A Comparative Study of AWS Cloud Native Application Deployment Models

by

Khandakar Razoan Ahmed
16301188

Sadifuzzaman Diganta
14110013

A thesis submitted to the Department of Computer Science and Engineering
in partial fulfillment of the requirements for the degree of
B.Sc. in Computer Science

Department of Computer Science and Engineering
Brac University
October 2020

Declaration

It is hereby declared that

1. The thesis submitted is our own original work while completing degree at Brac University.
2. The thesis does not contain material previously published or written by a third party, except where this is appropriately cited through full and accurate referencing.
3. The thesis does not contain material which has been accepted, or submitted, for any other degree or diploma at a university or other institution.
4. We have acknowledged all main sources of help.

Student's Full Name & Signature:

Khandakar Razoan Ahmed
16301188

Sadifuzzaman Diganta
14110013

Approval

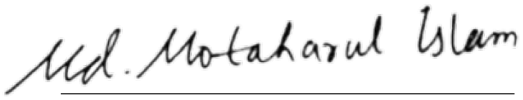
The thesis titled "A Comparative Study Between Three Native Cloud Application Deployment Models" submitted by

1. Khandakar Razoan Ahmed (16301188)
2. Sadifuzzaman Diganta (14110013)

Of Summer, 2020 has been accepted as satisfactory in partial fulfillment of the requirement for the degree of B.Sc. in Computer Science on October 02, 2020.

Examining Committee:

Supervisor:
(Member)



Prof. Dr. Md. Motaharul Islam
Department of Computer Science and Engineering
United International University
Former Associate Professor
Department of Computer Science and Engineering
Brac University

Thesis Coordinator:
(Member)

Md. Golam Rabiul Alam, PhD
Associate Professor
Department of Computer Science and Engineering
Brac University

Head of Department:
(Chair)

Mahbubul Alam Majumdar, PhD
Professor and Chairperson(CSE), Interim Dean, School of Sciences
Department of Computer Science and Engineering
Brac University

Ethics Statement

We, hereby declare that this thesis is based on the findings of our research. All other materials used have been properly noted in the document. This work, neither in whole nor in part, has never been submitted by anyone to any other university or institution for the award of any degree.

Abstract

Application deployment in cloud computing has become the most widely used system now. Developers use cloud computing instead of on-premises data centers in terms of deploying their application. For deploying our application, we need to follow some specific models. For a long time, we have deployed our application in virtual machines. Nevertheless, nowadays the Docker container and Serverless models are gaining popularity due to its important features. Therefore, we need to know which model is better in terms of request-response, which model is error-prone and gives better throughput. In our paper, we have done a comparative study between these three native cloud application deployment models. The parameters of this study are HTTP response in a fixed time, error percentage, standard deviation, and throughput. We have used Amazon EC2 service for the Virtual Machine model, Amazon ECS service for the Docker Container model, and AWS Amplify for the Serverless model in the implementation part and Apache JMeter tool in the test part. In our study, we have found that the Serverless performed essentially better than the Docker container and Virtual Machine in the total HTTP response and throughput part although Docker Container did better than the Virtual Machine. And surprisingly, Virtual Machine did better than Docker Container and Serverless in the error percentage and standard deviation part.

Keywords: Cloud Computing, AWS, Native Cloud Application, Deployment Model, Virtual Machine, Docker Container, Serverless, Amazon EC2, Amazon ECS, AWS Amplify.

Dedication

We would like to dedicate this research work to our parents. Without their utmost support, we could not have achieved what we achieved.

Acknowledgement

To begin with, all praise to the Great Allah for whom our thesis have been completed without any major interference.
Further, we would like thank our honorable supervisor Dr. Md. Motaharul Islam sir for his support, feedback and guidence. He encouraged us to conduct the research, give guidance to us and always were present to offer any help we could ask for.
Moreover, We also thank our respected BRAC University for providing us with the necessary facilities and resources during our research period.
Finally, special thanks to our parents; without their constant support and encouragement it might not have been possible. With their kind support and prayer we are now on the verge of our graduation.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

The popularity of Cloud computing is increasing day by day. People are becoming dependent on Cloud computing interms of data store, collect and manage. Most importantly, it makes these tasks easy and make our daily life easier. Cloud computing depends on shared computing resources through the internet instead of having local machines or servers. Cloud computing is the most ideal choice for the store, handle, and maintain millions of data [1]. Now we can store and retrieve our data from anywhere at any time [3]. A cloud can be private or public. There are some service providers available who give us the public cloud service, for example, Amazon Web Services (AWS), Google Cloud Platform, and Microsoft Azure [2].

Among them, AWS is better public cloud service provider in light of four advantages like fault-tolerant, high scalability, resilience, and availability zone. Fault tolerance is the property that empowers a framework to keep on working appropriately in case of the disappointment of a portion of its segments. Scalability is the attribute of a system that depicts its capacity to adapt and perform well under an expanding workload likewise will have the option to increment or lessening its level of performance according to the demands. Resilience is the ability to recoup from the troublesome and harder conditions. AWS gives us the resources we need as well as gives us the highest fault-tolerant advantage, high scalability, and best resilience facilities. AWS also has the most number of availability zone 48 with the 20 regions on over the world. So, we can choose any zone from any region where we want to deploy our application. Also, our data will be stored in another two data centers for backup, so under any misfortune or pandemic, our information will be protected.

AWS provides us more than 90+ services like Amazon Elastic Compute Cloud (Amazon EC2), Amazon Elastic Container Service (Amazon ECS), AWS Lambda, Amazon Simple Storage Service (Amazon S3), AWS IoT Greengrass, AWS CloudFormation, virtual private cloud (VPC), AWS Identity and Access Management (IAM), AWS Amplify etc [11][8]. We have utilized a large number of them in our project. Amazon EC2 is a web service that gives us a secure, resizable process limit in the cloud [11]. EC2 is intended to make web-scale [10] distributed computing simpler for developers. It does not just give us the unlimited authority of computing resources yet in addition lets us run on Amazon's proven computing environment. Amazon

VPC gives us the serious security features, for example, security groups and network access control lists, to empower inbound and outbound filtering at the instance and subnet level [11].

Amazon S3 is a scalable, rapid, web-based cloud storage service intended for online backup and archiving of information and applications on AWS. Additionally, it is an object storage administration that offers industry-driving scalability, data accessibility, security, and performance. ECS is an Amazon container administration service. AWS ECS is an exceptionally adaptable, super-quick, and fault-tolerant service help that makes it easy to create, launch, stop, and maintain our containers on the cluster[14]. We can run our tasks on a serverless infrastructure (managed by AWS Fargate) for full command over our infrastructure. We additionally can run our services on a cluster of EC2 instances that we can oversee. AWS Amplify is an end-to-end solution that allows the mobile and front-end web developers to develop and deploy reliable, scalable full-stack applications.

To deploy applications, cloud is the most brisk, reliable, and useful approach. From the very beginning, people deploy application on-premises basis. For quite a while, this was the main method of deployment. On-premises always require a greater expense in getting off the ground as hardware and installation are necessary for the housing of on-site servers. In addition, maintaining the infrastructure and wasting resources are some big disadvantages of on-premises [13]. To diminish these issues Cloud computing is very much helpful. As every year, many applications are deployed so it is necessary to look for smooth computation for large data and performance of those applications. To deploy any kind of application or software we have four kinds of deployment models such as Public Cloud, Private Cloud, Community Cloud, and Hybrid Cloud. Among them, public cloud is the best model [13]. Some of the advantages are unsophisticated installation and use, easy access to data, cost-effectiveness, continuous operation time, eliminated need for software, and simple admittance to data, cost-effectiveness, and scalability [13].

A virtual machine (VM) is an image file that is overseen by the hypervisor [1]. Hypervisor permits the computing resources (RAM, CPU, Memory, Storage, etc.) of the server to every single Virtual Machine [12], and keep them isolated to avoid interference. VM represents the behavior of a separate machine and the capacity to perform tasks as a separate machine. In our proposed project, we have built the infrastructure for deploying our web application in the VM. AWS provides us the VM's services by the Amazon EC2 service and we have utilized EC2 instance for the execution part.

Docker containers are primarily used to deploy our applications in microservice-based architecture [9]. We divide the whole services into small parts that we call microservices [4]. The benefits of Docker containers are fast speed, lightweight, compactness, quick conveyance, and scalability [2][7]. Because of these advantages, it becomes attractive to develop containerized services nowadays. Docker container is less costly than the virtual machine [9]. In a container environment, we virtualized the applications and executed them [6]. Containers are running in an isolated way on top of the operating system's kernel [7]. Docker container guarantees that our

application is going to work consistently in every environment. In our proposed project, we are going to build the infrastructure for deploying our web application by using the Docker container model. AWS provides us the Amazon ECS services and we have used Cluster and Task Definition of the Amazon ECS services for the implementation part [7].

Serverless processing is an execution model for the cloud in which a cloud specialist service provider consequently distributes and afterward charges the client for just the computing resources and storage expected to execute a specific bit of code. Generally, there are still servers included, yet their provisioning and support are altogether dealt with by the service provider. The two major advantages of serverless computing are developers can fully focus on the business objectives of the code they compose, instead of on infrastructural buildup and organizations just compensation for the computing resources they use. Also, the organizations do not have to look for buying physical hardware or leasing cloud instances.So we do not have to think for the server or any infrastructure build up we just have to look after the code that we want to deploy. So, it is one of the most easiest model to deploy your application. In our proposed project, we are going to deploying our web application by using the Serverless model. AWS provides us the AWS Amplify services and we have used this services for the implementation part [21].

1.2 Research Problem

From the very beginning of our research, we have noticed almost applications or software are being deployed on the data center which we can call on-premise software. As we are following this traditional type of application deployment, lots of problems and issues are kept happening around the software deployment domain.Some major issues with the on-premise application are, on-prem will always require a higher investment in getting off the ground because hardware and installation are necessary for the housing of on-site servers. As we know that hardware and software installation is very much time consuming, complicated, and overpriced for most of the companies or organization. Scaling with an on-prem solution will require hardware or software upgrades to fully account for the increased data flow. Most of the companies do not need all the storage they may allocate for them. Mostly it depends on the time and situation that some time they need more storage than they have or sometimes less but they cannot scale their resources all the time.

Besides, companies also have to pay more than they used which is critical for a small company. In addition, maintaining the hardware is also a big issue. By using the concept of Cloud computing, almost all the issues have been solved. After facing the above problems, almost all the companies are shifting to cloud computing from on-premise. And a clear knowledge regarding the native cloud application deployment method was badly needed. Cloud-native application deployment means an application that is created and build in the public cloud. It implies that the apps live in the public cloud, as opposed to an on-premises data center. So, in our proposed paper our main focus will be analyzed that in which cases we should follow which deployment method. Then the virtual machine model of cloud computing became very popular in terms of application deployment.

However, the virtual machine model has become a traditional system in terms of deploying applications as nowadays people are using the other two models. These models are Docker Container and Serverless. These models have their advantages and disadvantages also they give different performances. As these three models are being used daily in different circumstances we need a proper analysis to understand their performance and behavior. We need to understand which model gives better performance in which cases. And also to get a clear idea about their performances and behavior we need to have a real-life scenario-based performance test. We also need to implement the deployment model adequately so that we get the best performances from these models. During the implementation part, infrastructure build-up with the AWS services is a tough task to do because to implement these services anyone needs AWS certification types of skills. In these infrastructures build-up sessions, we tried to do these by using the automation concept which is a big thing. To make a good performance analysis, setting up the parameter, and creating a real-life scenario-based test is also a big challenge.

