CSE 546 — Project Report

Aakash Patel	*Disha Bhukte*	*Sapan Desai*
1219522499	*1219495225*	*1219080070*

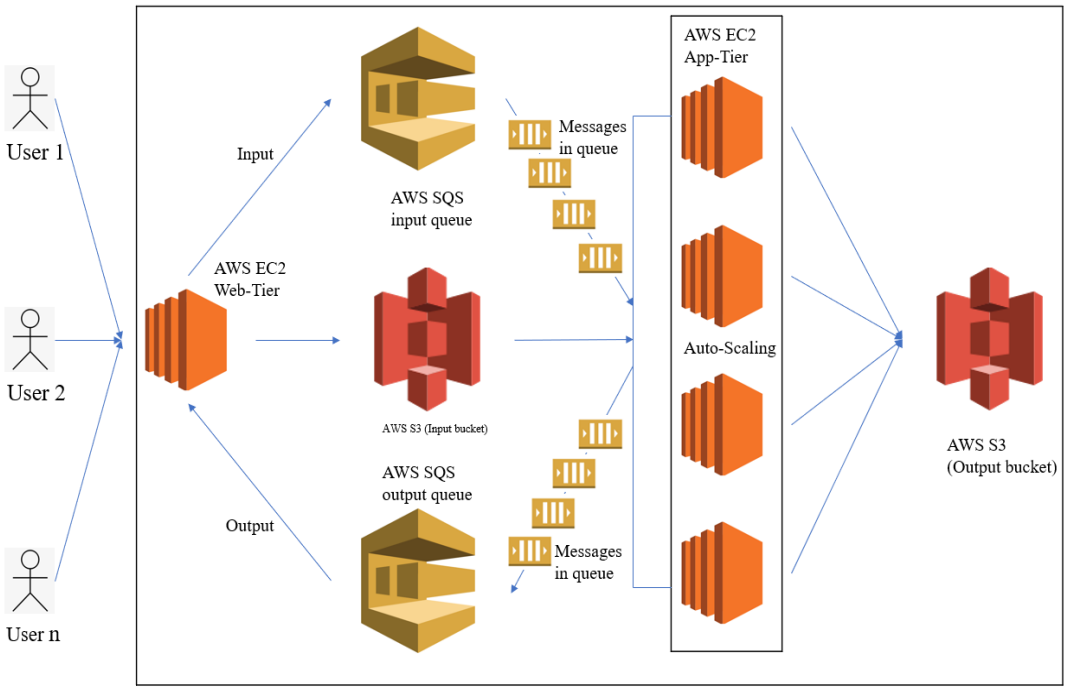
1. Problem statement

Build an elastic application using an IaaS cloud, preferably using the IaaS resources from Amazon Web Services (AWS). This cloud app will provide image recognition service to the users by performing deep learning on the images that they provide based on the pre-trained model provided to us. Following are the requirements of the cloud app:-

1) The app should perform image recognition on the images provided by the user and return the result using the provided deep learning model. Input is .png and the output is the predicted result.
2) Provided AWS image details :-
   ID: ami-0ee8cf7b8a34448a6, Name: app-tier, Region: us-east-1
3) The application should be able to handle multiple requests concurrently. It should scale in and scale out based on the number of requests.
4) All the input and outputs should be stored in S3.
5) The application is expected to handle requests as fast as possible with correct results without missing any request.

2. Design and implementation

2.1 Architecture



The application has been developed using Spring Boot. There are primarily two modules in the project which are as follows:-

- Web Tier (web-app)

  The web tier provides a user friendly UI for the user to upload images. It then stores the image in the S3 input bucket and puts a message in the input queue for further processing. Once the image recognition is done, the webapp reads the output from the output queue and displays the result on the UI. In case of multiple requests, it automatically initiates multiple app instances for faster processing of the multiple requests. The maximum count of the app instances initiated through the web app is 19.

- App Tier (app-tier_consumer)

  The app tier is used for processing the input request. Each app instance reads the message from the input queue, downloads the image from the S3 input bucket based on the key for the image present in the message of the input queue. It then recognizes the image using the provided image recognition model. The output is stored in the S3 output bucket and the input output pair is passed in the message to the output queue. When the number of requests in the input queue exceeds the maximum number of app instances created by the web tier i.e. 19, the app instance uses multithreading for handling the excess load of requests.

