployment with application components spread across different cloud providers.

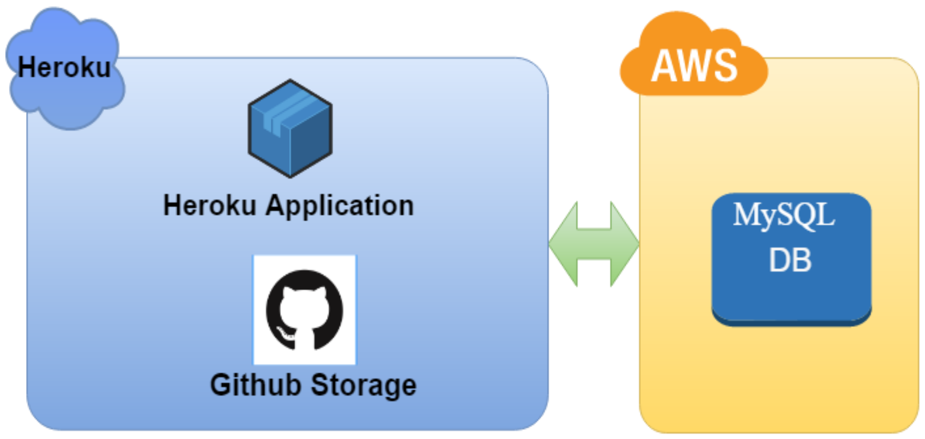


Figure 6.5 Heterogenous Heroku Deployment -Cross Cloud

CHAPTER 7

EXPERIMENTS AND RESULTS

7.1 Introduction

In this chapter the experiments that aided in structuring the proposed framework are described. The details of the experiments are discussed in each of the below sections.

The experiment area includes single tier web application deployment and tear down on both Heroku and AWS clouds. These set of experiments were repeated on both AWS and Heroku platforms for capturing deployment and tear down times of multi-tier web application which included a flask framework middle tier and a My SQL database to act as a data store. Further extending the experiment sets, we deployed our application across clouds in a heterogeneous architecture, keeping the application middleware on one cloud platform and creating the data store of the application on other cloud platform. Experiments were conducted for both Heroku and AWS cross-cloud deployments.

After the deployment and tear down, performance analysis of the application was done on both Heroku and AWS platforms. 100, 1000, 10000, 100000 DML operations were done (Insert, Update, Select) in both single and heterogeneous cloud deployments and very interesting results were analyzed. The next sections will discuss each type of experiment in detail.

7.2 Environment Details for The Experiments

The experiments were conducted on the below configurations:

7.2.1 Web Client

The Web client is developed using Python Flask application and used MySQL drivers to connect to RDS and Heroku My SQL instances. The web framework is hosted on a local windows machine. The version specifications and the environment details are as below:

- Python 2.7
- Flask 0.11
- Intel Core i7, 1TB HDD, 16 GB RAM, Windows 10 Machine @2.50 GHz

7.2.2 AWS Configurations

The AWS environment details are as below:

- Python 2.7
- Flask 0.11
- Amazon Beanstalk
- t2.micro instance, Windows 2012 R2 Server with Intel Xeon(R) CPU@2.4 G Hz, 1GB RAM
- db.t2.micro, 1V CPU, 1GB RAM My SQL Instance on RDS

7.2.3 Heroku Cloud Configurations

The Heroku environment details are as below:

- Python 2.7
- Flask 0.11
- Dyno Type: Free
- 512 Mb RAM, 1X CPU, 1x*4x Compute
- Ignite tier Clear DB instance with 5MB database size, 10 Connections

7.3 Single Tier Web Application Deployment on Heroku and AWS.

This section describes the single tier web application deployment experiments on Heroku and AWS cloud platforms. The purpose of these experiments was to analyze the deployment time of a single tier flask application on the above-mentioned cloud platforms.

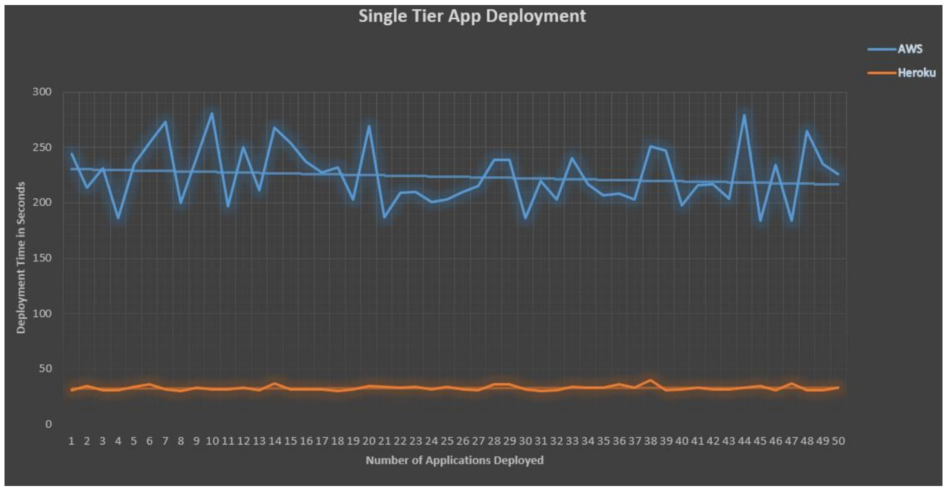


Figure 7.3.1 Single Tier Web Application Deployment on Heroku and AWS

For these experiments, we deployed a single tier Flask web application with no data storage. Through our POC web application Cloud Merge, we deployed 50 flask web applications on AWS Elastic Beanstalk environment and noted the deployment times. Fig 7.3.1 shows the web application deployment times on Heroku and AWS cloud platform for 50 applications. Figure 7.3.2 shows the average deployment time of a single tier web application on Heroku and AWS cloud.

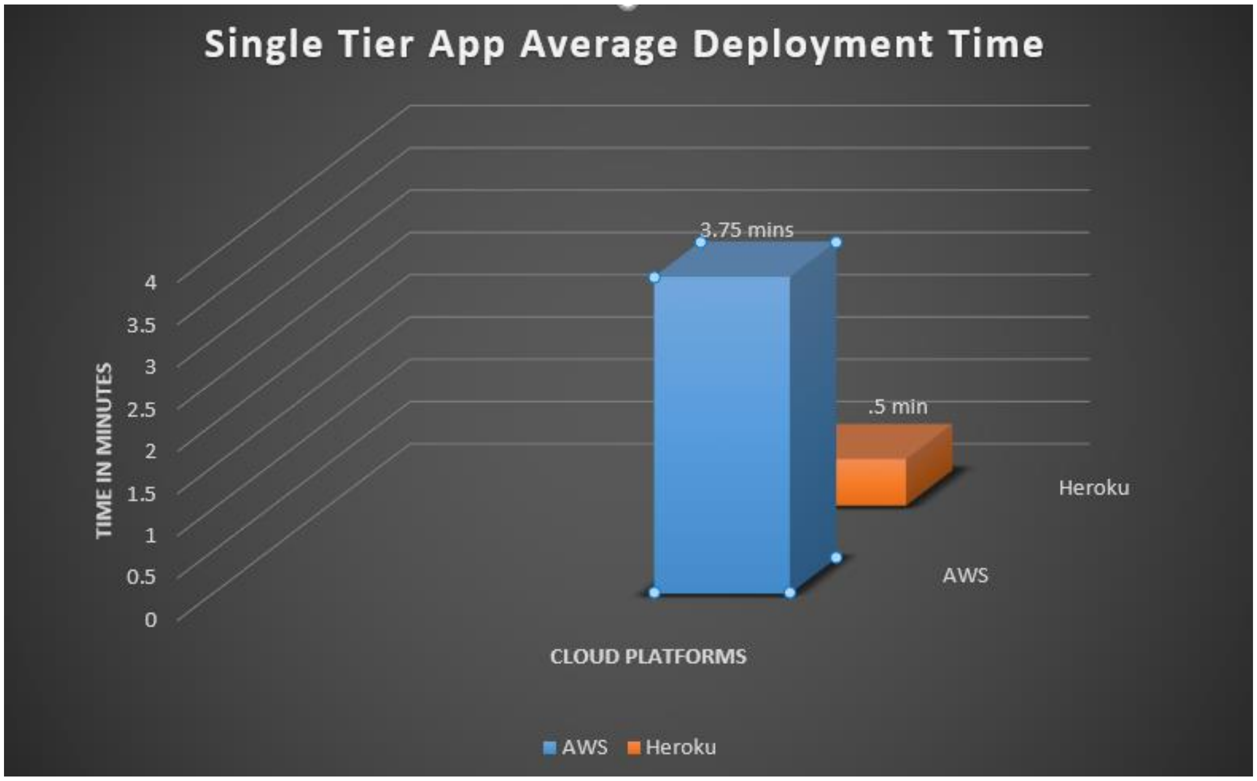


Figure 7.3.2 Single Tier Web Application Average Deployment Times on Heroku and AWS

7.4 Single Tier Web Application Tear Down Experiments on Heroku and AWS.

This section describes the single tire web application tear down experiments on Heroku and AWS cloud platforms. The purpose of these experiments was to analyze the teardown time of a single tier flask application on the above-mentioned cloud platforms.