1.3 Research Objective

This thesis aims to build the infrastructure for the three native cloud application deployment model. The infrastructure will be built using AWS services and performance will be tested using Apache JMeter therefore the objectives of this research are:

1. Deeply understand the Cloud computing and Application Deployment.
2. Understanding the various native cloud application deployment methods and finding the necessary AWS services to implement these.
3. Build up the cloud infrastructure with the help of AWS services using Automation concept.
4. Get the highest amount of advantages of cloud computing.
5. Get a clear knowledge about how to analyze the performance of a cloud native application
6. Creating a real-life sceario-based performance test by using Apache JMeter.

1.4 Scope and Limitation

The analysis topic we have selected is the most widely used thing in our software world. After shifting from traditional on-premise data center deployment to cloud-based deployment model Virtual Machine, Docker Container and Serverless have become the most used model. So our paper can set a standard performance test between these models. And this paper is fully implementation based study.

All necessary steps have been taken to make sure the models give their best performances which will be easy to analyze. To implement this study we needed the skills of AWS certified engineer which is tough. Also using services like EC2, ECS, and Amplify is costly as AWS is a public cloud service provider. So we have to spend a healthy amount of money to make this project possible. Also, need a clear basic knowledge and skills of Virtual Machine, Docker Container. With this much of the scope of the project, the limitations couldn't create much affect on the project.

Chapter 2

Related Work

In [1], authors have analyzed the performance of VM and containers with the same hardware and software configuration on the basis of performance efficiency, security, storage, isolation, and networking.

In [2], a performance comparison has been shown between Linux containers and VM where the author gave the reasons why the container is more useful and attractive to software developers and infrastructure administrators. And later conducted a comparison on the basis of performance and scalability. In [9], authors have described Docker System, Docker's applications , and advantages in the cloud Computing through an application instance.

In [4], authors have investigated and analyzed the status of the development of container network, container network model, and network solutions. They have also designed an experimental way to measure the performance of three mainstream network solutions such as Calico, Swarm Overlay, and Flannel where they have been able to prove that Calico is of the highest performance but its configuration is really the most complicated among all. Besides, the other two solution's configuration is simple but the performance is slow.

In [5], the authors proposed a comparative study between VM and Docker Containers. They discussed some basics of VM, Docker containers,virtualization techniques, and their advantages and disadvantage. Lastly compared them on the basis of booting time, standardization, portability, security, low redundancy, resource distribution, etc.

In [6], authors have operated a comparative study between Xen and Docker. They have used one extensive micro-benchmarks to compare virtualization upward of distinct elements on a host system between these two virtualization technologies. They have also added another two macro-benchmarks called HPL and YCSB where HPL is for a case of high-performance computing applications and YCSB for a case of online transaction processing applications, to understand the virtualization impact on real-world application.

Though in [7], authors have pointed that services that have been deployed by AWS ECS perform worse in comparison with services that have been deployed by Ama-

zon EC2 VMs. They have conducted the performance test based on throughput, response time, and CPU utilization. However, in our paper, we have tried to conduct the performance test with a proper testing plan and strategy where we have conduct real-life scenario-based performance tests.

In [8], the authors presented a performance study to examine the proper way to utilize containers from different viewpoints. They have led some experiments to measure performance variations between application containers and system containers. Furthermore, they have inspected the service quality of ECS and Google Container Engine.

In [19], They have claimed that serverless computing environments can support dynamic applications in correspondence when a partitioned task is executable on a tiny function instance. They have likewise demonstrated results of throughput, network bandwidth, a file I/O, and compute performance concerning the concurrent invocations. Additionally, they have deployed a set of functions for shared data processing to address the elasticity and later demonstrated the variety between serverless computing and virtual machines for cost productivity and resource utilization.

In [20], They have created Firecracker which is an open-source VMM specialized for serverless workloads, yet generally useful for containers, functions and other compute workloads within a sensible arrangement of requirements. They have deployed Firecracker in two serverless services of AWS which are Lambda and Fargate, where it upholds thousands of production workloads and billions of requests for each month. They have additionally depicted how specializing for serverless informed the plan regarding Firecracker, and what they gained from consistently relocating Lambda clients to Firecracker.

In [21], authors have introduced a cost examination of a web application created and deployed utilizing similar scalable scenarios with three different methodologies such as monolithic architecture, microservice architecture operated by the cloud client, microservice architecture operated by the cloud service provider. Test outcomes demonstrated that microservices can help lessen infrastructure costs in contrast with standard monolithic architectures. Also, the utilization of services specifically designed to deploy and scale microservices diminishes infrastructure costs by 70% or more. Finally, they have likewise depicted the difficulties that they have confronted while implementing and deploying microservice applications.

In [22], this paper, they have briefly studied about serverless model and contended that it does not need to be restricted to the cloud. They have shown how a similar model can be applied at the micro-level of a single machine. In such a model, certain operating system commands are treated as events that trigger a serverless response. This response comprises of deploying and running code just in light of those events.

Chapter 3

Methodology

3.1 Proposed Idea

The goal of this paper is to conduct a comparative analysis of three native cloud application deployment models. Therefore the idea has been designed keeping in mind the need for creating infrastructure for those models. Here to build the infrastructure we need a public cloud service provider such as AWS, Azure, and GCP. Among them, we have chosen AWS. We have customized an application named We-Care Hospital which we have deployed in those models to see their performances.
After customization of the application has been done there rises the question of collecting the data in terms of the implementation of those models. In solving this problem different studies done in the past have presented solutions that are the most suitable services of AWS are Amazon EC2 for Virtual Machine, Amazon ECS for Docker Container, and AWS Amplify for Serverless. Here, we also need services like Amazon VPC, Amazon Cloudformation, AWS IAM, etc. For creating the template of Cloudformation we have used the Visual Studio Code tool.

3.2 Evaluation Methods

During evaluating the data we have selected some parameters such as Total HTTP response, Average time for response, lowest time for response, highest time for response, Error percentage, Standard deviation, and Throughput. After collecting the information about the data we have analyzed each models performance based on these parameters. We have also compared their performance based on real-time data visualization pictures. However, of course, different models give different performances on different cases and we have evaluated that which model is better in which cases.

Chapter 4

System Architecture

Here we describe the architectures of the three native cloud application deployment model which are EC2 based, ECS based and Amplify based. We have build these architectures with the help of AWS services.

4.1 EC2 based System Architecture

The EC2 based cloud-native application system architecture is introduced in the Fig. 1. We have deployed a sample healthcare website application and analyzed this web application's performance by Apache JMeter from our local machine. We need to do this performance test through the Internet. So we have to setup Apache JMeter in our local machine and then we can send an HTTP request through the internet to our web address. Here we need Amazon Route 53 service. Amazon Route 53 is a highly accessible and scalable cloud DNS web service. Amazon Route 53 connects user requests to the infrastructure successfully. Those infrastructure run in AWS – such as Amazon EC2 instances, Elastic load balancers, or Amazon S3. Amazon Route 53 service connects our HTTP request to our infrastructure so that we can access the hospital website. When our HTTP request tries to access the hospital application hosted in EC2 Webserver, HTTP requests served through Internet gateway attached in the VPC Network. Internet gateway sends the HTTP request to the EC2 instance or machine which is working as a web server.
In the Fig. 1. the peripheral box signifies the AWS cloud. The immediate inner box signifies a region that implies we are working in a specific region of the AWS cloud. We have deployed EC2 instance in the closest location which is in Mumbai(India) and the short name of the region is ap-south-1, Mumbai. This region has two Availability zone which implies there are two data centers in this region and correspondingly AWS has more than 48 availability zone where we can deploy our application. As Fig. 1. shows we have built a VPC network for network requirements of the application. There is a public subnet in the VPC to isolate network traffics and the EC2 box is placed in the public subnet. Our hospital website application is running on the Apache HTTPD service on our web server.

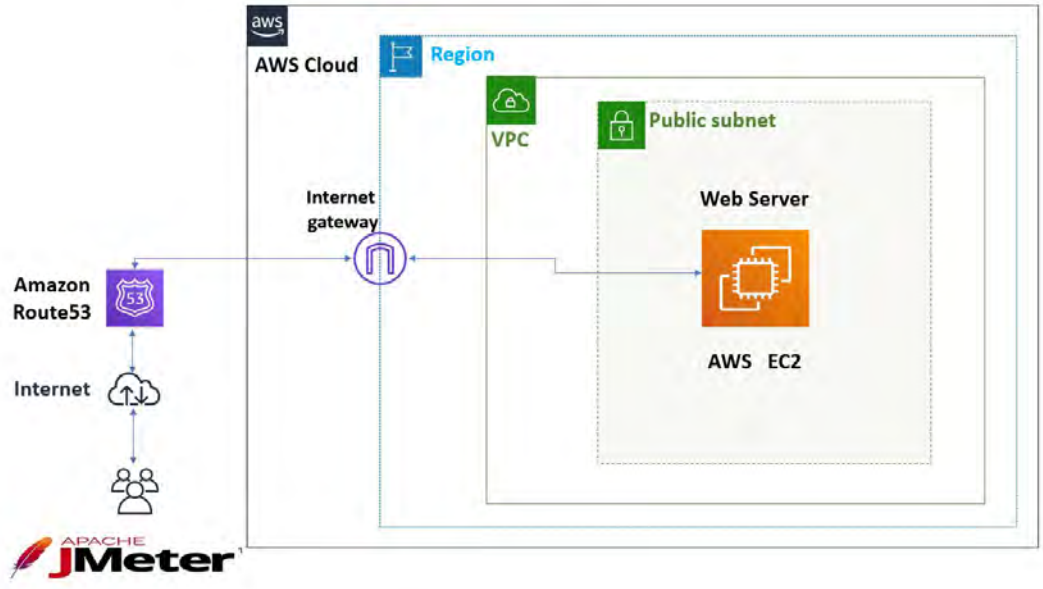


Figure 4.1: EC2 System Architecture

4.2 ECS based System Architecture

The ECS based cloud-native application system architecture is introduced in the Fig. 2. We have deployed the same healthcare website application and analyzed this application/web application performance by the Apache JMeter from our local machine. In Fig. 1. and Fig. 2. clearly shows that two of our system architecture is quite similar because we use the ECS service in the EC2 infrastructure. Here we have to use the Docker container model deployment in the EC2 instance.

