CSE 546 - Project 1 Report

Sai Charan Papani - 1222323895 - spapani@asu.edu
Lakshmi Venu Raghu Ram Chowdary Makkapati - 1222296439 - lmakkapa@asu.edu
Sarath Chandra Mahamkali - 1222281736 - smahamka@asu.edu
Jayanth Kumar Tumuluri - 1221727325 - jtumulur@asu.edu

Problem Statement:

The main aim of the project is to create an image recognition software that can automatically scale in and scale out based on demand. This elasticity is achieved by using the Amazon web service. The application utilizes Amazon EC2 for computing power, Amazon Simple Storage Service for data storage, and Amazon Simple Queue Service to store the requests. For Load Balancing, we have used the Amazon Simple Queue Service.

Design and Implementation:

There are mainly two parts to this whole project namely, web tier and app tier. The main reason behind breaking up the project is to use the advantages of decoupling and using the Amazon Simple Queue Service to connect the web tier with the EC2 instances where the actual computation is taking place. The Web Tier runs on one EC2 instance that handles all the requests. These requests are then sent to the Amazon Simple Queue Service. The requests are then distributed to EC2 instances to be handled, which includes the recognition of the face present in the given input image; these EC2 instances have a deep learning algorithm that is used to recognize the person in each image. We are storing the input images in the Amazon Simple Storage Service bucket with name “ccproj-group40-input-bucket-cse546” and outputs in the bucket “ccproj-group40-output-bucket-cse546”. Then this output is sent into an AWS Response Queue and Web tier consumes it.

Architecture:

The web tier continuously executes on a single Amazon EC2 instance. The load balancer is responsible for the creation and deletion of EC2 instances. The load balancer logic mainly uses the number of requests present in the Amazon Simple Queue Service.
To provide contention avoidance and fault tolerance visibility timeout is maintained by ensuring that the message request that is made after the threshold time, in the request queue (SQS) is visible to other App instances.

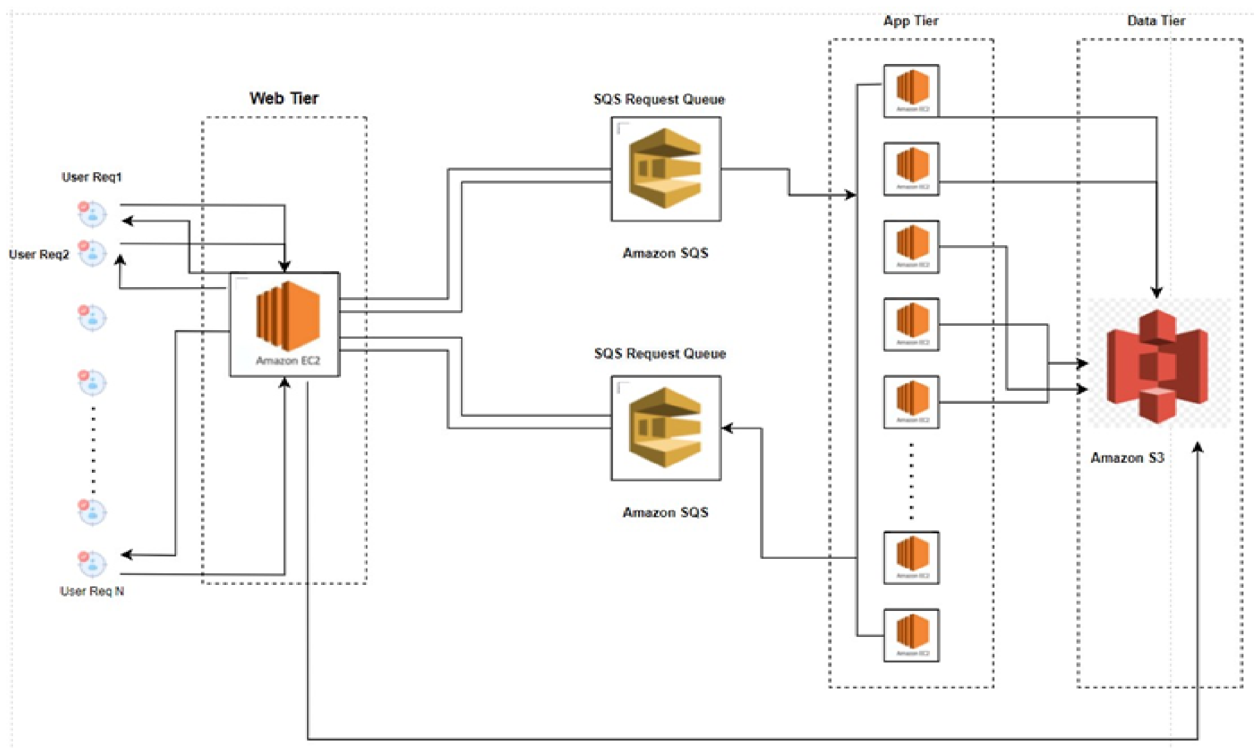


Figure 1 - Architecture

Autoscaling:

Scaling out means increasing the resources(App tier) according to the requests made. In this scenario, to achieve this, a band of 19 instances is maintained which ensures that at any given point of time the maximum number of instances that are created in the AWS account is 20 obeying the limits of the free tier usage of an AWS account. A load balancer is maintained to allow the creation of instances according to the instance band constraints. This process of using a maximum number of instances and a load balancer ensures that the user of the application avails of the provided service as quickly as possible.