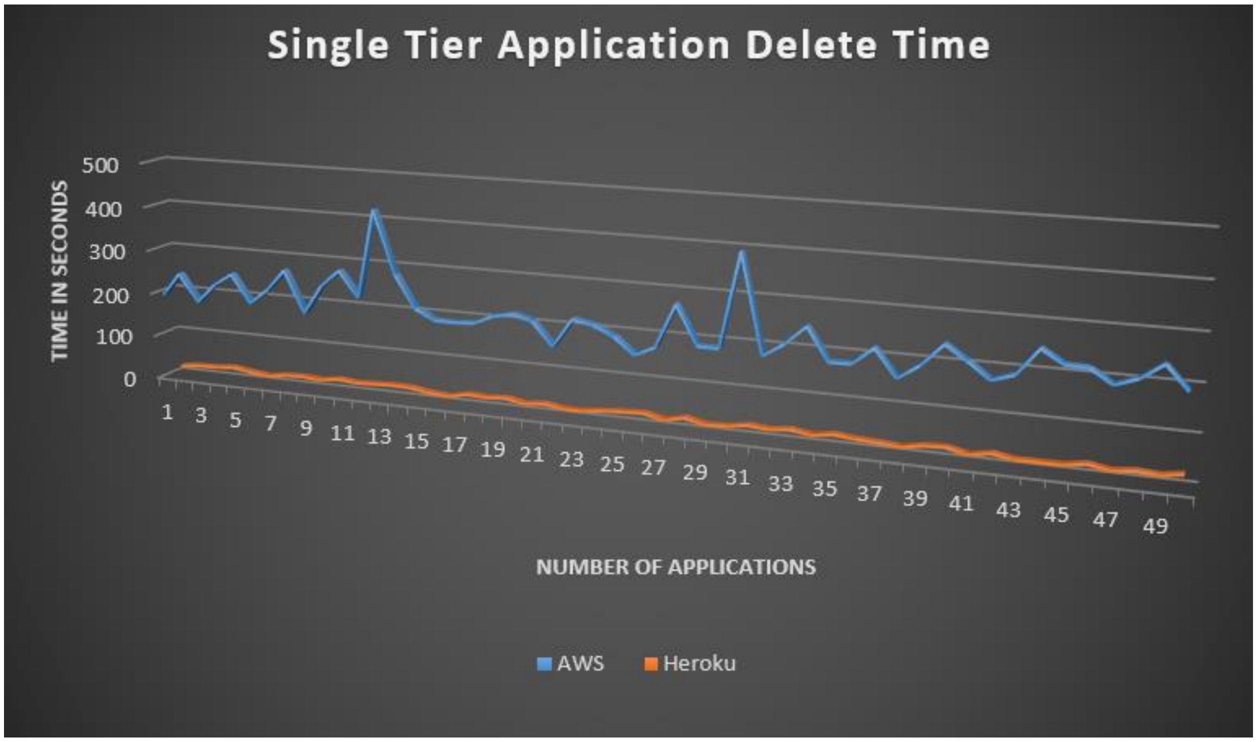


Figure 7.4.1 Single Tier Web Application Tear Down Experiments on Heroku and AWS

For these experiments, each single tier web application created in section 7.3 was first tested for proper functioning on AWS and Heroku clouds after deployment and then deleted. The teardown time for each application was noted on both the platforms. Fig 7.1.1 shows the web application tear down times on Heroku and AWS cloud platform for 50 applications. Figure 7.4.2 shows the average tear down time of a single tier web application on Heroku and AWS cloud

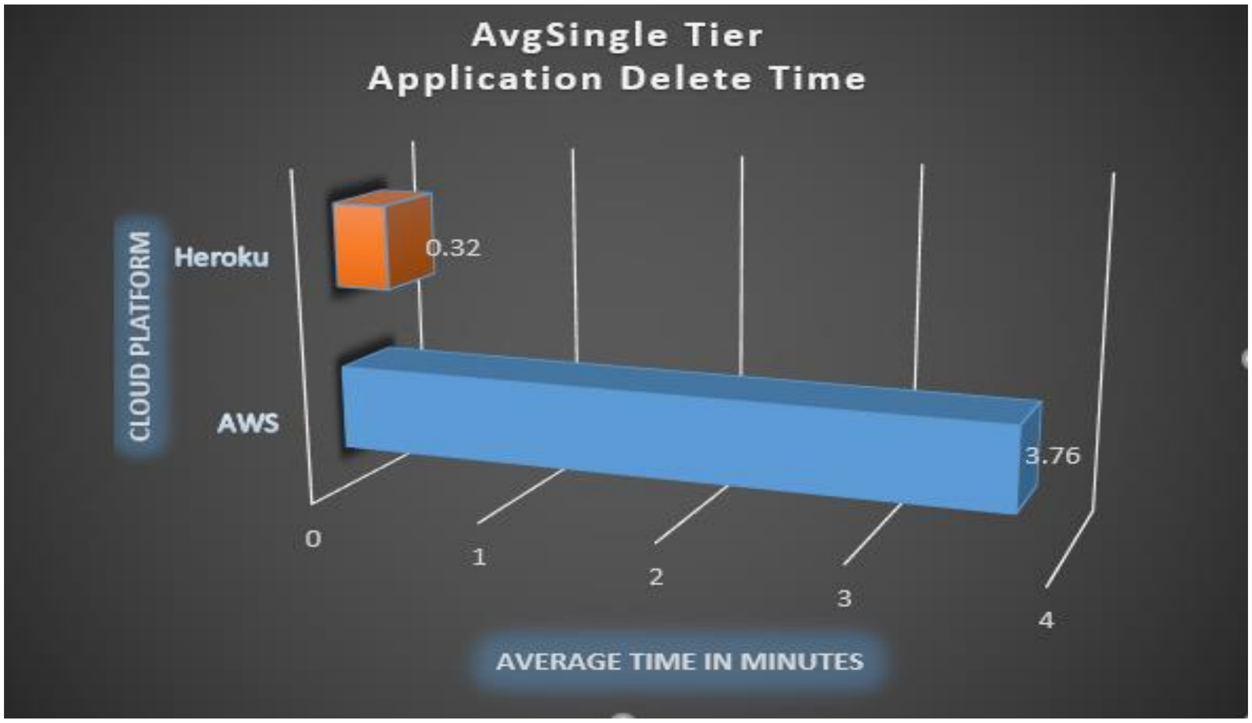


Figure 7.4.2 Single Tier Web Application Average Tear Down Time on Heroku and AWS

7.5 Multi-Tier Web Application Deployment Experiments on Heroku and AWS.

This section describes the multi-tier web application deployment experiments on Heroku and AWS cloud platforms. The purpose of these experiments was to analyze the deployment time of a multi-tier flask application on the above-mentioned cloud platforms.

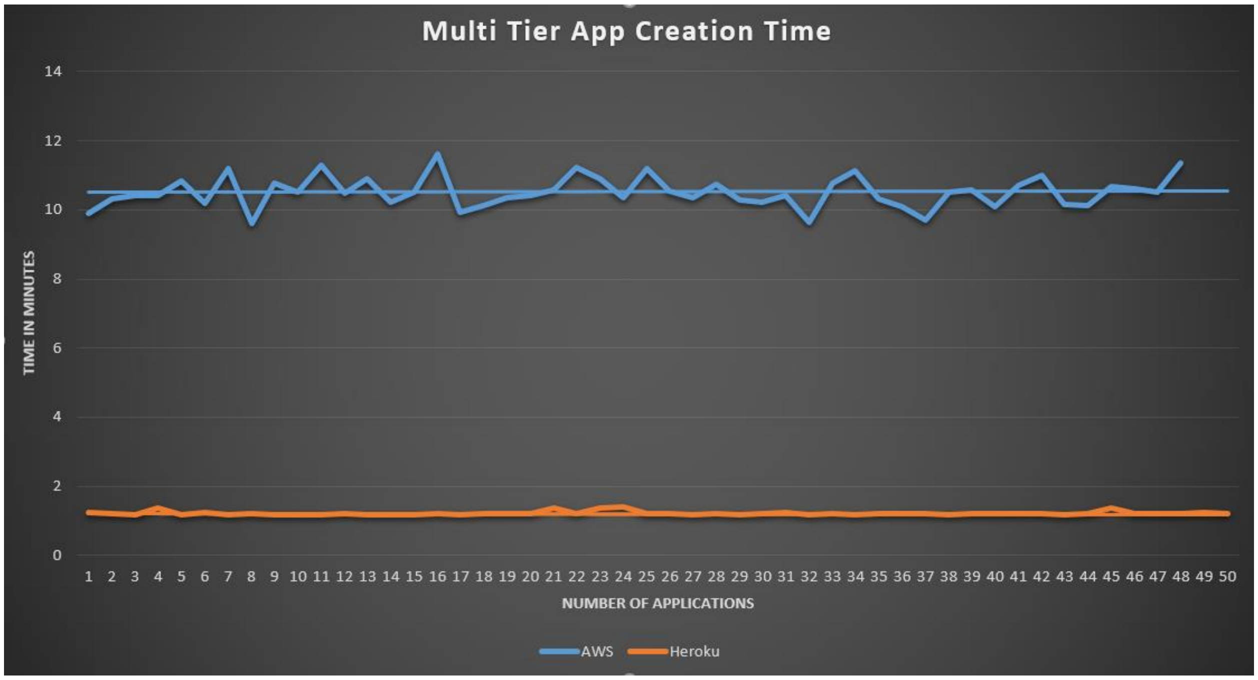


Figure 7.5.1 Multi-Tier Web Application Deployment Times on Heroku and AWS

For these experiments, we deployed a multi-tier Flask web application with a data storage. The configurations for the database and the instance used on AWS and Heroku are mentioned in section 7.2.2 and 7.2.3. These experiments were conducted keeping the application and data store on the same cloud provider as shown in figure 6.2. Through our POC web application we deployed 50 flask web applications on AWS Elastic Beanstalk environment and noted the deployment times. Fig 7.5.1 shows the web application deployment on Heroku and AWS cloud platform for 50 applications. Figure 7.5.2 shows the average deployment time of a multi-tier web application on Heroku and AWS cloud.