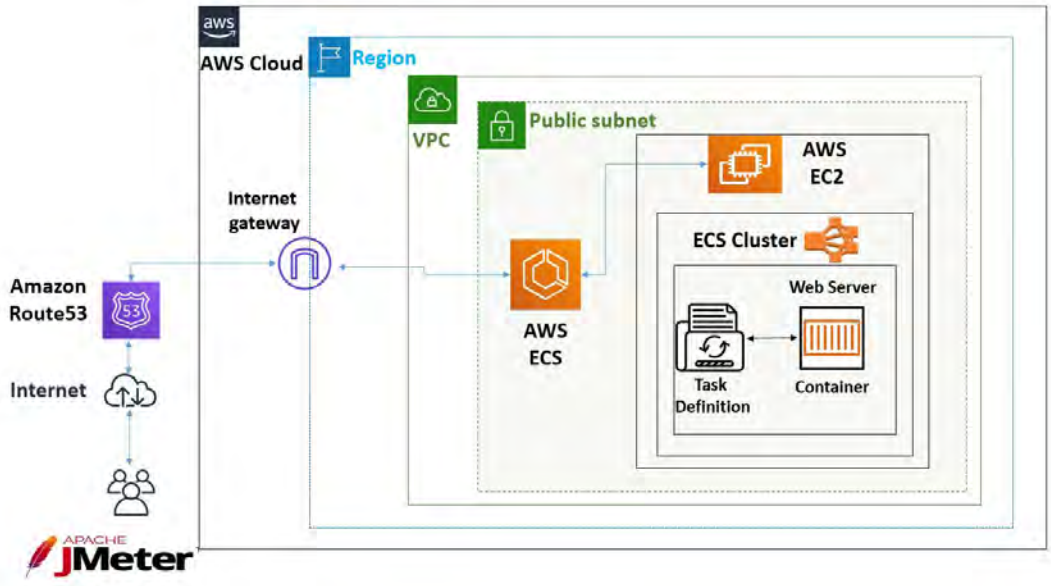


Figure 4.2: ECS System Architecture

Firstly, deploy the application in the ECS service first we need Amazon EC2 instance where we can create an ECS Cluster. Amazon ECS cluster is a logical gathering of tasks or services [14]. We can register one or more Amazon EC2 instances, likewise

referred to as container instances with the cluster to run the tasks that we make. So the cluster manages the task definition and registers the EC2 instance with the container where our webserver is running. As cluster manages the group services, to prepare our application to run on Amazon ECS service, we need to create a task definition which we can call it as a blueprint for our application. Task definition is a text file, in JSON format. It depicts one or more containers, up to a limit of ten, that structure the application. We can determine various parameters in the Task definitions for our application. The parameters that we need to determine are which containers to utilize, how much data volumes ought to be utilized, and which ports ought to be opened for the application. The particular parameters for the task definition rely upon the requirements of our application. Moreover, we can utilize the serverless infrastructure that Fargate provides. At the point when our tasks are run on Fargate infrastructure, the cluster resources will supervise by the Fargate. However, in this project, we have used the EC2 infrastructure [14].

4.3 Amplify based System Architecture

The Amplify based cloud-native application system architecture is introduced in the Fig. 3. We have deployed the same healthcare website application and analyzed this application/web application performance by the Apache JMeter from our local machine. In Fig. 3. clearly shows that this system architecture is bit different from other two architecture because we use the Amplify service here.

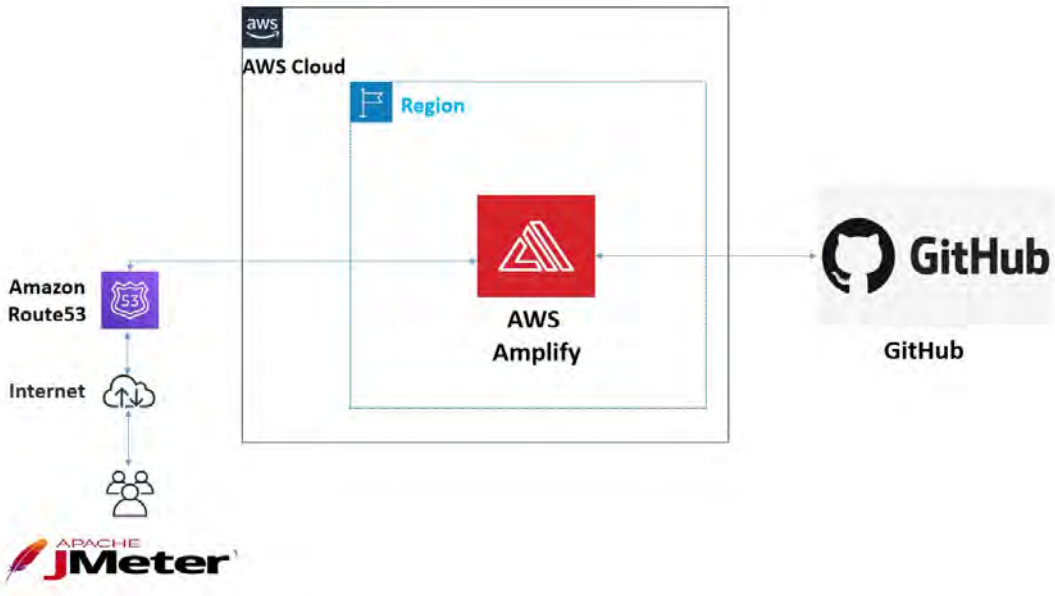


Figure 4.3: Amplify System Architecture

Here, to deploy the application in the AWS Amplify service first we need the Github repository. We have connected the AWS Amplify service with our Github repository where stored all the code related to our application. The rest of the things have been managed and created by the Amplify service. This is the main advantage that here we just have to store the code and connected to the service. Amplify created all the necessary things to host the application.

Chapter 5

Implementation

5.1 Infrastructure Build

For the implementation part, we have utilized some AWS services. First of all, to construct the network part, we have used VPC service where we have set the subnets, route tables, internet gateway, Elastic IPs. To store our data and file we have utilized the S3 bucket where we need to create our bucket and store the data. We have used AWS IAM which is a web service. By IAM we obtain the securely control access to our AWS resources. The complete infrastructure is build using an automation concept and the automation is done by using an AWS service called CloudFormation. This service is considered as Infrastructure as a Code (IAAC) service. We have made a YML layout document for building the EC2 and ECS administration.

In the format document, we have referenced and select everything that we have to do manually for making these services. In table 1, we can see, we have chosen the T2.Nano type instance for EC2 service which consists of 1 vCPU and 500 MB Memory. For our project, this type of instance is a decent choice. But for ECS service we have chosen the T2.2xlarge type instance consist of 8 vCPUs and 32 GB Memory [14]. To support the microservices we need to choose this big instance yet we have restricted the instance to create 1 container consist of 1vCPU and 500 MB memory so we can play out the test with a similar sort condition. For the AWS Amplify, we didn't have to take care of the instances as it is a serverless service. We have created a repository in Github. Github is a cloud-based repository which is a location where all the code for a particular project is saved. Each project has its own repository, and you can access it with a unique URL. So, we have created a repository and stored all the code related to our application bu pushing it. Then, we have created an app in AWS Amplify and connected it with our Github repository. AWS Amplify then collected the code and arranged all the libraries and binaries. After that Amplify deployed our application and hosted it.

5.2 EC2 Based Infrastructure Development

The complete infrastructure of EC2 is build using automation and the automation is done by using an AWS service called CloudFormation which is considered as Infrastructure as a Code (IAAC) service. Here have included this part to show how

Table 5.1: Environment and Tools

AWS Services	AWS IAM
	Amazon ECS
	Amazon VPC
	Amazon EC2
	Cloudformation
	AWS Amplify
Tools	Apache JMeter
	Visual Studio Code
	Github
Network performance	EC2: Low to Moderate
	ECS: Moderate
Instance Type	EC2: T2.Nano: 1 vCPU and 500MB
	ECS: T2.2xlarge: 8 vCPU and 32 GB

we have created the infrastructure by using the automation concept.

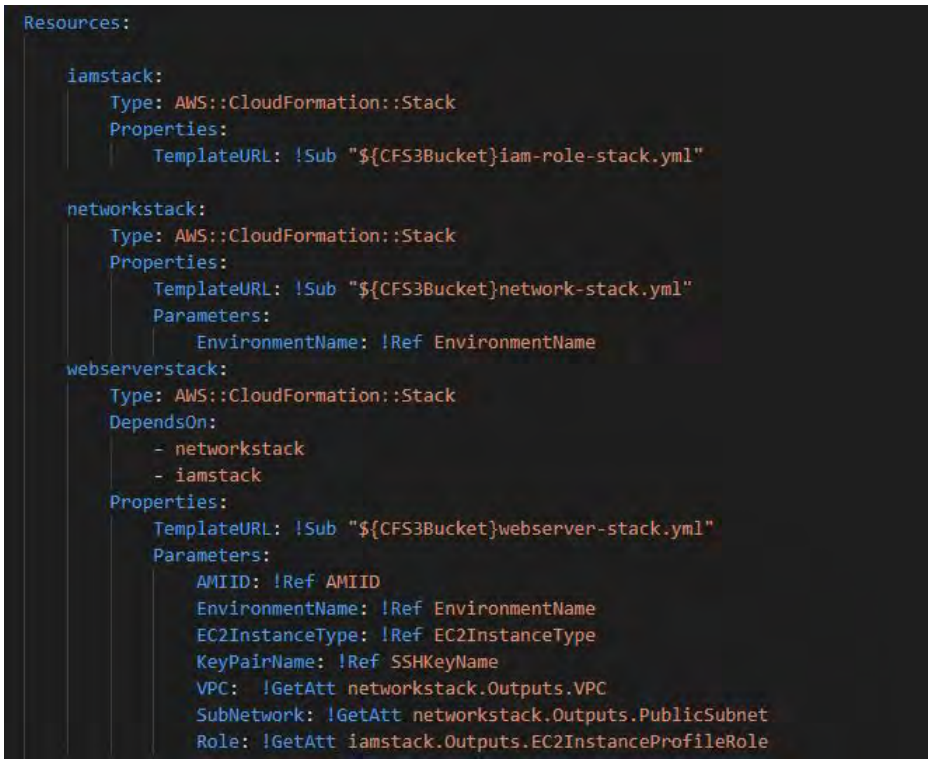
Infrastructure as a code is the process of managing and provisioning computer data centers through machine-readable definition files, rather than physical hardware configuration or interactive configuration tools. AWS CloudFormation is one kind of IaaC service offered by AWS. We are creating the main stack through CloudFormation service from the AWS management console, it is called Master-Stack. This master-stack will call all the template YML files from the S3 bucket to create a nested stack. See below the content of Master Stack YML file:

5.2.1 Master-Stack-Template File (yml)

The Master stack will first create the IAM stack and Network stack which is required for other EC2 instances. Upon creation of these two stacks, the master stack will create one EC2 machine – which is Webserver Stack. The properties of these machines are defined in the YML file. Here we have to keep in mind that without a master stack other three-stack are nested because we are calling these stacks inside our Master-Stack and our Master-Stack is root stack. If we try to explore the Web Server Stack part we will notice that when the master stack tries to create Web Server Stack it is shown this stack depends on the IAM stack and Network Stack. SO, Master-stack first have created these stacks then created rest of the three.

5.3 Performance Test

We have stored the healthcare website template in the S3 bucket and run on the webserver. The website has been customized by us for the test to create a real-life scenario based comparison. For the comparison test part, we have utilized Apache JMeter [15]. Apache JMeter is an adaptable and smart tool where we can test the performance of both static and dynamic web applications. We expect to lead a wise and realistic test plan. Our customized website has run on the webserver on both models and we have tested on both the models. In the deployment part, we have



selected the ap-south-1(Mumbai) region which is the closest from our destination.