2.1.1 Amazon Services Used

- **Amazon EC2**: The EC2 instances have played a major role in deploying the project. The EC2 instances were used in App-tier instances and Web-tier instances. EC2 instance helped to fetch the data from the user, process the data and provide the output to the user. EC2 has multiple functionality as in the application-tier it was used to implement the auto-scaling process and it was the only service that had been linked to each other and used throughout the project. As a result Amazon EC2 was the backbone structure of the project.
- **Amazon SQS**: There are two Amazon SQS implemented in this project. First is an input queue where the images come from the Web-tier and are passed to the App-tier while the output queue works in the opposite way where it takes the images and results from the App-tier and passes to Web-tier. The main usage of Amazon SQS is done to implement the auto-scaling concept in the project. As with the limit of only 19 EC2 App-tier instances, the Amazon SQS helps to autoscale according to the requirement and only allows 19 instances to work at a time.
- **Amazon S3**: There are two Amazon S3 buckets used throughout the project. First is an input bucket where the images from the web-tier, uploaded by the user, are directly stored. Later the output bucket is where the image name and the result of the classification is stored. The images are first processed in the app tier and later forwarded to the output bucket.

2.1.2 Multithreading

There are 3 key areas where multithreading has been used in the application. Firstly, the web app continuously monitors the input queue so as to initiate new app instances if needed. Secondly, the running app instances which continuously monitor the input queue for reading the messages in it, serve the user request and classify the user images, put it in the S3 bucket and output queue. Lastly, the web tier continuously monitors the output queue and maintains a hashmap so as to temporarily store the result and return the result to the same user who requested the image.

2.1.3 New AMI

On initiation of a new app instance, we expected the instance to be created using the AMI already provided to us with the app tier jar. This involved first creating the instance using the provided image and then transferring the app tier jar file to the instance along with running a few commands so as to get the instance functioning for serving the requests. This process would be time consuming if handled through code or done manually. So we decided to create a new Image from the existing running EC2 app instance and used User Data feature to run our app tier jar file. So when it comes to final deployment, we won't have to deploy the app tier explicitly on EC2 instances. We need to just deploy the Web tier on EC2 instances and it will automatically start the App tier instances using this image.

2.1.4 FIFO Queue

There are two variants of SQS offered by AWS. One is the standard queue and the other is FIFO queue. The standard queue delivers the message at least once. This increases the chances of introduction of duplicate messages which will in turn lead to wasting multiple requests to S3 buckets and output queues which adds on to the cost of the project . In case of FIFO queue, the messages are delivered exactly once. Therefore, duplicates are not introduced in this case.Our application has this specific requirement. However, if we use the same message group id for the messages, the FIFO queue will process the next request only after the previous request has been processed which does not happen in the standard queue. We needed a queue which would be a combination of both standard queue and FIFO queue. So, we made the FIFO queue work as a standard queue for us. This can be achieved using the different message group id for each message.

In the FIFO queue, we have also used content based deduplication by setting its flag which will avoid duplication of any user request.

2.1.5 Termination Protection Policy For Web Tier

For termination protection, we made sure that the web instance is not shut down once all the instances are terminated until we shut it down explicitly. It will keep on running throughout the whole project time because the DisableApiTermination was set to false. To prevent your instance from being accidentally terminated, we can enable termination protection for the instance. If termination protection is enabled for the instance, then it cannot be terminated using the EC2 console, APIs, or AWS command line tools.

2.1.6 IAM User

We created an IAM user, used credentials to work with AWS. With the help of IAM users, we improved the security of the users and it was easy to share the information platform throughout the project. All the processes were taking place concurrently and admins were able to monitor different tasks at a time.

2.2 Autoscaling

The scale in and scale out has been handled by creating new app instances and by using multi-threading. The app instances are triggered automatically from the web tier and the multi-threading is handled by the app tier.

In the web tier, the application continuously monitors the number of requests in the input queue and the number of running app instances. When a situation comes wherein the number of instances are less than the number of app instances, the web tier automatically initiates the app instances as per the demand. For instance, if the input queue has 5 requests and there is only 1 app instance running, the web tier will automatically create 4 more instances to serve the requests in the input queue.

In the app tier, the application heavily relies on multithreading for serving the increasing demand of the requests. When the number of input requests in the input queue exceeds the maximum count of app instances i.e. 19, the app instance divides the total number of requests by the number of app instances running and spawns the required number of threads. For instance, if there are 100 requests in the SQS and there are already 19 running app instances, each app instance will divide the number of visible messages by 19 and spawn the required number of threads.