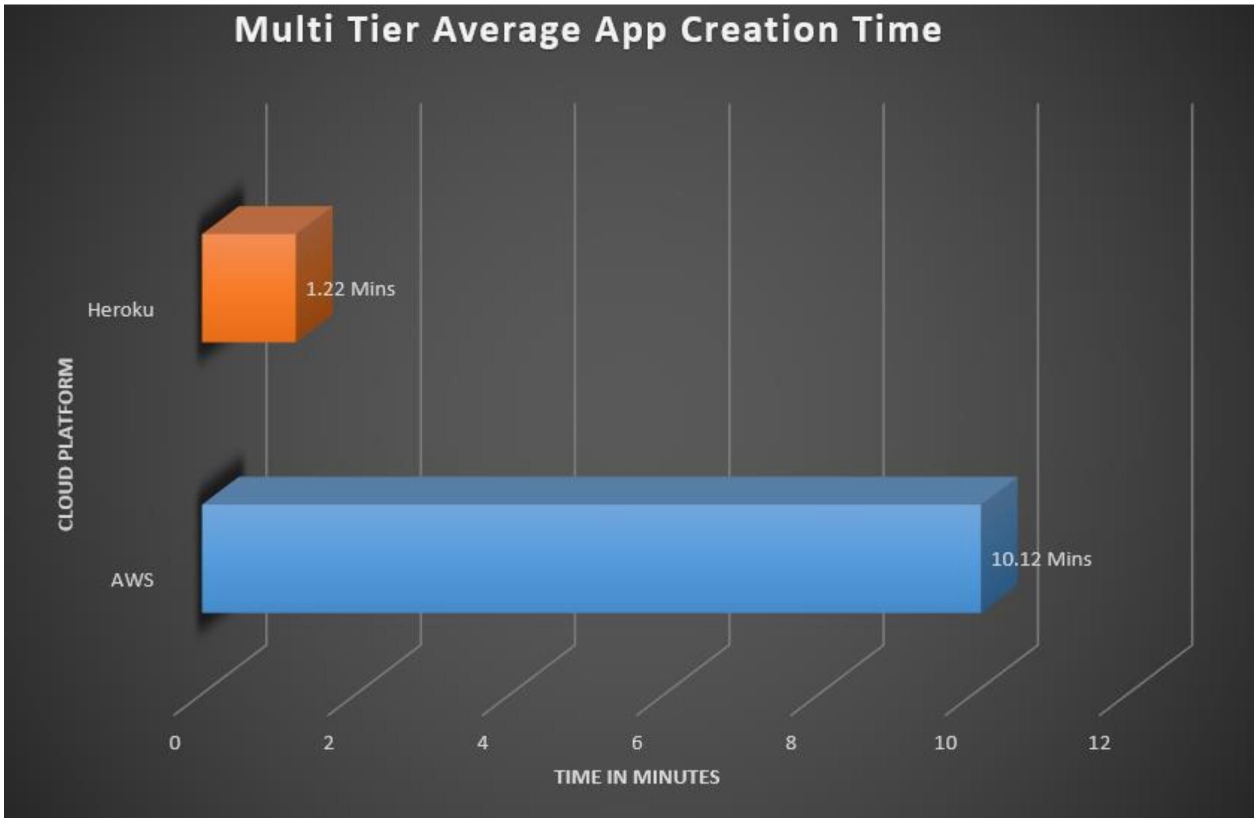


Figure 7.5.2 Multi-Tier Web Application Average Deployment Times on Heroku and AWS

7.6 Multi-Tier Web Application Tear Down Experiments on Heroku and AWS.

This section describes the multi-tier web application tear down experiments on Heroku and AWS cloud platforms. The purpose of these experiments was to analyze the teardown time of a multi-tier flask application on the above-mentioned cloud platforms.

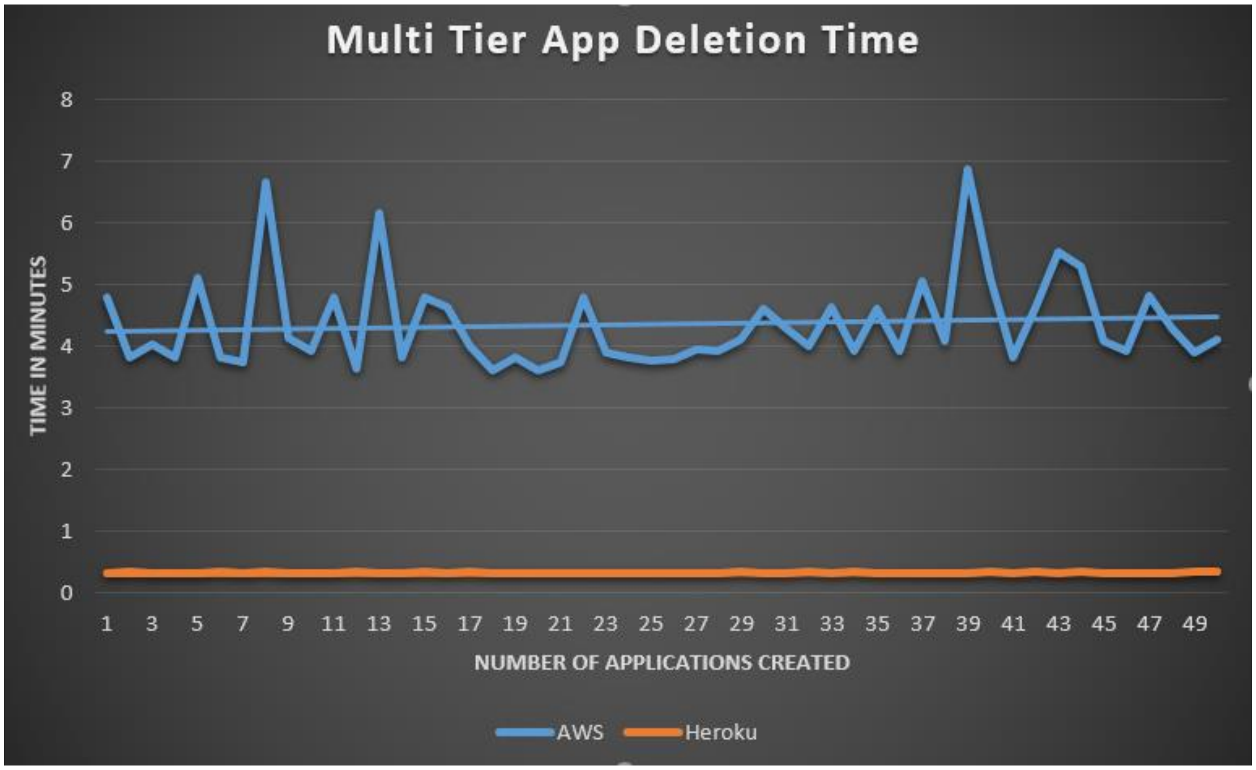


Figure 7.6.1 Multi-Tier Web Application Tear Down Experiments on Heroku and AWS

For these experiments, each multi-tier web application created in the previous section was first tested for proper functioning on AWS and Heroku clouds after deployment and then deleted. The teardown time for each application was noted on both the platforms. Fig [7.6.1] shows the web application tear down times on Heroku and AWS cloud platform for 50 applications. Figure 7.6.2 shows the average tear down time of a multi-tier web application on Heroku and AWS cloud.