However, at the test period, we have selected the us-east-1(North Virginia) region in the JMeter with the goal that Latency can be there and make this test realistic. To lead this test, we have to pick and select some particular ideas and plans [15]. We have set the test on three steps to make the test more realistic and real-life scenario-based. The three steps are Linear Load, Step Load, and Peak Load. Native JMeter doest not support three complex load tests like this so we have used a third-party plugin called Ultimate Thread Group. This plugin helps to set the thread group parameters and examine the system behavior after every new group of users is added. JMeter does not offer a native listener to identify the number of active virtual users for a thread group at every given point of time, so we have utilized another third-party plug-in called Active Threads Over Time. It presents this number on some simple yet concise graphs.

In our test plan, the initial step is Linear Load where all the models give their basic performance because this is one of the typical performance testing. Here we have chosen 500 threads implies 500 virtual user for the test. We have set the parameters that this test begins with the 10 seconds of initial delay and then it increases from 1 to 500 in a few seconds. After that 500 threads hit consistently and we need to see how our web server performs. The load had been generated for 600 seconds. In fig 3 you can see the active threads over time where Jmeter is visualizing how the load is generating. Jmeter also collects the data that during the test time how many requests are being responded from our website in total, how many requests are being responded in every second how many seconds this website is taking to respond to the request, and how many errors are being created by the webserver. This Linear Load test run in both models EC2 and ECS.

Secondly, We need to conduct a step load test. Here the initial stage is similar to the Linear load test but after a particular time, one more thread group is added. In this way, a total of 6 threads is added through a loop. For example, we have set the initial parameters as the initial stage is 500 threads send the HTTP request to our webservers in every second in 70 seconds afterward one more step is added where more 500 threads start to hit with 15 seconds initial delay and hit for 220 seconds. Another 500 threads start to hit after 30 seconds with the initial delay of 60 seconds and hit for 180 seconds. In this way total, 6 thread group send the request to our website so in this Step load test our webserver has to respond to almost 3000 threads, and this increase from 500 in step by step. We need to check our websites how it keep the pace and increase it with the requirement. Ultimate Thread Group plugin helps to set the thread group parameters and examine the system behavior after every new group of users is added.

Lastly, we have utilized a short time intensive load test called Peak load. It creates an unusual environment to evaluate the reaction of the server. The server needs to respond on such extreme load jumps and we can see that in a normal load test when an extreme situation takes place how our webserver faces it and also how well it recovers from the traffic burst. All the models react differently so we need to monitor their reaction. In the Peak load test, JMeter start with the usual 500 threads with the 0 seconds initial delay. 500 users hit our website for 600 seconds and then suddenly the threads number jump from 500 to 3000 at a glance and then we perceive how the server responds in this situation and we assemble the data in the meantime.

Chapter 6

Data

6.1 Data Processing

As we have used Apache Jmeter for the performance test, this tools creates virtual threads and hit the website for a specific time and process the data about how the website response to those HTTP request. It processes Summary Report and Aggregate report in CSV format. Then we collect that CSV file and collect the data of performances.

Table 6.1: Performance Analysis for Linear Load Test

Label	Samples	Average	Std. Dev.	Error%	Throughput
Linear: Index Page	34465	4148	1921.05	0.00%	48.44
Linear: Doctors Page	30777	2564	1342.31	0.00%	43.38
Linear: Facilities Page	20461	1888	1002.67	0.00%	28.83
Linear: Contact Page	11916	4424	2014.28	0.00%	16.80
Linear: Pricing page	8499	1715	877.97	0.00%	11.99
TOTAL	106118	3089	1872.48	0.00%	149.15232

As we have got our data performance analysis for the Linear Load test, Table 5.1 shows the results that we get from our Apache JMeter tool. It processes Summary Report and Aggregate report in CSV format. Here in the table, we getting information regarding Total HTTP responses, Average time for HTTP response, Standard Deviation, Error(%), and Throughput about every page of our websites. Then we merge that and collect the average data of performances.

Table 6.2: Performance Analysis for Step Load Test

Label	Samples	Average	Std. Dev.	Error%	Throughput
Step: Index Page	21028	3252	3608.87	0.00%	29.01
Step: Doctors Page	18605	1275	15643.54	6.20%	23.53
Step: Facilities Page	12260	1009	11021.88	1.58%	15.70
Step: Contact Page	7112	1297	8561.22	0.62%	9.05
Step: Pricing page	5089	527	4284.11	0.59%	7.10
TOTAL	64094	1816	10449.18	2.22%	80.95

As we have got our data performance analysis for the Step Load test, Table 5.2

shows the results that we get from our Apache JMeter tool similar like Linear load test.

Table 6.3: Performance Analysis for Peak Load Test

Label	Samples	Average	Std. Dev.	Error%	Throughput
Step: Index Page	12829	103301	240540.06	15.42%	13.00
Step: Doctors Page	9242	9312	73086.93	7.85%	9.38
Step: Facilities Page	5984	9733	77569.56	3.36%	6.07
Step: Contact Page	3486	9188	74917.32	1.98%	3.54
Step: Pricing page	2462	8504	75151.88	1.67%	2.50
TOTAL	34003	44776	165530.38	8.86%	34.46

Similarly, Table 5.3 shows the results that we got of Peak Load test similar like Linear load test.

6.2 Data Visualization

As we have mentioned before that we have conducted our performance test by utilizing the Apache JMeter tool. To get clear visualization of the real-time performance of the models we used a third-party plug-in called Active Threads Over Time. Also, we have collected three types of visualized data to measure the performances of each model in three step-based load test. These types are Graph Results, Active Threads Over Time, and Response Time Over Time.

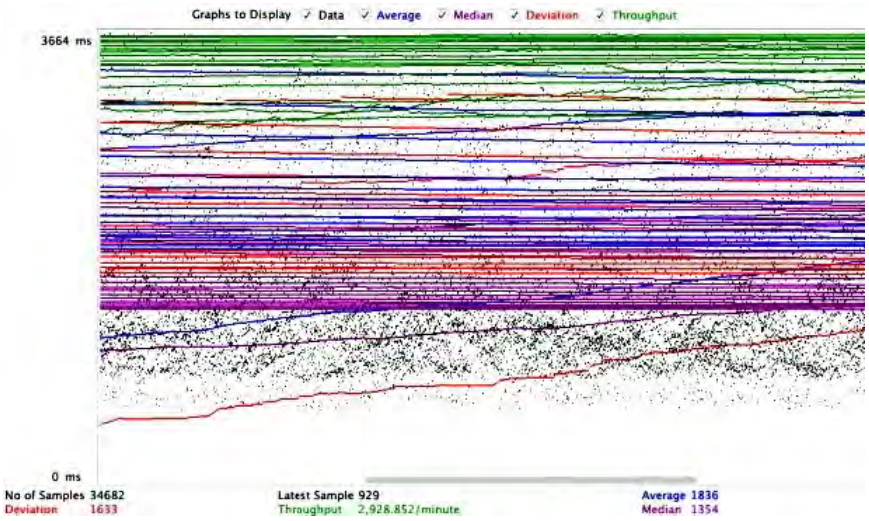


Figure 6.1: Graph Result Ec2 Linear Load

In the Fig. 5.1, we can see that we have got Graph result basis data visualization of EC2 based Website real-time performance in Linear Load test.
In the Fig. 5.2, we can see Active Threads Over Time basis data visualization of EC2 based Website real-time performance in the Linear Load test. However, it is similar to Fig. 5.1 that both figures are of the same model and the same test but the visualization is different.
In the Fig. 5.3, it shows Response time Over Time basis data visualization of EC2 based Website real-time performance in the Linear Load test.

Figure 6.2: Active Threads Over Time Ec2 Linear Load

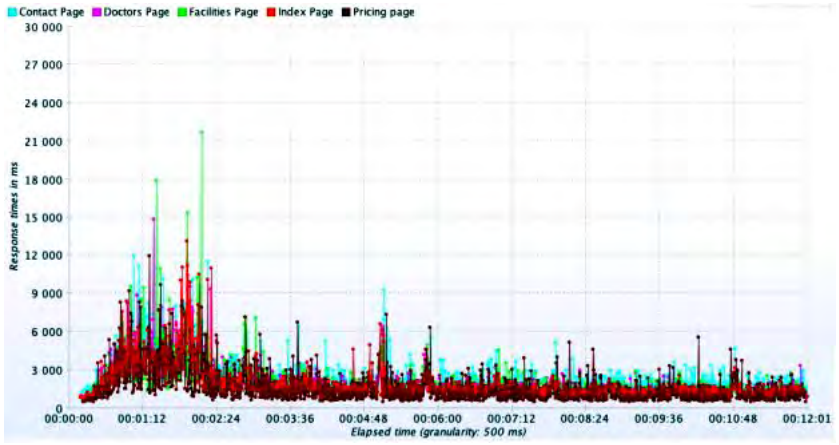


Figure 6.3: Response Time over Time Ec2 Linear Load

Chapter 7

Evaluation

This section describes the way we have collected data. We have applied the Apache JMeter tool for this performance test. In this section, we also examined our data, and based on the data we decide which models perform better. Here the data has been collected based on the response time over time. Then we have examined the models performance on the basis of error percentage, standard deviation.

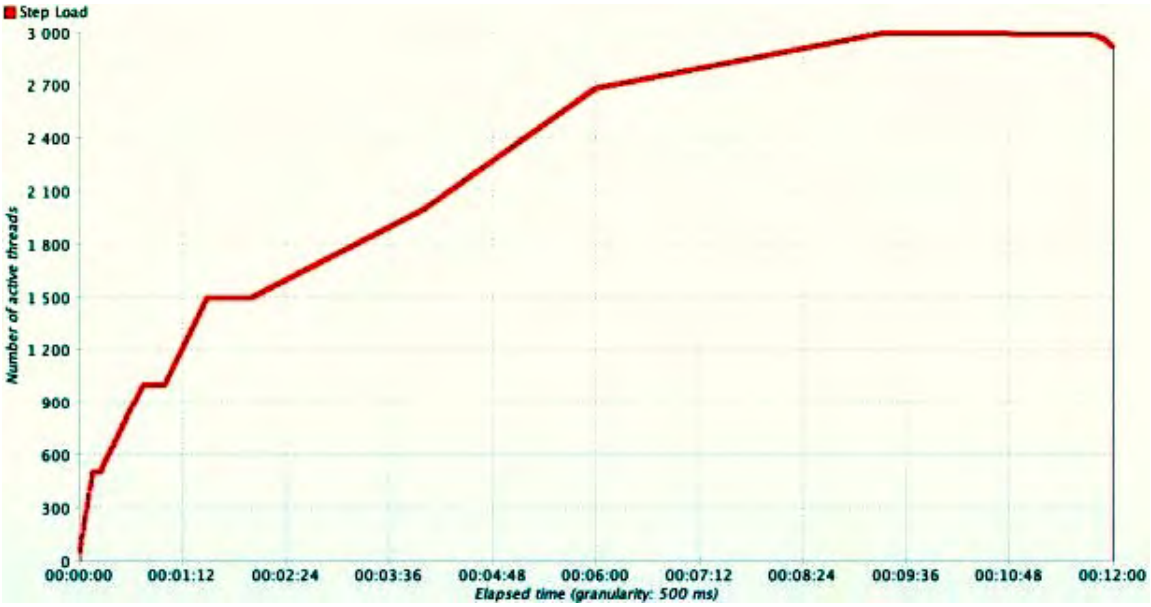


Figure 7.1: Active threads over time EC2 Step load

In Fig. 4. we can see how it looks like when the Step load is running. In the y-axis number of active threads are from 0 to 3000 and in the x-axis elapsed time is from 0 - 12 minutes. This graph shows how many threads are hitting per second to the server and how 500 threads group joins after a specific time. In this way, a total of 3000 threads hit the server every second and it continuously hits for 12 minutes. The whole step load test has worked according to our plan just like other load tests. In Fig. 5. we can see the response time overtime of the EC2 model in the Linear Load test. In the graph, the x-axis presents elapsed time from 0 to 12 minutes and the y-axis presents response time in milliseconds. As the graph shows EC2 performs well in the Linear Load test and it has taken an average of 15000 milliseconds which means an average of 15 seconds Later it has taken only an average of 6000 to 9000 milliseconds which means 6-9 second to respond to all the HTTP requests. The