When there are not enough messages to be consumed by the app instances, the app instance terminates itself. This is how the scaling down of the app instances is handled.

3. Testing and evaluation

Test Case	output
User request pushed into SQS Input queue	The requests are pushed into the SQS input queue and later ready to go to the output queue via messages in flight.
Input image stored in S3 input bucket	The image was stored in the S3 input bucket
App instance initiated on sending image as input	The app instance was able to successfully send the image to the S3 bucket.
App instances scaled up when number of images are below or equal or greater than 19	The app instances were scaling according to the number of messages in the input queue. If there were more than 19 messages (images) uploaded at once, there will be maximum (19) EC2 instances working at the same time. While when lowering down the demand of the image process, the EC2 instance scales down according to the needs.
App instances downloading images from S3 Input Bucket	The app instance was able to download images from the S3 input bucket and then run its process on the EC2 instance.
Output of image classification stored in S3 Output Bucket	After processing the classification of the image, the result is stored in the output bucket
Result of image classification pushed to SQS Output Queue	The result of the classification was pushed to the SQS output queue in the form of text.

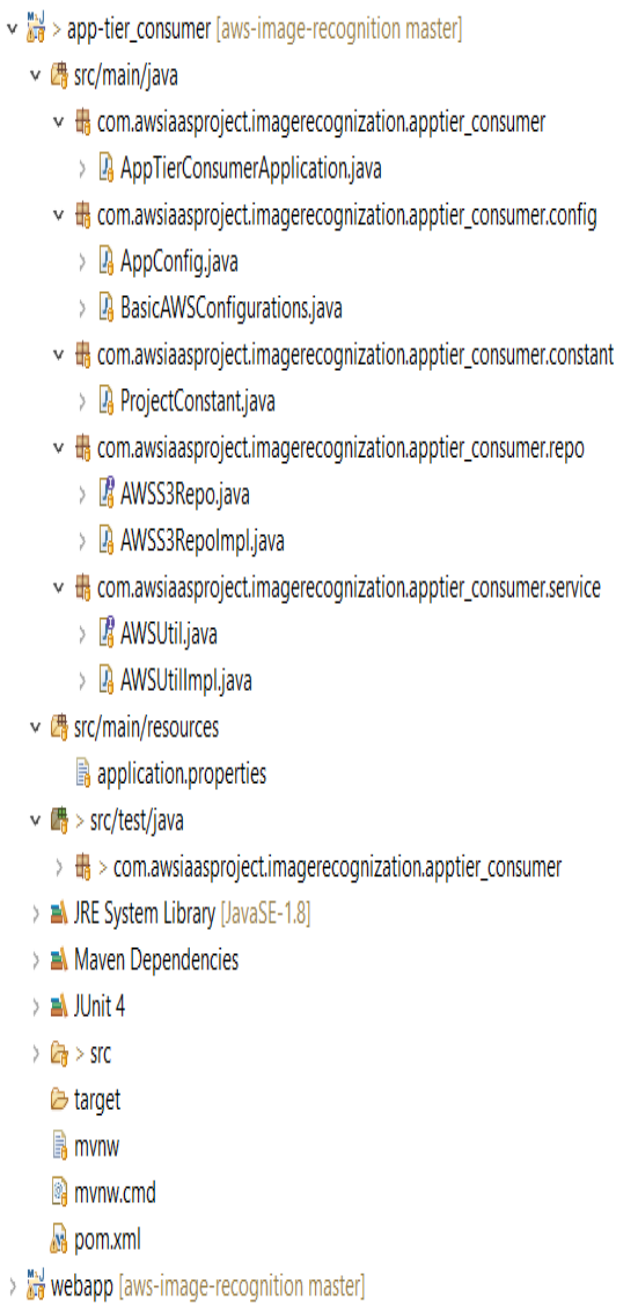
Image classification result displayed for 1 image input	The classification result for image 1 was displayed in the form of a table.
Image classification result displayed for multiple image inputs	The classification result for multiple images were displayed on the front end.
Image Classification result displayed for 2 users submitting images simultaneously	The image classifier was able to concurrently process the request of multiple users.