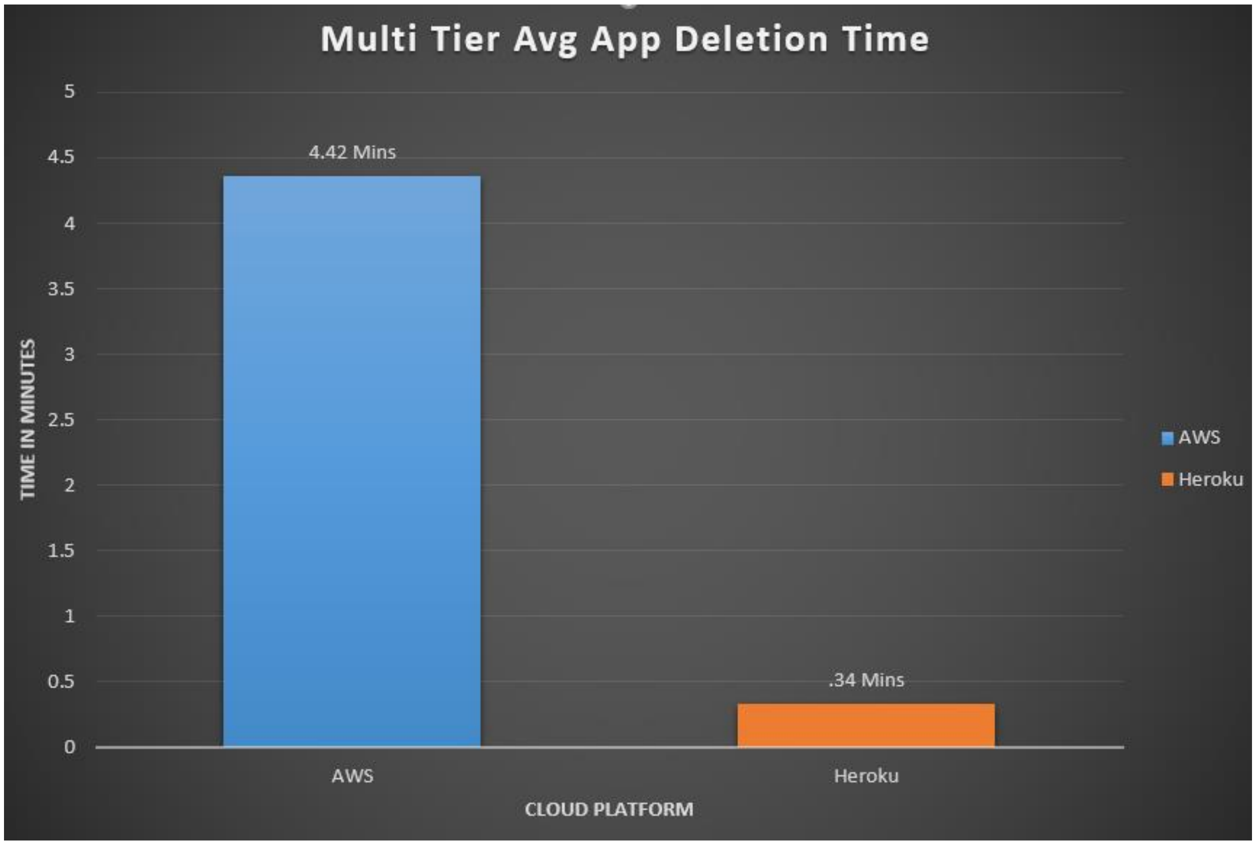


Figure 7.6.2 Multi-Tier Web Application Average Tear Down Time on Heroku and AWS

7.7 Cross Cloud Multi-Tier Web Application Deployment Experiments.

In this section, we will describe the “Cross Cloud” multi-tier web application deployment. By cross-cloud we intend to say that unlike the previous deployments where application and data store were on a same cloud platform, we will make the deployments heterogenous i.e. The application will be hosted on AWS and database on Heroku or application will be hosted on Heroku and database on AWS cloud. The configurations of the environment remain the same as described in section 6.3

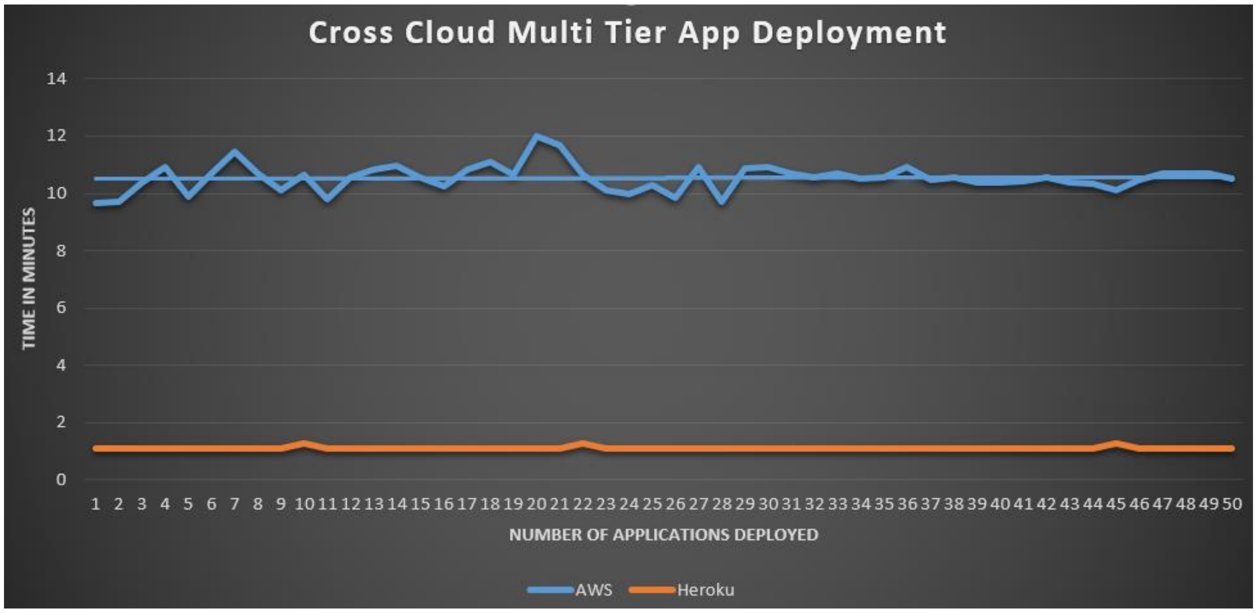


Figure 7.7.1 Cross Cloud Multi-Tier Web Application Deployment Experiments

For these experiments, we deployed a multi-tier Flask web application with a data storage. The configurations for the database and the instance used on AWS and

Heroku are mentioned in section 7.2.2 and 7.2.3. These experiments were conducted keeping the application on one cloud platform and data store on the other cloud provider as shown in figure 6.3 and figure 6.5. Through our POC web application, we deployed 50 flask web applications on AWS Elastic Beanstalk environment and noted the deployment times. Fig 7.2.1 shows the web application deployment on Heroku and AWS cloud platform for 50 applications. Figure 7.2.2 shows the average deployment time of a multi-tier web application on Heroku and AWS cloud.

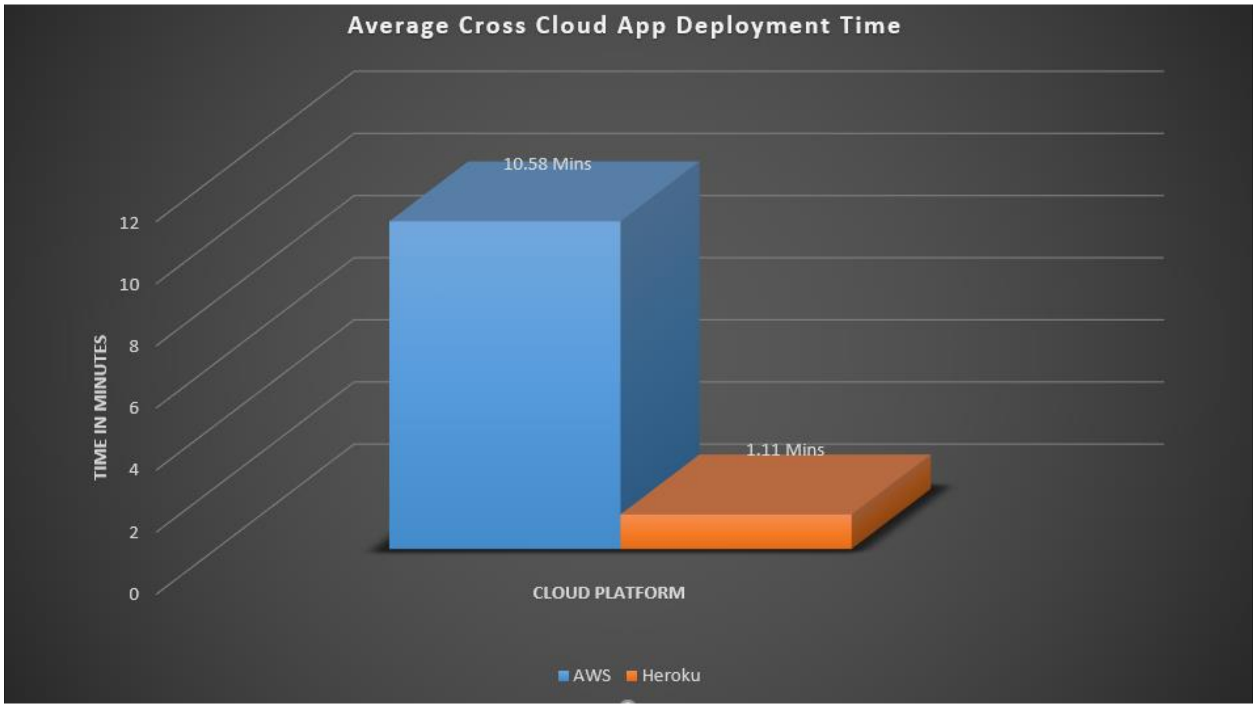


Figure 7.7.2 Cross Cloud Multi-Tier Web Application Average Deployment Times.

7.8 Cross Cloud Multi-Tier Web Application Tear Down Experiments.

This section describes the cross-cloud multi-tier web application tear down experiments on Heroku and AWS cloud platforms. The purpose of these experiments was to analyze the teardown time of a multi-tier flask application on the above-mentioned cloud platforms.