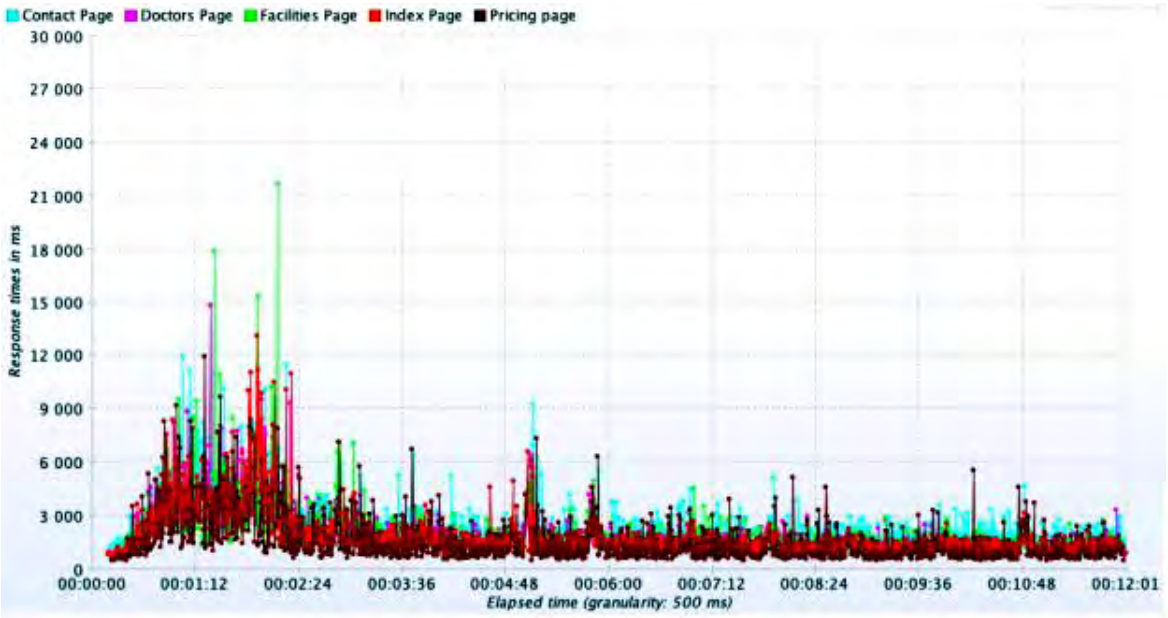


Figure 7.2: Response Time over Time Ec2 Linear Load

performance of EC2 in this test very consistent as it hadn't face any traffic storm and a consistent number of threads have been hit for a long time.

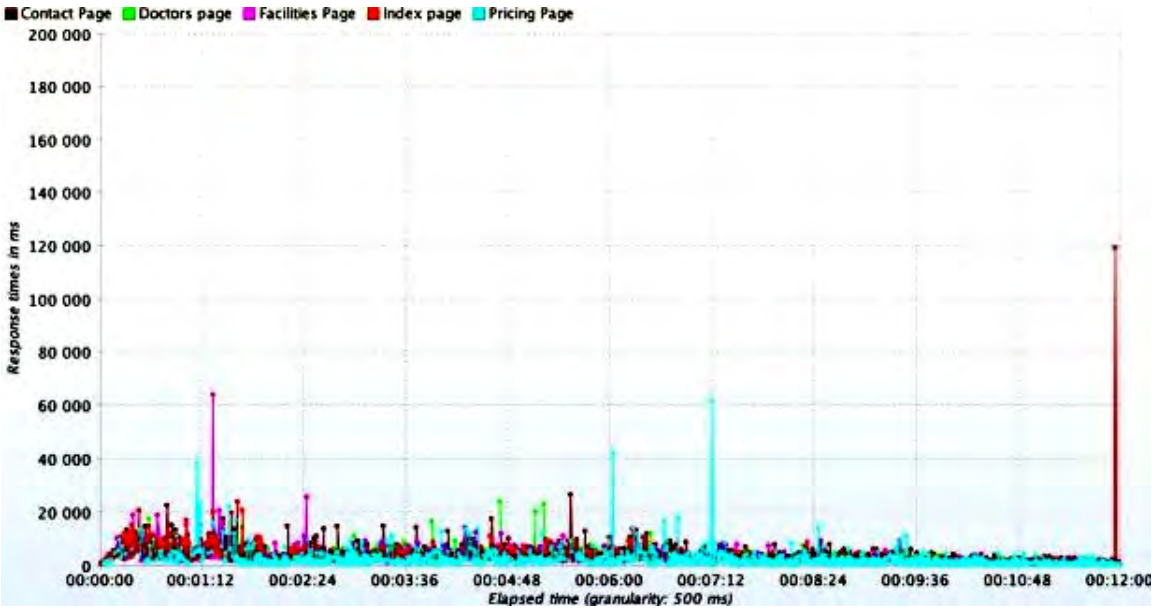


Figure 7.3: Response Time over Time Ec2 Step Load

In Fig. 6. we can see the response time overtime of the EC2 model in the Step Load test. In the graph, the x-axis presents elapsed time from 0 to 12 minutes and the y-axis presents response time in milliseconds. As the graph shows EC2 performs well in the Step Load test and it has taken an average of 20000 milliseconds which means an average of 20 seconds to respond to all the HTTP requests. Only in some moments when new chunks of threads have added server then it takes 40 to 60 seconds. But at one moment it takes 120 seconds to respond though it has happened only one time so we can overlook it. The performance of EC2 in this Step test very

consistent even though a specific number threads group consistently has been added and a huge number of threads have been hit for a long time.

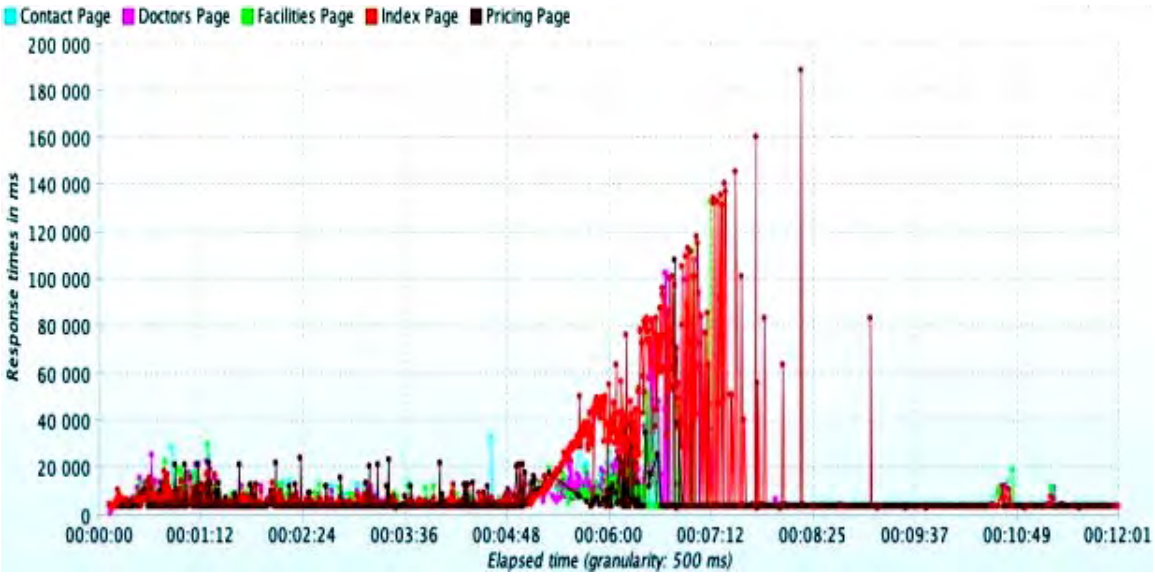


Figure 7.4: Response Time over Time Ec2 Peak Load

In the Fig. 7. EC2 performs well from the very beginning but when the is a traffic burst has been created and threads become 3000 from just 500 EC2 could not react well. From this part of the test, the peak load average response increase much higher which means the server has become slow now it has responded at average 120-140 seconds. The performance of EC2 in this Peak Load test very inconsistent as at the beginning it performed well but when a traffic storm took place the performance of the EC2 has been poor.

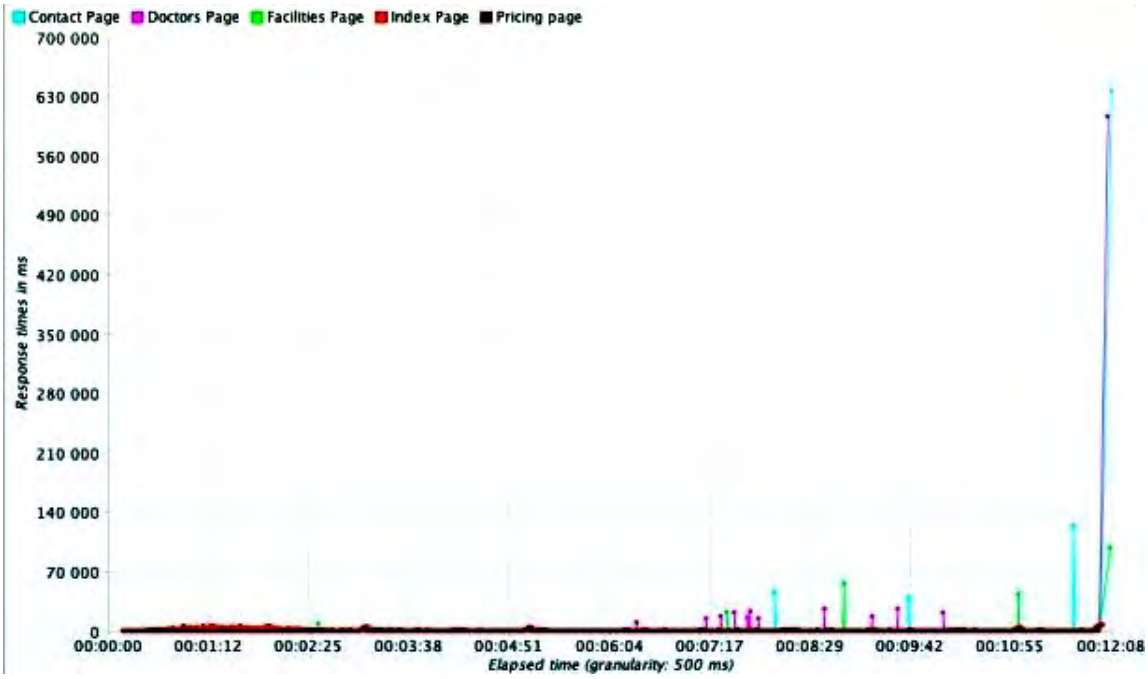


Figure 7.5: Response Time over Time ECS Linear Load

In Fig. 8. ECS is surprisingly performed much better than the EC2 model because

as we can see ECS-based webserver has taken almost 0 seconds to respond to the HTTP request whereas EC2 has taken average 0 seconds sometimes 40 to 60 seconds. The number of threads were always consistent so ECS had not to face any problem during the test and performed constantly and responded at almost 0 second in most of the time. We cannot expect better performance than this in any case say that ECS has given his best performance in a linear Load test.