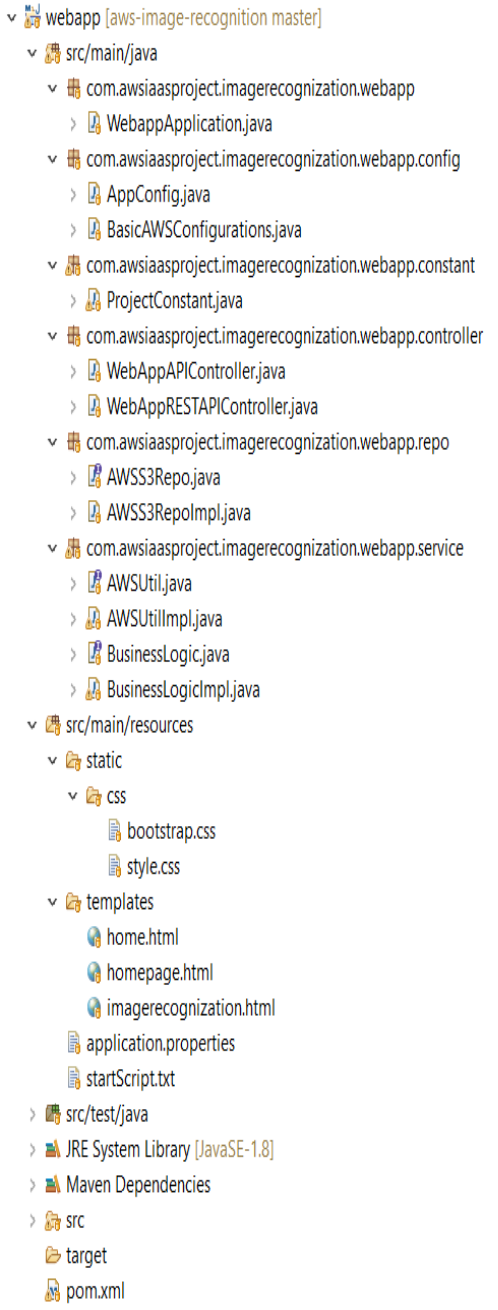
4. Code

Code file name	**Explanation**
Web App	The files mentioned below are the content of the sub folder, web App.
WebappApplication.java	This is the starting point of the web instance and it configures all the beans. It will initialise all the objects in the whole file.
AppConfig.java	The configuration class that produces beans necessary for the web tier to run. All these beans are handled by IoC controller and configured when the application starts, making the application loosely coupled as it injects dependency automatically. As Spring beans can be reusable these objects are created only once during the start of the application where our Spring IoC is initialized and configured for all the beans.
BasicAWSConfigurations.java	This java file configures all the AWS services that are going to be used in the project. Here all configurations supply bean metadata to Spring IoC controller
ProjectConstant.java	This java file consists of all the constants present throughout the project. The project constants are called from this file throughout the project. It helps to keep the code clean and if there are any variable changes, we only

	have to change in this file rather than changing all the variables.
WebAppAPIController.java	All the APIs are defined here that provide necessary services to run the application. This is the backbone of the application.
AWSS3Repo.java	In this file all the methods are defined that can be implemented by another class to upload a file to S3.
AWSS3RepoImpl.java	The logic of uploading the images to S3 is written here.
AWSUtil.java	In this class all the services are defined such as create and run instances that will be implemented in AWSUtilImpl.java
AWSUtilImpl.java	This class implements all the method of AWSUtil and defines specific behaviour of each method that is required to run the application
BusinessLogic.java	In this file all the methods are defined that can be implemented by businesslogicImpl.java
BusinessLogicImpl.java	This java file contains the logic to implement the whole project and provide the output to the user. It mainly focuses on the structure of the project and acts as a backbone of the project.
bootstrap.css	This is the cascading style sheet of the html file. IT is a CSS framework for developing responsive web pages.
style.css	This is the cascading style sheet of the html file. It contains all the presentation, dimensions, and style of a web page that helps to make the user experience easier. The cascading style sheet describes the presentation of the HTML elements.
imagerecognization.jsp	This is the homepage of the web page where

	a user can upload the image.
result.jsp	This is the final page of the front end. Once the images are processed, it will be displayed later on this page. The user can find all the output data in the form of a table on this page.
pom.xml	pom.xml also known as Project Object Model .xml


app-tier_consumer [aws-image-recognition master]
src/main/java
com.awsiaasproject.imagerecognization.apptier_consumer
AppTierConsumerApplication.java
com.awsiaasproject.imagerecognization.apptier_consumer.config
AppConfig.java
BasicAWSConfigurations.java
com.awsiaasproject.imagerecognization.apptier_consumer.constant
ProjectConstant.java
com.awsiaasproject.imagerecognization.apptier_consumer.repo
AWSS3Repo.java
AWSS3RepoImpl.java
com.awsiaasproject.imagerecognization.apptier_consumer.service
AWSUtil.java
AWSUtilImpl.java
src/main/resources
application.properties
src/test/java
com.awsiaasproject.imagerecognization.apptier_consumer
JRE System Library [JavaSE-1.8]
Maven Dependencies
JUnit 4
src
target
mvnw
mvnw.cmd
pom.xml
webapp [aws-image-recognition master]