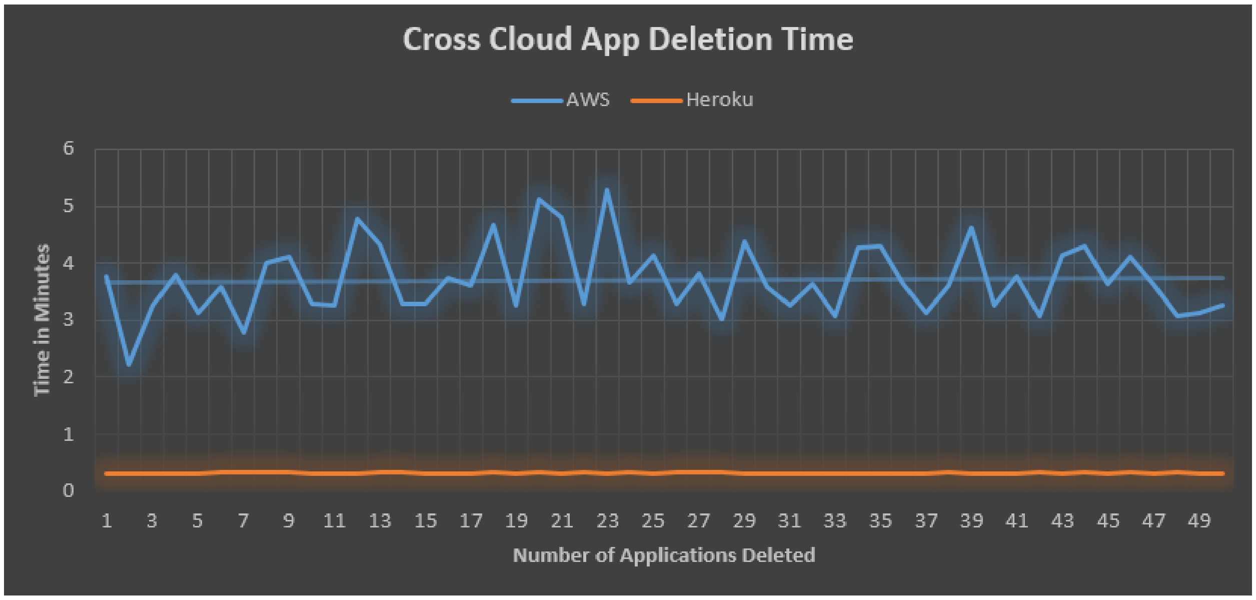


Figure 7.8.1 Cloud Multi-Tier Web Application Tear Down Experiments.

For these experiments, each cross-cloud multi-tier web application created in the previous section was first tested for proper functioning on AWS and Heroku clouds after deployment by posting some DML operations through deployed application and then terminating the application and its dependencies. The teardown time for each application was noted on both the platforms. Fig 7.8.1 shows the web application tear down times on Heroku and AWS cloud platform for 50 applications. Figure 7.8.2 shows the average tear down time of a multi-tier web application on Heroku and AWS cloud.

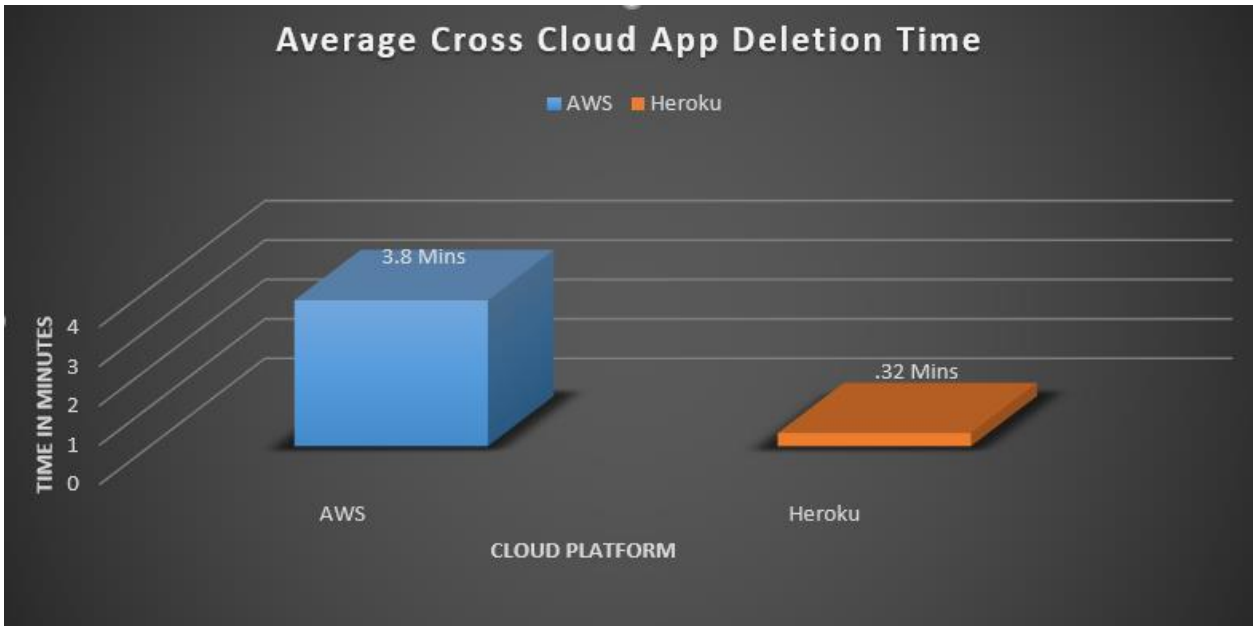


Figure 7.8.2 Cross Cloud Multi-Tier Web Application Average Tear Down Time.

7.9 Select Query Experiments on Homogenous Cloud.

This section describes the set of experiments done to check the performance of the web application deployed in the previous sections on AWS and Heroku cloud platforms. In this experiment, the application was first deployed on AWS cloud platform with data store also on the AWS cloud. The environment configurations are shown in section 7.2.2 and 7.2.3. After successful deployment, the application was stress tested with 100, 1000, 10000, 100000 select queries in separate batches to analyze the performance of the application and to analyze the database request response times. The same procedure was repeated by deploying the same application on Heroku cloud and keeping the database on Heroku cloud.

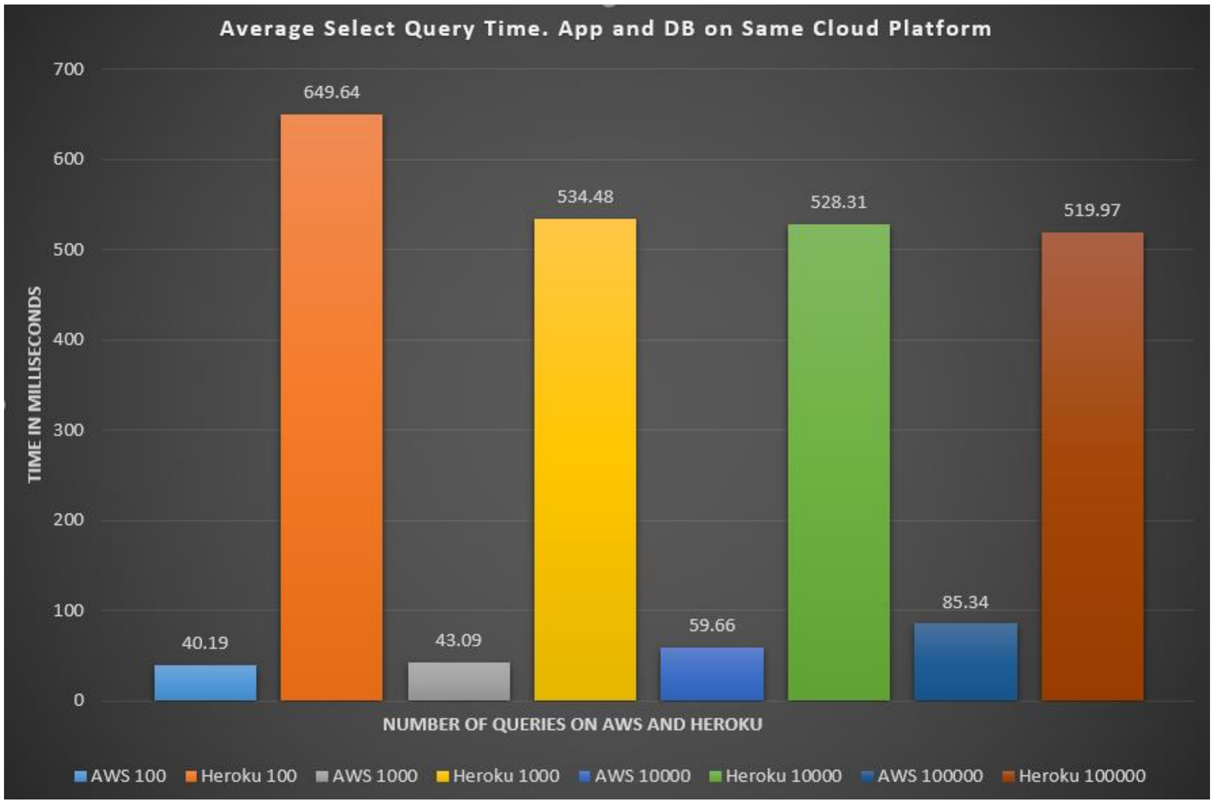


Figure 7.9 Select Query Experiments on Homogenous Cloud.