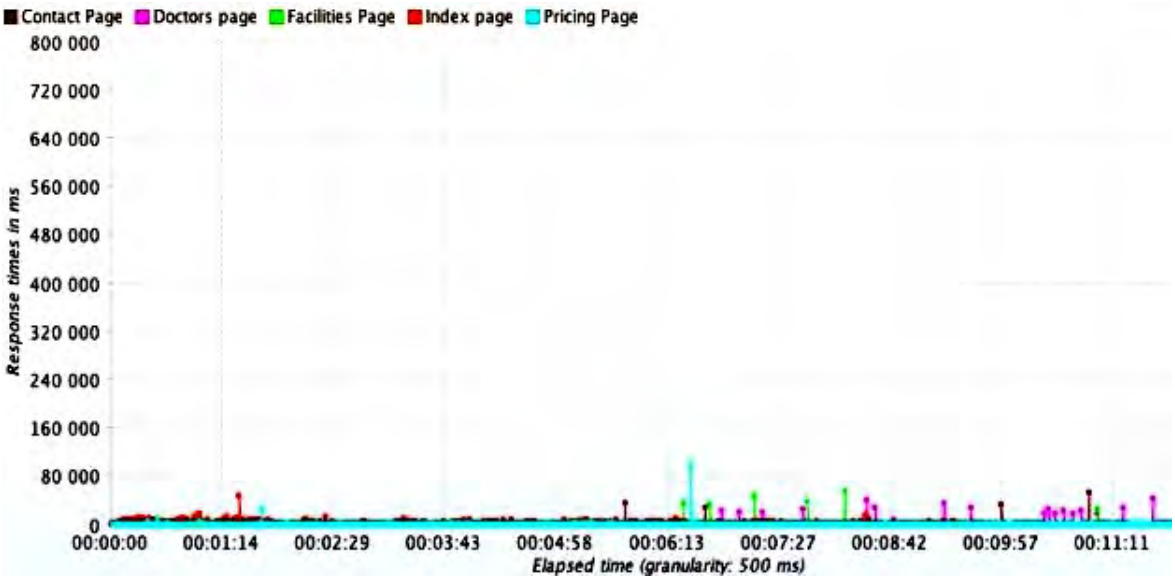


Figure 7.6: Response Time over Time ECS Step Load

In Fig. 9. ECS is likewise similarly performed much better than the EC2 model because as we can see ECS-based webserver has taken almost 0 seconds to respond to the HTTP request whereas EC2 has taken average 20 seconds sometimes 40 to 60. The number of threads was not always consistent and a specific number of threads group have been added but ECS did not create any problem during the test and performed constantly and responded at almost 0 seconds most of the time.ECS has given his one of the best performance in step Load test even though at a certain time it had to respond to 3000 threads at a time.
ECS performs significantly well in the Fig. 10. We can see the server was responding at an average time of almost 0 seconds but when a traffic burst has been created it gets slower and it was tough but still, they responded at an average of 10-20 seconds which is much lesser then EC2. Though at some point, it took more than 20 seconds for fewer responses but yet that is in very few times and still a better performance than EC2. The performance of ECS in this Peak Load test very inconsistent as at the beginning it performed well but when a traffic storm took place the performance of the ECS has been lower but better than EC2.
In Fig. 11. we can see the response time overtime of the Serverless model and we have used AWS Amplify service of the Serverless model in this Load test. In the graph, the x-axis presents elapsed time from 0 to 12 minutes and the y-axis presents response time in milliseconds. As the graph shows, Amplify performs well in the Linear Load test and it has taken an average of 2-6 seconds to respond to all the HTTP requests. The performance of Amplify in this test very powerful and consistent as it hadn't face any traffic storm and a consistent number of threads have been hit for a long time.

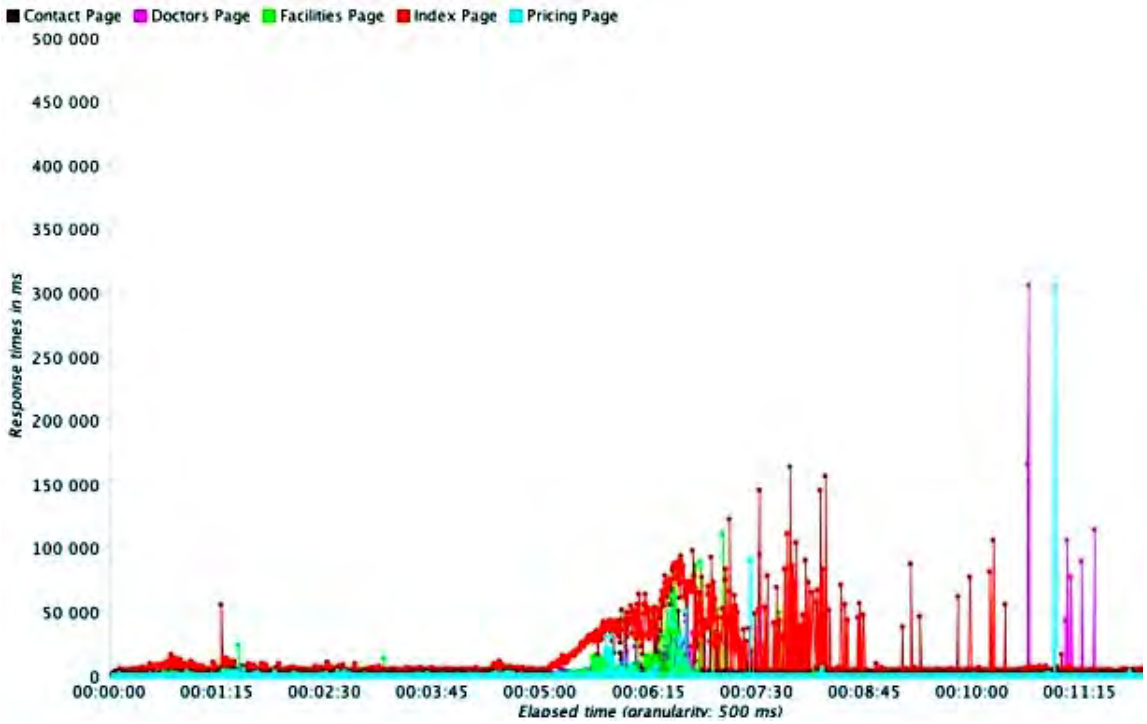


Figure 7.7: Response Time over Time ECS Peak Load

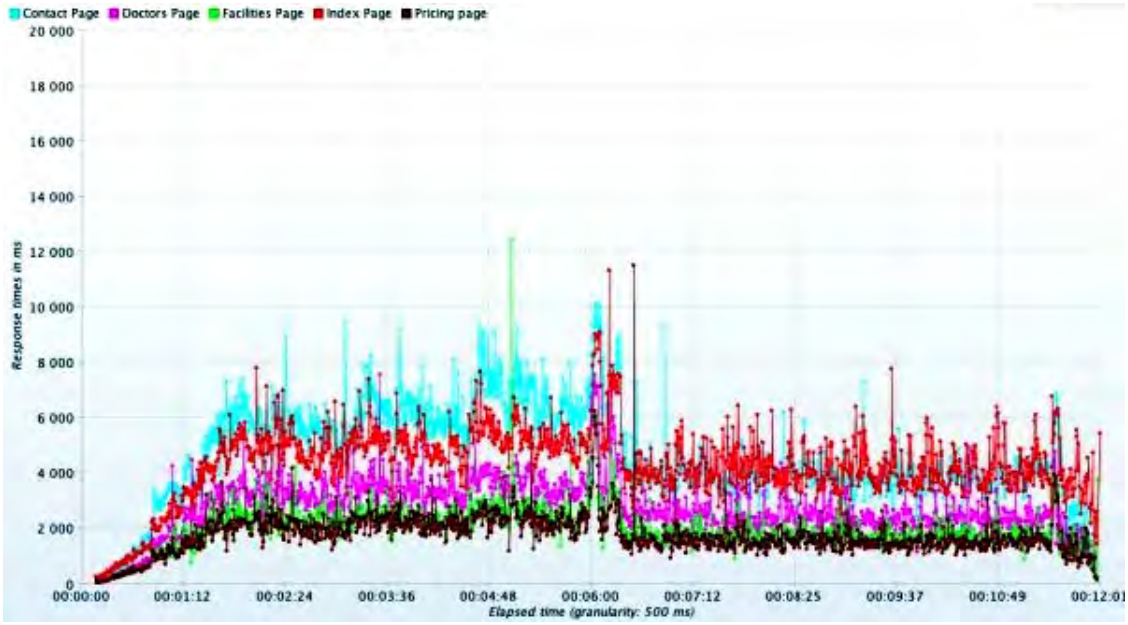


Figure 7.8: Response Time over Time Amplify Linear Load

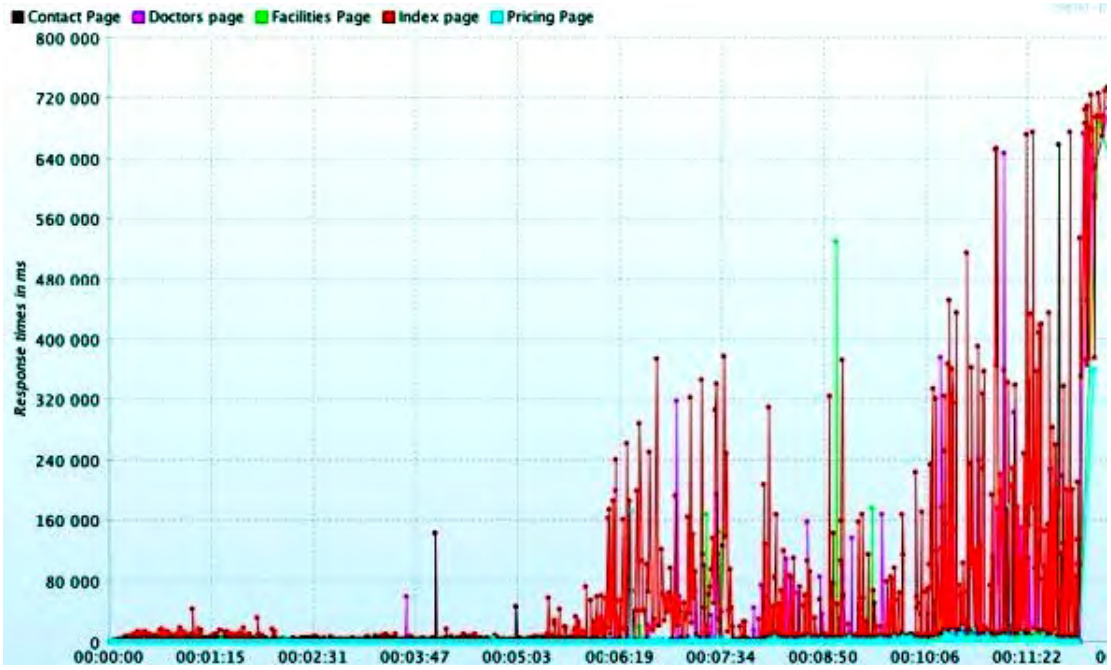


Figure 7.9: Response Time over Time Amplify Step Load

In Fig. 12. Amplify has performed impressively as we can see the Amplify-based webserver has taken almost 0 seconds to respond to the HTTP request in the first half of the test. The number of threads was not always consistent and a specific number of threads group have been added and Amplify controlled the performance for a certain period of time and responded at almost 0 seconds but after the first, it could not maintain the performance and started to take a huge time in most of the cases at the second half. Amplify has given average performance in step Load test even though at a certain time it had to respond to 3000 threads at a time.
In Fig. 13. Amplify has performed really impressive as we can see the Amplify-based webserver has taken almost 0 seconds to respond to the HTTP request in the first half of the test. But when a traffic burst has been created it gets slower Amplify could not control the performance and responded at almost 24-32 seconds. Amplify has given average performance in the Peak Load test also in terms of response time over time.