webapp [aws-image-recognition master]
src/main/java
com.awsiaasproject.imagerecognization.webapp
WebappApplication.java
com.awsiaasproject.imagerecognization.webapp.config
AppConfig.java
BasicAWSConfigurations.java
com.awsiaasproject.imagerecognization.webapp.constant
ProjectConstant.java
com.awsiaasproject.imagerecognization.webapp.controller
WebAppAPIController.java
WebAppRESTAPIController.java
com.awsiaasproject.imagerecognization.webapp.repo
AWSS3Repo.java
AWSS3RepoImpl.java
com.awsiaasproject.imagerecognization.webapp.service
AWSUtil.java
AWSUtilImpl.java
BusinessLogic.java
BusinessLogicImpl.java
src/main/resources
static
css
bootstrap.css
style.css
templates
home.html
homepage.html
imagerecognization.html
application.properties
startScript.txt
src/test/java
JRE System Library [JavaSE-1.8]
Maven Dependencies
src
target
pom.xml

Steps to run and install the code:-

1. Git clone: **git clone** is a **Git** command line utility which is used to target an existing repository and create a **clone**, or copy of the target repository. With the help of git clone, we can download the project that is uploaded in the private repository.

   git clone https://github.com/disha-bhukte/aws-image-recognition.git

2. Import as maven project: Once the project is downloaded in the directory, import the project as a maven project in the eclipse workspace.
3. maven clean: In order to run the project, the first step is to maven clean the file.
4. maven install: The next step is to maven install on the project.
5. Change the ACCESS_ID, ACCESS_KEY as per your AWS set up in both modules(web-tier and app-tier).
6. Hit Maven-install again.
7. Create an instance from the provided AMI.
8. Use scp command to transfer the app-tier jar file to the instance created in the previous step.

   scp -i security_key.pem /path_of_app_tier_jar ubuntu@ec2-public-dns:~

9. Create a new AMI after transferring the jar to an instance.
10. Copy the AMI ID created in the previous step and paste it under the AMI_ID constant in ProjectConstant.java file for the web-tier module.
11. Hit maven-install again to update the jar of web-tier.
12. Use scp command to transfer the jar file to the instance:

    scp -i security_key.pem /path_of_web_tier_jar ubuntu@ec2-public-dns:~

13. login to the web-tier instance using ssh command

    ssh -i ccproject2021.pem ec2-user@ec2-100-27-12-166.compute-1.amazonaws.com

14. setup java environment in the instance

    yum list java*
    sudo yum install java-1.8.0

15. change the permissions of the jar

    chmod +x <jarname>

16. run the web jar file

    java -jar <path to jar file>

5. Individual contributions (optional)

Aakash Patel (ASU ID:- 1219522499)

I. Design

The whole architecture was actively discussed by all the team members. There were a number of small features that we had to implement so as to get the application running. I have majorly contributed to two of the features that were thoroughly used throughout the project. They are autoscaling, creating and stopping EC2 instances.

II. Implementation

There were a few challenges that had to be handled while designing and implementing the auto scaling module. Firstly, the app instances were to be scaled till the maximum count of 19. As the free tier can allow a maximum of 20 EC2 instances at a time, I had to make sure to design the autoscaling with that parameter. Secondly, they were to be scaled up and scaled down on demand. When a user sends 10 images, only 10 EC2 app instances have to run at a time. While, when a user uploads 100 images, it can only run 19 Application instances and run concurrently to provide the output. I have implemented the code for auto scaling the app instance from the web instance using multithreading. The reason for using thread to perform autoscaling is because it is an application controller to handle load and we have implemented a controller in the web tier instance itself in order to modularize our code. Another reason is we have used Spring boot for our project which has a Tomcat server, and Tomcat can serve upto 10000 HTTP requests parallelly. And sometimes while serving requests it can interfere with our main thread of the web tier thus decreasing the performance of the application. So to resolve this I have kept a separate thread that runs the controller without interfering

with the main thread and Tomcat can smoothly handle all the requests from concurrent users. I have also implemented the utilities related to the EC2 instances such as configuration,creation, and stopping of the EC2 instances. I have also given the idea to my team member to format the input request in order to serve concurrent users requests. My team members have nailed it and now we were able to serve concurrent user requests correctly and fastly.