As shown in the figure 7.9, the respective queries were sent and response time was noted. We are showing the average query fetch time for each type of query to better illustrate the analysis. The response time of AWS application varied from 40.10 milliseconds to 85.34 milliseconds. The difference could be clearly noted as the number of requests to the database were increased, the query response time started increasing. In the case of Heroku, the initial 100 queries took more time , the reason being the application goes to sleep after inactivity and that time caused a delay in query processing. After that, the query response time was almost constant in the case of Heroku based Database varying between 534.48 milliseconds to 519.97 milliseconds.

With this experiment, we saw a difference between the response times from AWS based database and Heroku based database for the same application which indicates AWS RDS instance is way better than Heroku add-ons under the free tier configurations we used for both the platforms.

7.10 Insert Query Experiments on Homogenous Cloud.

This section describes the set of experiments done to check the performance of the web application deployed in the previous sections. In this experiment, the application was first deployed on AWS cloud platform with data store also on the AWS cloud. The environment configurations are shown in section 7.2.2 and 7.2.3. After successful deployment, the application was stress tested with 100, 1000, 10000, 100000 INSERT queries in separate batches to analyze the performance of the application and to analyze the database request response times. The same procedure was repeated by deploying the same application on Heroku cloud and keeping the database on Heroku cloud.

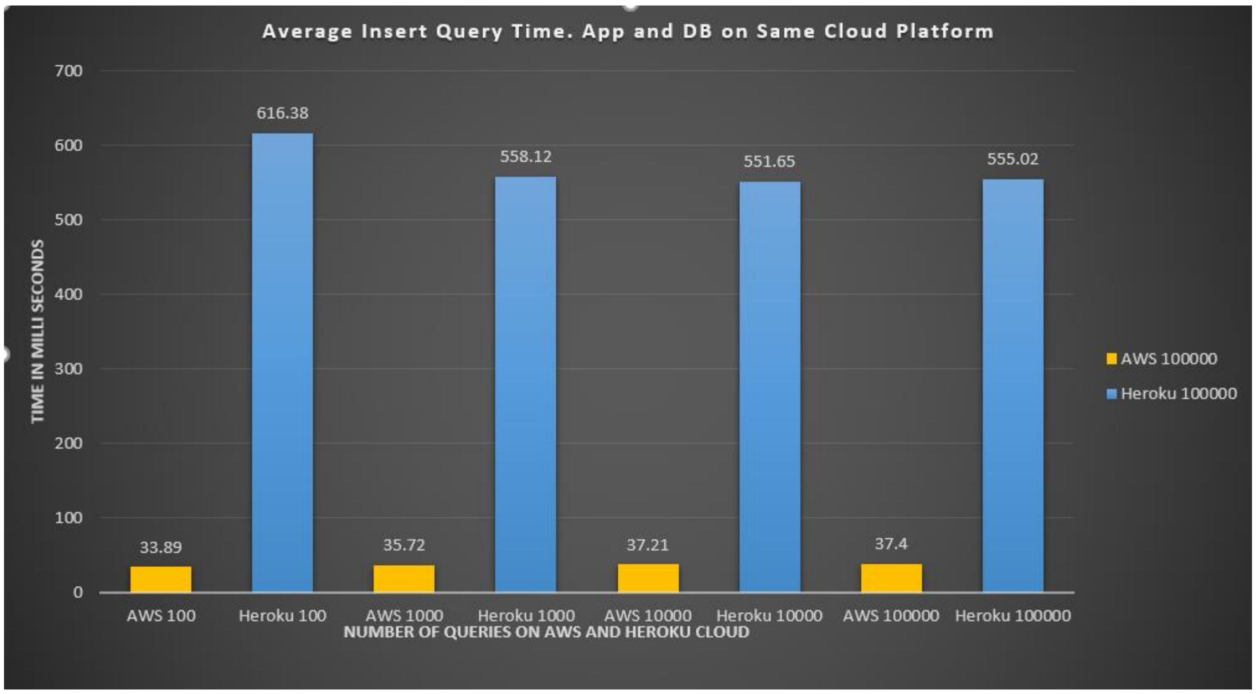


Figure 7.10 INSERT Query Experiments on Homogenous Cloud.

As shown in the above figure 7.10, the respective queries were sent and response time was noted. We are showing the average query response time for each type of query to better illustrate the analysis. The response time of AWS application varied from 33.89 milliseconds to 37.4 milliseconds. The response time remained almost constant for any number of requests on RDS My SQL instance. In the case of Heroku, the initial 100 queries took slightly more time , the reason being the application goes to sleep after inactivity and that time caused a delay in query processing. After that, the query response time for INSERT operations was almost constant in the case of Heroku based database instance varying between 616.38 milliseconds to 555.02 milliseconds.

With this experiment, we saw a difference between the response times for Insert operations from AWS and Heroku based databases for the same application which indicates AWS RDS instance is way better than Heroku add-ons under the free tier configurations we used for both the platforms.

7.11 Update Query Experiments on Homogenous Cloud.

This section describes the set of experiments done to check the performance of the web application deployed in the previous sections. In this experiment, the application was first deployed on AWS cloud platform with data store also on the AWS cloud. The environment configurations are shown in section 7.2.2 and 7.2.3. After successful deployment, the application was stress tested with 100, 1000, 10000, 100000 UPDATE queries in separate batches to analyze the performance of the application and to analyze the database request response times. The same procedure was repeated by deploying the same application on Heroku cloud and keeping the database on Heroku cloud.

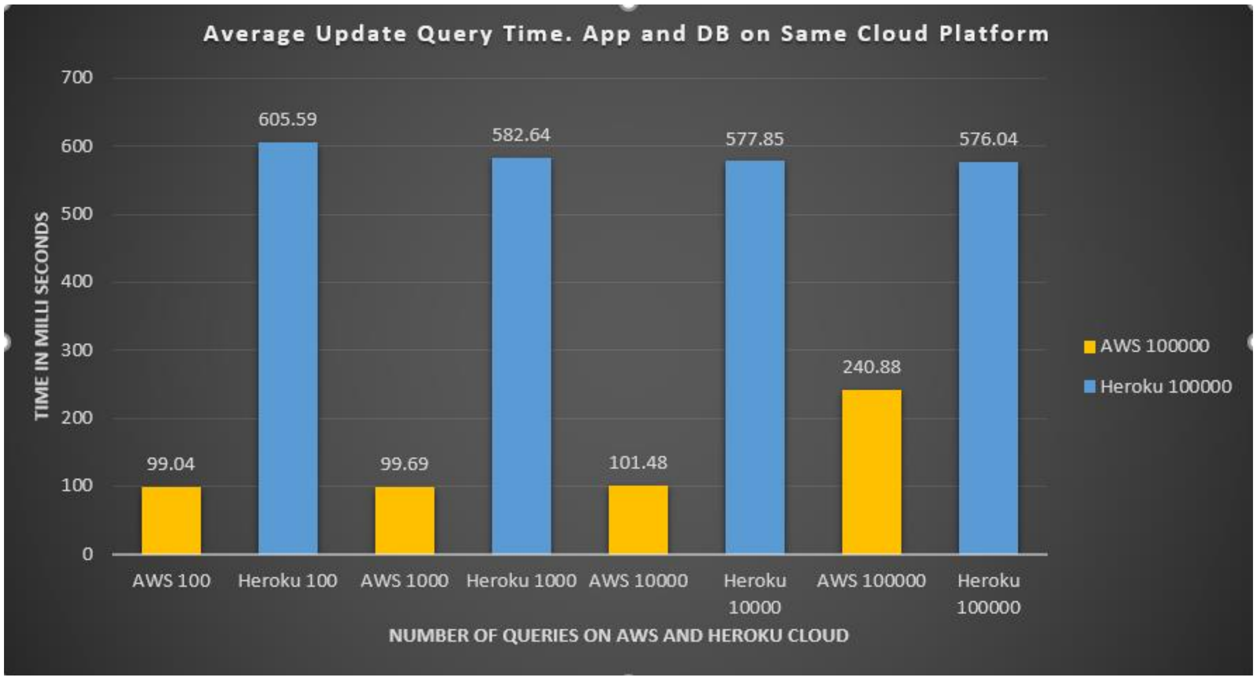


Figure 7.11 UPDATE Query Experiments on Homogenous Cloud.

As shown in the above figure 7.11, the respective queries were sent and response time was noted. We are showing the average query response time for each type of query to better illustrate the analysis. The response time for UPDATE operations of AWS application varied from 99.08 milliseconds to 240.88 milliseconds. The response time remained almost constant for till 10,000 update requests on RDS My SQL instance. After that when the load on database was increased the updates caused a tremendous increase in response time averaging 240.88 milliseconds which is almost double than the normal response time.

In the case of Heroku, the query response time for UPDATE operations was almost constant in the case of Heroku based Database instance varying between 605.59 milliseconds to 576.04 milliseconds. With this experiment, we saw a difference between the response times for Update operations from AWS and Heroku based databases for the same application which indicates AWS RDS instance is way better than Heroku add-ons under the free tier configurations we used for both the platforms.