EC2 The table 2 below shows the data that we get from the Performance test. In the data we have presented Samples, Average, Error and Throughput.

Table 7.1: Performance Analysis Table for EC2

Model	*Label*	*Samples*	*Std. Dev.*	*Error(%)*	*Throughput*
EC2	Linear	33946	1753.47	0.00%	47.77
EC2	Step	29043	6946.18	0.01%	35.89
EC2	Peak	18337	18119.48	2.93%	25.41

In TABLE 2, we can see EC2 performs really well as the total response in the Linear load test is 33946, in Step load test is 29043 and in Peak load test is 18337 in the entire test time. In the Linear Load test, EC2 has created 0 error which is the best we can get from a model whereas in Step load EC2 has created 0.01 % error which is also almost 0 error. In the peak Load, EC2 created 2.93 % which is an average

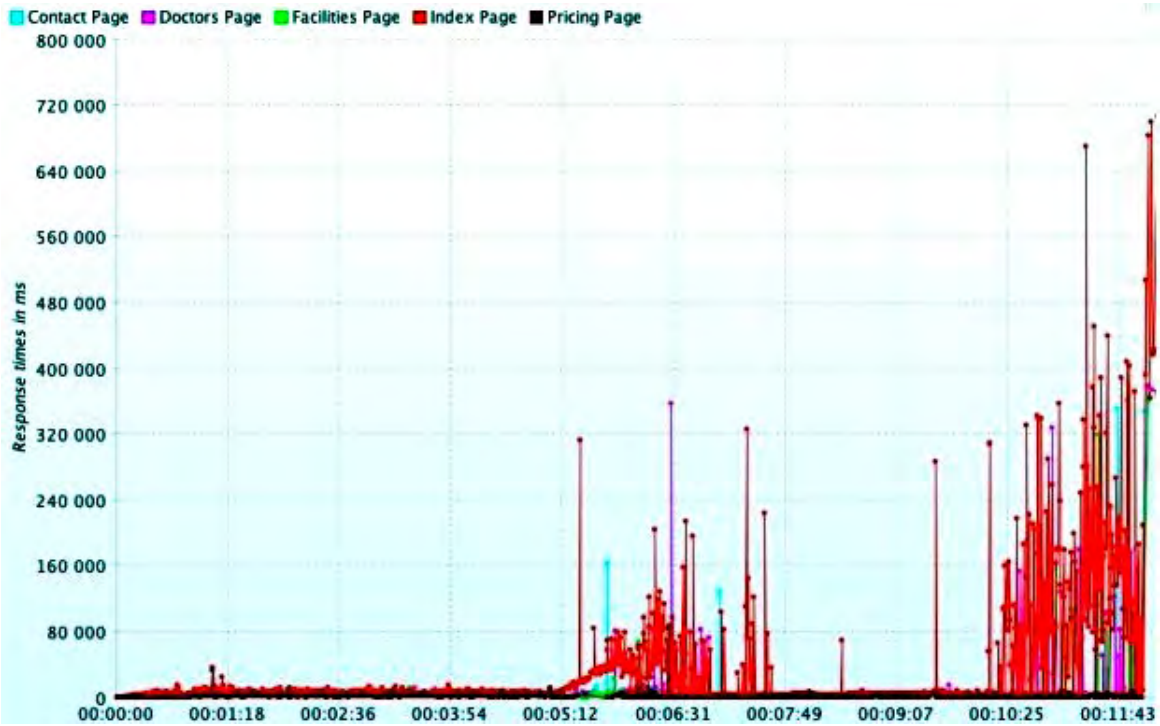


Figure 7.10: Response Time over Time Amplify Peak Load

performance. Standard deviation represents how many unusual cases were found which were deviating from the average value of the receiving time. The lesser the rate more consistent the time pattern is considered and here we have presented the average standard deviation of each load test. In the Linear load test, the standard deviation is 1753.47 which is not so high, in the Step load test the standard deviation is 6946.18, and in the Peak load test, it is 18119.48. In Throughput EC2 responded 47.77 requests per second in the Linear Load. In the Step Load, EC2 responded 35.89437 requests per second and lastly, in the Peak, Load EC2 responded 25.41 requests every second. Ec2 has given a decent performance in the throughput part.

ECS

Table 7.2: Performance Analysis Table for ECS

Model	***Label***	***Samples***	***Std. Dev.***	***Error(%)***	***Throughput***
ECS	Linear	78163	55677.03	1.05%	92.10
ECS	Step	64094	10449.18	2.22%	80.95
ECS	Peak	34003	165530.38	8.86%	34.46

In TABLE 3, we can see ECS performs really well as the total response in the Linear load test is 78163, in the Step load test is 64094 and in Peak load test is 34003 in the entire test time. So, in every load test, ECS has responded way better than EC2. In terms of error percentage, ECS has created 1.05% error in the Linear Load test whereas in Step load ECS has created 2.22% error. In the peak Load, ECS created 8.86% which is an average performance. So, in terms of error percentage, EC2 has done better than ECS. Though ECS has made a higher mistake but the amount of response it has created is way better which can not be overlooked. In the standard deviation part, the standard deviation of ECS is 55677.0 in the Linear load

test which is way higher than EC2, in the Step load test the standard deviation of ECS is 10449.18, and in the Peak load test, it is 165530.38. So EC2 has performed way better as it had a lesser standard deviation. In the Throughput part, ECS responded 92.10 requests per second in the Linear Load. In the Step Load, ECS responded 80.95 requests per second and lastly, in the Peak, Load ECS responded 34.46 requests every second. ECS has given a better performance than EC2 in the throughput part.

Serverless

Table 7.3: Performance Analysis Table for Amplify

Model	***Label***	***Samples***	***Std. Dev.***	***Error(%)***	***Throughput***
Amplify	Linear	143984	7411.56	1.26%	201.35
Amplify	Step	106118	1872.48	0.00%	149.15
Amplify	Peak	23272	60844.68	4.06%	31.03

In TABLE 4, we can see AWS Amplify service of serverless model performs really well as the total response in the Linear load test is 143984, in the Step load test is 106118 and in Peak load test is 23272 in the entire test time. So, in Linear and step load test, Amplify has responded way better than ECS whereas ECS has done better in peak load test. In terms of error percentage, Amplify has created 1.26% error in the Linear Load test whereas in Step load Amplify has created 0.00% error. In the peak Load, Amplify has created 4.06% which is better performance than ECS but still EC2 did better here. So, in terms of error percentage, EC2 has done better than ECS and amplify. Though Amplify has made a higher mistake but the amount of response it has created is way better which can not be overlooked. In the standard deviation part, the standard deviation of Amplify is 7411.56 in the Linear load test which is way better than ECS, in the Step load test the standard deviation of Amplify is 1872.48, and in the Peak load test, it is 60844.68. So EC2 has still better performance as it had a lesser standard deviation. In the Throughput part, Amplify responded 201.35 requests per second in the Linear Load. In the Step Load, Amplify responded 149.15 requests per second and lastly, in the Peak Load, 31.03 requests every second. Amplify has given a way better performance than ECS in the Linear and step load test and ECS has done better in peak load test at throughput part.

Compare

Table 7.4: Performance Comparison Table

Dep Modell	***Error (%)***	***Std. Dev. Avg***	***Throughput Avg***
EC2	1.00%	8939.71	109.08
ECS	4.04%	77218.86	207.51
Serverless	1.77%	23364.24	381.53

Now, if we consider the Fig. 7.11, Fig. 7.12, Fig. 7.13 and Table 7.4 the performance of the amount of request Amplify service of Serverless model has been responded to is higher than ECS and way higher than EC2. So, if we need a model that can give a faster response to the HTTP request then we should use a serverless model. If we consider the performance in error percentage, the EC2 service of the VM model is better than Amplify and way better than ECS. So, if we need the lowest error-prone

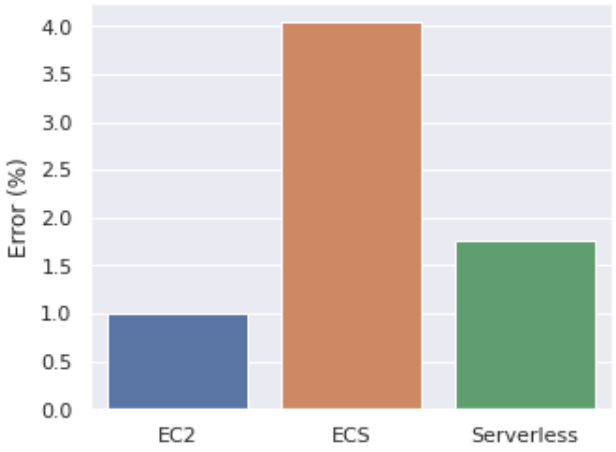


Figure 7.11: Error Comparison

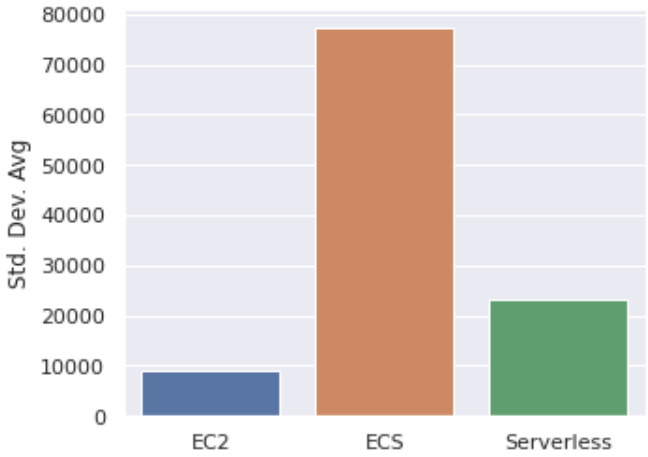


Figure 7.12: Standard Deviation Comparison

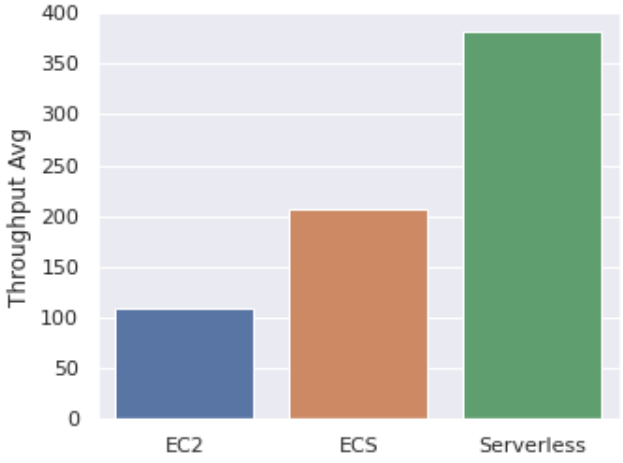


Figure 7.13: Throughput Comparison

model which can create the lowest error while responding to the HTTP request then we should use the EC2 service of the VM model. In the standard deviation part, again EC2 service of the VM model has given way better performance than ECS and Amplify. So if we want to choose a model on the basis of standard deviation, we should select the EC2 service of the VM model. Lastly, in the throughput part, Amplify performs significantly better than EC2 and way better then ECS. If we want to select a model that can respond to the highest number of HTTP request in every second then we must select Amplify service of serverless model.