III. Testing

In the testing phase, I have tested the modules that I worked on individually. I have also been involved in regression testing of the modules I implemented and integration testing of the whole application. I tested the autoscaling with various numbers of images ranging from 1 to 200 images, where I was checking if the autoscaling is working properly. Here, the time wasn't considered as a parameter for testing.

Disha Bhukte (ASU ID:- 1219495225)

I. Design

Among the various modules in the project, I have proposed the design for the usage of SQS, hashmap for storing the output, scale down in autoscaling and the MVC model in Spring Boot in order to incorporate the UI in the application. The design of all the modules of the project were finalized only after a lot of discussions and brainstorming with all the group members.

II. Implementation

In terms of autoscaling, I have implemented a scaling down mechanism by terminating the instances once they stop receiving the messages from the input queue. In the web tier, I have implemented a thread which would continuously read the output queue and store the messages in a hashmap. This was done mainly to obtain the result of the image recognition with complexity O(1). It was done in order to reduce the processing time of any request. Also I have set a parameter to receive the maximum number of messages to 10 from the output queue to reduce the number of hits to Amazon SQS. This will bring down the costing. Also if we do not get any messages from the output queue I put the thread to sleep for 10 more sec so as to reduce costing. I have also implemented the MVC model in Spring Boot(i.e. WebAppAPIController) which was used for incorporating UI in the application. Apart from that I have implemented the utilities needed for usage

of the SQS such as configuring the two queues, creating input and output queue if they are not present, pushing and polling messages from the queues.

III. Testing

I have actively contributed along with the other members to the integration and regression testing of the project. On an individual basis, I have tested the modules that I have implemented along with multiple end to end testing of the project. My modules were designed in such a way that it handled multiple test cases. I checked my module with different image set sizes from 1 to 200.

Sapan Desai (ASU ID:- 1219080070)

I. Design

Among the various designs proposed for the usage of Amazon services in the project, I have designed the module concerning the usage of input and output S3. I have also contributed to scaling of app instances by using multithreading within the app tier to handle the load if the user requests exceed the maximum number of app instances i.e. 19. I have single handedly designed the web page used for uploading multiple images in a single submission. The design of the components in the project and their interaction was finalized after a lot of discussion among the team members.

II. Implementation

In the autoscaling efforts, I have designed as well as implemented the multithreading module. Given a scenario where the number of visible requests(95) in the input queue exceeds the maximum number of allowed app instances i.e. 19 significantly, each instance will spawn the threads on the basis of the demand. Given that one app instance started to run and keep pooling the request queue. It will then read the number of visible messages from the queue and based on that it will decide the number of threads. So thus creating 95/19 = 5 threads and starting serving the request. By the time the second app instance starts to run and assume that our first app instance already served 10 requests then the number of visible messages is only 85 then the number of threads will spawn will be 85/19 = 4 and so on for other app instances. The main reason behind doing is to distribute load equally on every app instance. I have also implemented the

web page using HTML, CSS and Javascript for making the application user friendly. The reason behind using javascript is to perform validation at client side. This will decrease the response time as we are not making any unnecessary hits on the API residing on the server. So this will make sure we only get images as input. Additionally, I have implemented the S3 related utilities which enable the application to configure and create input and output S3 buckets, insert input request and image recognition result in it as well as empty the S3 buckets and delete them. At present the application does not delete the S3 buckets.

III. Testing

After integrating the whole application, my main goal was to reduce the time to serve each user request. So I have tested the application with 1000 requests. I was able to improve the system performance by a lot. Initially application was able to serve 20 requests in 15 minutes and after implementing multithreading application was able to serve 1000 requests in under 4 minutes

Group Contribution:

We have followed Agile methodologies to complete the project. We have considered two sprints of 10 days each. We have conducted a stand up meeting everyday and discuss our problem and put together our thoughts for the approach and decide one final approach. We discuss our goals everyday, we decide what we will do today, what we faced yesterday, what is still in progress. After the first sprint our app took 15 minute to serve 20 requests. In retrospective, we mainly focused on how to improve the performance of the application. Every day each team member came up with a new approach, we discussed it and finalized one approach and implemented that. After the second iteration our app served 1000 requests in under 4 minutes which was a real improvement to the first version of our application. We have used git so that each team member can contribute to a certain task.