7.12 Select Query Experiments on Heterogenous Cloud Deployment.

This section describes the set of experiments done to check the performance of the web application deployed in the previous sections. In this experiment, the application was first deployed on AWS cloud platform with data store also on Heroku cloud. The environment configurations are shown in section 7.2.2 and 7.2.3. After successful deployment, the application was stress tested with 100, 1000, 10000, 100000 select queries in separate batches to analyze the performance of the application and to analyze the database request response times. The same procedure was repeated by deploying the same application on Heroku cloud and keeping the database on AWS cloud.

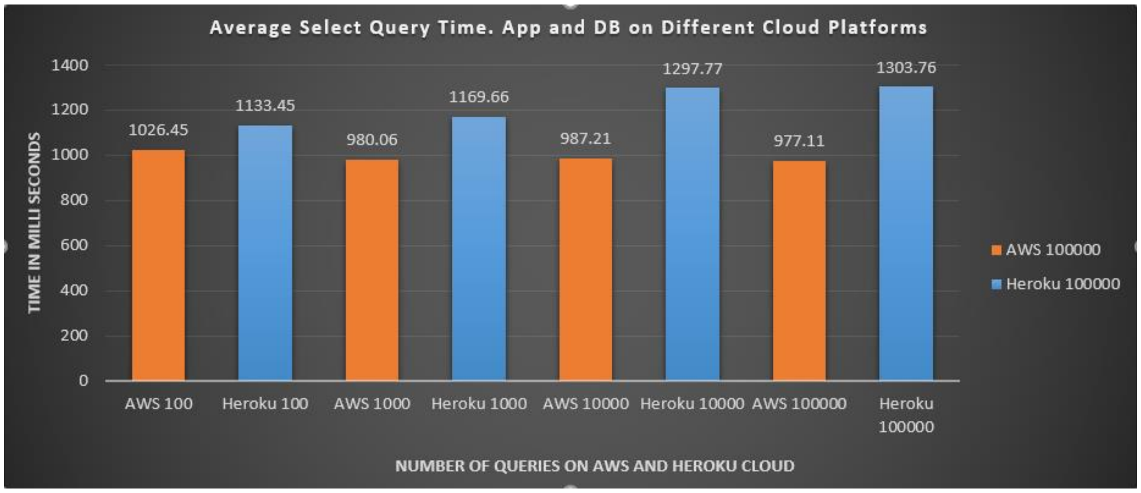


Figure 7.12 Select Query Experiments on Heterogenous Cloud Deployment.

As shown in the above figure 7.12, the respective queries were sent and response time was noted. We are showing the average query fetch time for each type of query to better illustrate the analysis. The response time of AWS application varied from 1026.10 milliseconds to 977.21 milliseconds. In the case of Heroku, the query response time was almost constant till 1000 queries after which as the number of queries increased, the response time also increased varying between 1297.77 milliseconds to 1302.76 milliseconds.

With this experiment, we saw a difference between the response times from AWS based database and Heroku based database for the same application deployed on other cloud platforms. The interesting things to note here will be the spike in response time for AWS application as the datastore was hosted on Heroku, which implies that Heroku database performance is far below then the AWS RDS database instance. Whereas when the AWS RDS instance was used with Heroku deployed application, the response times did not improved. This gave us another interesting observation that Heroku deployed application also performs slow when compared to AWS Elastic Beanstalk-based application. In this experiment AWS cloud deployment clearly outnumbered Heroku based deployments in terms of performance for both compute and storage.

7.13 Insert Query Experiments on Heterogenous Cloud Deployment.

This section describes the set of experiments done to check the performance of the web application deployed in the previous sections. In this experiment, the application was first deployed on AWS cloud platform with data store also on Heroku cloud. The environment configurations are shown in section 7.2.2 and 7.2.3. After successful deployment, the application was stress tested with 100, 1000, 10000, 100000 INSERT queries in separate batches to analyze the performance of the application and to analyze the database request response times. The same procedure was repeated by deploying the same application on Heroku cloud and keeping the database on AWS cloud.

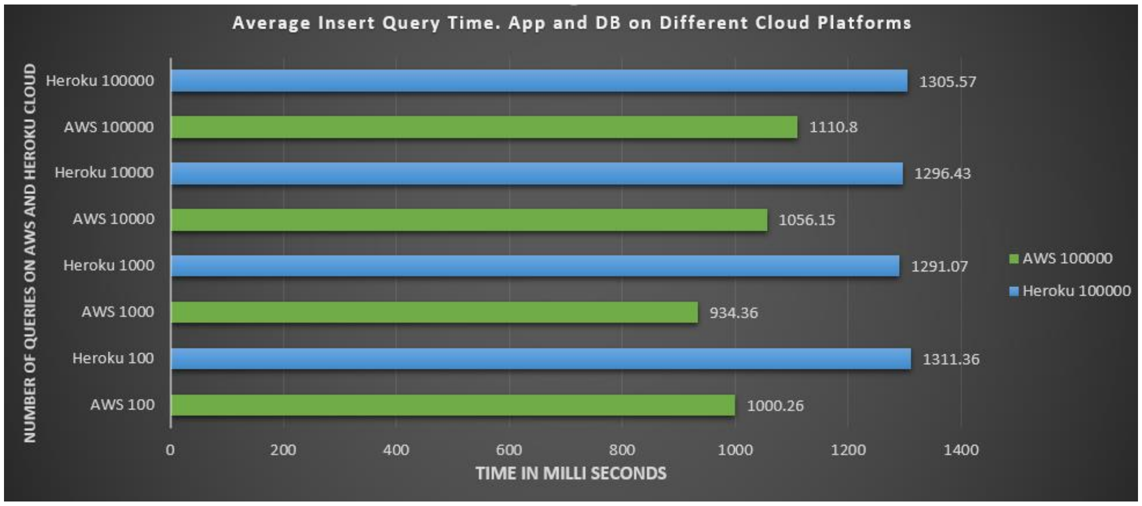


Figure 7.13 Insert Query Experiments on Heterogenous Cloud Deployment.

As shown in the above figure 7.13, the respective queries were sent and response time was noted. We are showing the average query fetch time for each type of query to better illustrate the analysis. The response time of AWS application varied from 1000.26 milliseconds to 1110.8 milliseconds. In the case of Heroku, the query response time was almost constant varying between 1311.36 milliseconds to 1305.57 milliseconds.

With this experiment, we saw a difference between the response times from AWS based database and Heroku based database for the same application deployed on other cloud platforms. The interesting things to note here will be the spike in response time for AWS application as the datastore was hosted on Heroku, which implies that Heroku database performance is far below then the AWS RDS database instance. Whereas when the AWS RDS instance was used with Heroku deployed application, the response times did not improve. This gave us another interesting observation that Heroku deployed application also performs slowly when compared to AWS Elastic Beanstalk-based application. In this experiment AWS cloud deployment clearly outnumbered Heroku based deployments in terms of performance for both compute and storage.

7.14 Update Query Experiments on Heterogenous Cloud Deployment.

This section describes the set of experiments done to check the performance of the web application deployed in the previous sections. In this experiment, the application was first deployed on AWS cloud platform with data store also on Heroku cloud. The environment configurations are shown in section 7.2.2 and 7.2.3. After successful deployment, the application was stress tested with 100, 1000, 10000, 100000 UPDATE queries in separate batches to analyze the performance of the application and to analyze the database request response times. The same procedure was repeated by deploying the same application on Heroku cloud and keeping the database on AWS cloud.

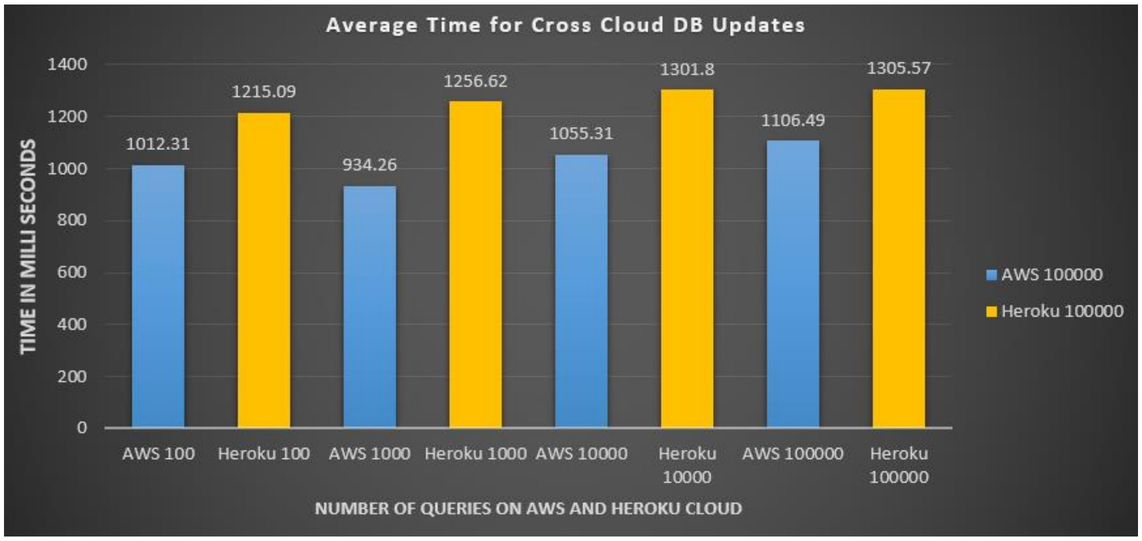


Figure 7.14 Update Query Experiments on Heterogenous Cloud Deployment.