Chapter 8

Conclusion and Future Work

8.1 Research Overview

In this paper, we have presented a comparative study of three native cloud application deployment models. We have developed the infrastructure for those models and deployed our healthcare application there. We have lead a comparison test that how these three models perform by using the Apache JMeter tool. Additionally, to make the performance test more realistic we have utilized a three-step based test plan. The parameters of our comparison test were HTTP response, error percentage, standard deviation, and throughput. Serverless performed essentially better than Docker container and Virtual Machine in the total HTTP response and throughput part although Docker Container did better than Virtual Machine. And surprisingly, Virtual Machine did better than Docker Container and Serverless in the error percentage and standard deviation part.

8.2 Recommendation and Future Work

As we have no clear information about the performances of the models, our paper can recommend any user based on their preference. So the user should have the clear knowledge about what type of deployment they want, what are the benefits they are looking for. If they want to use a model where better throughput is important more than anything else then Serverless should win the race. If the error percentage is the main concern then the Virtual machine. Similarly, for standard deviation Virtual Machine is the best choice and for the highest HTTP response, it should be Serverless. But if the customers want a model who can be average or good in many of these parameters then the answer should be Docker Container. In our paper, we have conducted a performance test by deploying a Static website and this performance test will be more interesting if we can do this with a dynamic website. Then, in the serverless part, we will be able to use the Amazon Lambda service. Moreover, as per our belief bringing a dynamic website for testing will create more impact on the performance test. We have further interest in introducing the dynamic website and Amazon Lambda service in our analysis. And also deploy the dynamic website into those models to see how the model performs.

Bibliography

[1] R. K. Barik, R. K. Lenka, K. R. Rao and D. Ghose, "Performance analysis of virtual machines and containers in cloud computing," 2016 International Conference on Computing, Communication and Automation (ICCCA), Noida, 2016, pp. 1204-1210.

[2] A. M. Joy, "Performance comparison between Linux containers and virtual machines," 2015 International Conference on Advances in Computer Engineering and Applications, Ghaziabad, 2015, pp. 342-346.

[3] Islam, M.M., Khan, Z. and Alsaawy, Y. A framework for harmonizing internet of things (IoT) in cloud: analyses and implementation. Wireless Netw (2019).

[4] H. Zeng, B. Wang, W. Deng and W. Zhang, "Measurement and Evaluation for Docker Container Networking," 2017 International Conference on Cyber-Enabled Distributed Computing and Knowledge Discovery (CyberC), Nanjing, 2017, pp. 105-108.

[5] A. K. Yadav, M. L. Garg, Ritika, "Docker Containers Versus Virtual Machine-Based Virtualization," 2018 International Conference on Emerging Technologies in Data Mining and Information Security (IEMIS), Kolkata, 2018, pp. 141-150.

[6] B. Wang, Y. Song, X. Cui and J. Cao, "Performance comparison between hypervisor- and container-based virtualizations for cloud users," 2017 4th International Conference on Systems and Informatics (ICSAI), Hangzhou, 2017, pp. 684-689.

[7] T. Salah, M. J. Zemerly, C. Y. Yeun, M. Al-Qutayri and Y. Al-Hammadi, "Performance comparison between container-based and VM-based services," 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN), Paris, 2017, pp. 185-190.

[8] B. Ruan, H. Huang, S. Wu and H. Jin, "A Performance Study of Containers in Cloud Environment," 2016 Asia-Pacific Services Computing Conference (APSCC), Zhangjiajie, 2016, pp. 343-356.

[9] D. Liu and L. Zhao, "The research and implementation of cloud computing platform based on docker," 2014 11th International Computer Conference on Wavelet Actiev Media Technology and Information Processing(ICCWAMTIP), Chengdu, 2014, pp. 475-478.

[10] C. Qu, R. N. Calheiros and R. Buyya, "Auto-scaling web applications in clouds: A taxonomy and survey", ACM Comput. Surv., vol. 51, no. 4, Sep. 2018.

[11] O. M. Hasan, K. R. Ahmed, M. M. Islam, "Parking Recommender System Using Q-Learning and Cloud Computing," 2020 International Conference on Cyber Security and Computer Science (ICONCS), Dhaka, 2020, pp. 607-618.

[12] M. M. Hassan, B. Song, A. Almogren, M. S. Hossain, A. Alamri, M. Alnuem, M. M. Monowar and M. A. Hossain, "Efficient Virtual Machine Resource Management for Media Cloud Computing," KSII Transactions on Internet and Information Systems, vol. 8, no. 5, pp. 1567-1587, 2014.

[13] Medium Homepage, www.medium.com, last accessed on 30 August, 2020

[14] AWS Homepage, www.aws.amazon.com, last accessed on 30 August, 2020

[15] DZone Homepage, www.dzone.com, last accessed on 30 August, 2020

[16] TensorFlow Homepage, https://www.tensorflow.org/, [last accessed on 15 February,2019]

[17] Keras Homepage, https://keras.io/, [last accessed on 11 March, 2019]

[18] MIT Deep Learning Homepage, https://deeplearning.mit.edu, [last accessed on 10 January, 2019]

[19] H. Lee, K. Satyam and G. Fox, "Evaluation of Production Serverless Computing Environments," 2018 IEEE 11th International Conference on Cloud Computing (CLOUD), San Francisco, CA, 2018, pp. 442-450.

[20] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D. Popa, "Firecracker: Lightweight Virtualization for Serverless Applications," 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20), Santa Clara, CA, 2020, pp. 419-434.

[21] M. Villamizar et al., "Infrastructure Cost Comparison of Running Web Applications in the Cloud Using AWS Lambda and Monolithic and Microservice Architectures," 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), Cartagena, 2016, pp. 179-182.

[22] A. Kanso and A. Youssef,"Serverless: beyond the cloud," 2017 Proceedings of the 2nd International Workshop on Serverless Computing (WoSC '17), New York, USA, 2017, pp.6–10.

[23] M. Islam, Z. Khan, Y. Alsaawy"An Implementation of Harmonizing Internet of Things in Cloud", Sensor Grid Internet of Things (SGIoT), Niagara Falls, Canada, 2018, pp 3-12.

[24] MIT Deep Learning Homepage, https://deeplearning.mit.edu, [last accessed on 10 January, 2019]

[25] MIT Deep Learning Homepage, https://deeplearning.mit.edu, [last accessed on 10 January, 2019]

[26] Islam, M.M., Zaheer Khan, Yazed Alsaawy "An Implementation of Harmonizing Internet of Things in Cloud", 2018 Sensor Grid Internet of Things (SGIoT), 2018, Niagara Falls, Canada.

[27] S. Brunner, M. Blöchlinger, G. Toffetti, J. Spillner and T. M. Bohnert, "Experimental Evaluation of the Cloud-Native Application Design," 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC), Limassol, 2015, pp. 488-493.

[28] A. Wahid and M. T. Banday, "Machine Type Comparative of Leading Cloud Players Based on Performance & Pricing," 2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI), Bangalore, 2018, pp. 2364-2368.

[29] T. Lynn, P. Rosati, A. Lejeune and V. Emeakaroha, "A Preliminary Review of Enterprise Serverless Cloud Computing (Function-as-a-Service) Platforms," 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), Hong Kong, 2017, pp. 162-169.

[30] A. Abuabdo and Z. A. Al-Sharif, "Virtualization vs. Containerization: Towards a Multithreaded Performance Evaluation Approach," 2019 IEEE/ACS 16th International Conference on Computer Systems and Applications (AICCSA), Abu Dhabi, United Arab Emirates, 2019, pp. 1-6.

[31] C. Kotas, T. Naughton and N. Imam, "A comparison of Amazon Web Services and Microsoft Azure cloud platforms for high performance computing," 2018 IEEE International Conference on Consumer Electronics (ICCE), Las Vegas, NV, 2018, pp. 1-4.

[32] Y. P. Baginda, A. Affandi and I. Pratomo, "Analysis of RTO and RPO of a Service Stored on Amazon Web Service (AWS) and Google Cloud Engine (GCE)," 2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE), Kuta, 2018, pp. 418-422.

[33] J. Manner, M. Endreß, T. Heckel and G. Wirtz, "Cold Start Influencing Factors in Function as a Service," 2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion), Zurich, 2018, pp. 181-188.

[34] A. Deese, "Implementation of Unsupervised k-Means Clustering Algorithm Within Amazon Web Services Lambda," 2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID), Washington, DC, 2018, pp. 626-632.

[35] H. Wang, D. Niu and B. Li, "Distributed Machine Learning with a Serverless Architecture," IEEE INFOCOM 2019 - IEEE Conference on Computer Communications, Paris, France, 2019, pp. 1288-1296.

[36] W. Felter, A. Ferreira, R. Rajamony and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, 2015, pp. 171-172.

[37] B. Cheng, J. Fuerst, G. Solmaz and T. Sanada, "Fog Function: Serverless Fog Computing for Data Intensive IoT Services," 2019 IEEE International Conference on Services Computing (SCC), Milan, Italy, 2019, pp. 28-35.

[38] A. Gopalasingham, D. G. Herculea, C. S. Chen and L. Roullet, "Virtualization of radio access network by Virtual Machine and Docker: Practice and performance analysis," 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), Lisbon, 2017, pp. 680-685.

[39] J. Kim, T. J. Jun, D. Kang, D. Kim and D. Kim, "GPU Enabled Serverless Computing Framework," 2018 26th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), Cambridge, 2018, pp. 533-540.

[40] T. Kämäräinen, Y. Shan, M. Siekkinen and A. Ylä-Jääski, "Virtual machines vs. containers in cloud gaming systems," 2015 International Workshop on Network and Systems Support for Games (NetGames), Zagreb, 2015, pp. 1-6.