As shown in the above figure 7.14, the respective queries were sent and response time was noted. We are showing the average query fetch time for each type of query to better illustrate the analysis. The response time of AWS application varied from 1012.31 milliseconds to 1106.49 milliseconds. In the case of Heroku, the query response time was almost constant varying between 1215.09 milliseconds to 1305.57 milliseconds.

With this experiment, we saw a difference between the response times from AWS based database and Heroku based database for the same application deployed on other cloud platforms. The interesting things to note here will be the spike in response time for AWS application as the datastore was hosted on Heroku, which implies that Heroku database performance is far below then the AWS RDS database instance. Whereas, when the AWS RDS instance was used with Heroku deployed application, the response times did not improve. This gave us another interesting observation that Heroku deployed application also performs slowly when compared to AWS Elastic Beanstalk-based application. In this experiment AWS cloud deployment clearly outperformed Heroku based deployments in terms of performance for both compute and storage.

CHAPTER 8

SUMMARY AND CONCLUSION

This thesis started with the analysis of manual deployment of a multi-tier web application on AWS and Heroku cloud platforms. The understanding of the deployment and configurations on both platforms helped us to analyze the changes required in the application config files to make it compatible on both the test bed cloud platforms. Once the deployment of the web application was made generic, we analyzed the strategies to migrate the database and map it to the new environments. Once these mappings were successfully tested manually, we began our analysis to automate these deployments.

The automated web application to manage the AWS and Heroku environments was developed using python Flask framework. This web application was REST API enabled which uses boto3 clients to communicate with Elastic Beanstalk, RDS and S3 services on Amazon cloud and Heroku platform and Add-On services on Heroku cloud. As part of deployment strategies, users could either host the application and database on either AWS or Heroku clouds or users had the option of spreading the application components across multiple clouds. An application hosted on AWS Elastic Beanstalk could be mapped to a database hosted on Heroku cloud as an add-on service.

Similarly, an application hosted on Heroku Platform could be mapped on AWS RDS MySQL instance. The whole framework worked autonomously and the application deployment did not require any code change, meaning the generic multi-tier web application was supported on both the cloud platforms seamlessly.

The last part of the thesis was the performance analysis of the migrated web application in both homogenous and heterogeneous cloud deployments. As it could be seen in the experiments, deployment of an application on Heroku cloud platform outperformed AWS Elastic Beanstalk application deployment by almost 88%. Similar was the case with tear down of the application in which Heroku cloud application teardown process was 90% faster that AWS cloud. When it came to performance of the application, AWS came out as the winner for all DML operations such as SELECT which were 11 times faster that Heroku cloud queries, INSERT which were 15 times faster on AWS cloud than Heroku application and UPDATES which were 5 times faster on AWS than Heroku. However, in case of cross-cloud deployment we can see a dip in the performance of AWS application, since the DB was on Heroku platform.

We can conclude from the above experiments that migrating and running an application on Heroku is easy to get started with cheap compute cost for low-performance test or dev deployments. Heroku has auto scaling which costs expensive for with additional Dynos. Elastic Beanstalk is a better bet for running large scale web applications, which offers better performance, control on application, at a lower cost as compared to Heroku.

CHAPTER 9

FUTURE WORK

Heterogeneous cloud application migration can be extended to many existing cloud vendors. In this paper experiments were conducted on only two cloud platforms namely AWS and Heroku, which can be extended to other cloud platforms such as Microsoft Azure, Google cloud, Rackspace, Open Stack to name a few. We have proposed an approach to migrate applications based on config files which were manually generated and environment values were fetched dynamically from the data store. These configurations can be automated in future using configuration management tools such as Chef or Puppet. The proposed framework can also be extended to Cloud Watch services in which we can monitor the stacks across heterogeneous cloud platforms. In future, we can also have RESTful API's through which other applications can communicate with Cloud Merge and services can be used across different applications.

With some more work and robust features added in future, the proposed framework can be used as an enterprise solution for managing heterogeneous cloud applications across several cloud platforms.

REFERENCES

[1] “Sea Cloud Approach” Retrieved December 3, 2016, from http://dci.ufro.cl/fileadmin/Cibse2014/CIBSE2014-SET_095-108.pdf

[2] Patterson’s Paper. Retrieved December 3, 2016, from https://www2.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf.

[3] CloudGenius: A hybrid decision support method for automating the migration of web application clusters to public clouds - IEEE Xplore document. (2016). Retrieved December 3, 2016, from http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6811183 R. Mietzner, T. Unger, and F. Leymann, “Cafe: A generic configurable customizable composite cloud application framework,” in Proc. Confederated Int. Conf. OTM Conf., pp. 357–364, 2009."

[4] " T. Binz, G. Breiter, F. Leyman, and T. Spatzier, “Portable cloud services using TOSCA,” IEEE Internet Compute., vol. 16, no. 3, pp. 80–85, May/Jun. 2012."

[5] “EU Cloud Clusters” Retrieved December 3, 2016, from https://eucloudclusters.files.wordpress.com/2015/05/inter-cloud-pp_dec-2015.pdf.

[6] Posted, & Caldwell, B. (2014, June 4). Cloud migration and portability: What VMware and AWS Aren’t telling you. Retrieved December 3, 2016, from http://www.rightscale.com/blog/enterprise-cloud-strategies/cloud-migration-and-portability-what-vmware-and-aws-arent-telling-you.

[7] Blog2016DoorDash. (2013). Migrating from Heroku to AWS (using Docker).

Retrieved December 3, 2016, from http://blog.doordash.com/post/115409532041/migrating-from-heroku-to-aws-using-docker.

[8] Us, C. (2015, June 18). Top 5 cloud computing challenges | trilogy. Retrieved December 3, 2016, from Blog, http://trilogytechnologies.com/top-five-challenges-of-cloud-computing/

[9] Retrieved December 3, 2016, from http://www.clei.org/cleiej/papers/v18i1p1.pdf

[10] Zhao, J.-F., & Zhou, J.-T. (2014). Strategies and methods for cloud migration. International Journal of Automation and Computing, 11(2), 143–152. doi:10.1007/s11633-014-0776-7

[11] Retrieved December 3, 2016, from https://www.computer.org/csdl/proceedings/srds/2012/2397/00/4784a463.pdf/

[12] Retrieved December 3, 2016, from https://eucloudclusters.files.wordpress.com/2015/05/inter-cloud-pp_dec-2015.pdf.

[13] Linthicum, b. (2015, November 17). Cloud computing APIs pose vendor lock-in risks. Retrieved December 3, 2016, from http://searchcloudcomputing.techtarget.com/tip/Cloud-computing-APIs-pose-vendor-lock-in-risks.

[14] Flask (web framework). (2016, November 28). In Wikipedia, The Free Encyclopedia. Retrieved 09:41, November 28, 2016, from https://en.wikipedia.org/w/index.php?title=Flask_(web_framework)&oldid=

751887607

[15] AWS elastic beanstalk – deploy web applications. (2016). Retrieved December 3, 2016, from https://aws.amazon.com/elasticbeanstalk/

[16] Configure the EB CLI. (2016). Retrieved December 3, 2016, from http://docs.aws.amazon.com/elasticbeanstalk/latest/dg/eb-cli3-configuration.html.

[17] Multitier architecture. (2016, November 30). In Wikipedia, The Free Encyclopedia. Retrieved 12:28, November 30, 2016, from https://en.wikipedia.org/w/index.php?title=Multitier_architecture&oldid=752281182.

[18] Retrieved December 3, 2016, from http://docs.aws.amazon.com/Amazon-RDS/latest/UserGuide/Welcome.html.

[19] (Website, Amazon, 2016) https://aws.amazon.com/rds/

[20] Retrieved December 3, 2016, from http://www.mongodbspain.com/en/2014/08/17/mongodbcharacteristics-future/

[21] ("Cloud application portability with TOSCA, Chef and Open stack"",2014) https://www.researchgate.net/publication/264829793_Cloud_Application_Portability_with_TOSCA_Chef_and_Openstack.

[22] Retrieved December 3, 2016, from https://aws.amazon.com/s3/?hp=tile&so-exp=below

[23] Add-ons - Heroku elements. Retrieved December 3, 2016, from https://elements.heroku.com/addons

BIOGRAPHICAL INFORMATION

Mayank Jain received his Bachelor’s Degree in Information and Technology from Graphic Era Institute of Technology, Dehradun in 2009. He worked as a product developer with Oracle, Bangalore and Infosys Ltd. Pune in India till December 2014, after which he decided to pursue his Master’s Degree in Computer Science at University of Texas at Arlington. During his studying period at Arlington he was more interested in Cloud Computing, Big Data and Machine Learning projects and started this research under the guidance of Mr. David Levine, his professor for the cloud course. He got an opportunity to intern at Viscosity North America from Dec 2016 to June 2016 at Dallas, Texas. He also Interned with Talon Systems, based in Grapevine, Texas. His areas of interest are Cloud Computing, Big Data, and Machine Learning.