At the app tier level Scaling takes place. The results produced after processing the input images using the deep learning algorithm are stored in the S3 bucket and the result is also sent to the web tier using SQS queue. The Wait Time feature ensures that the queue waits for about 10 seconds, listening to the request coming to the SQS request queue and if it does not, then the new request will terminate itself.

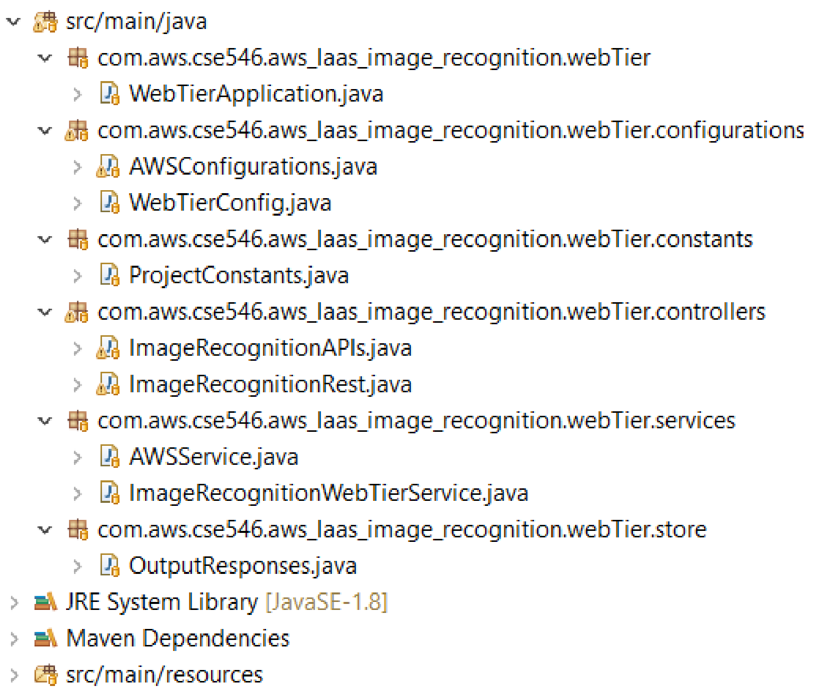
Testing and Evaluation:

The following procedure is strictly followed to test our application. Initially, the dataset of 100 images provided, was used to test the application. Checked whether the application is able to take in multiple images at once during a single upload time. Verifying the number of messages that are sent to the SQS queue is done by checking the status of the input queue. Then the number of uploaded input images in the S3 input bucket is verified. Finally, the number of running instances in the EC2 dashboard are observed keenly for testing the functionality of auto-scaling and load-balancing.

Results that were obtained after the above testing procedure:

The results generated by the classification are stored on S3 to maintain the persistent storage option. The results are sent to the SQS response queue that is used to display the results on the User Interface.

Code:

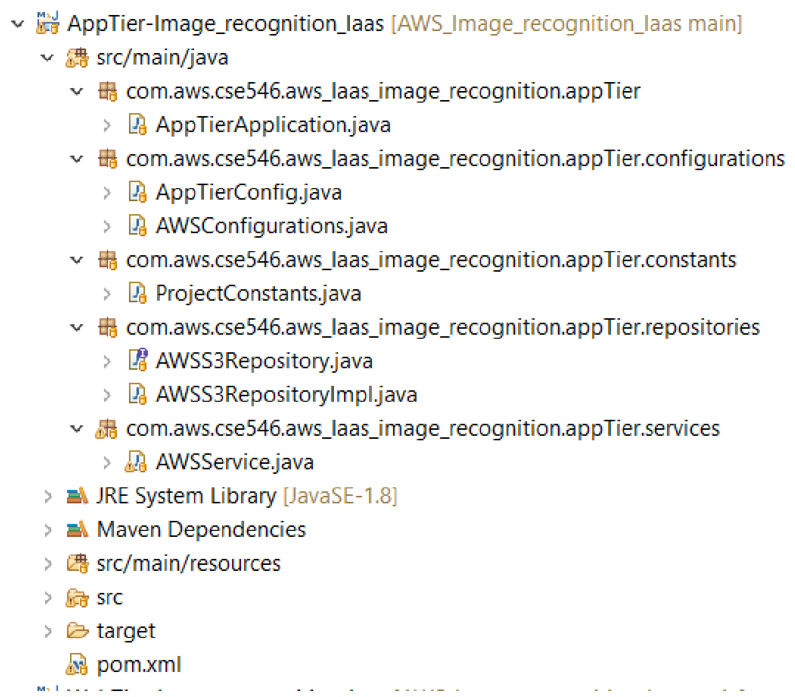


```
src/main/java
  com.aws.cse546.aws_Iaas_image_recognition.webTier
    WebTierApplication.java
  com.aws.cse546.aws_Iaas_image_recognition.webTier.configurations
    AWSConfigurations.java
    WebTierConfig.java
  com.aws.cse546.aws_Iaas_image_recognition.webTier.constants
    ProjectConstants.java
  com.aws.cse546.aws_Iaas_image_recognition.webTier.controllers
    ImageRecognitionAPIs.java
    ImageRecognitionRest.java
  com.aws.cse546.aws_Iaas_image_recognition.webTier.services
    AWSService.java
    ImageRecognitionWebTierService.java
  com.aws.cse546.aws_Iaas_image_recognition.webTier.store
    OutputResponses.java
JRE System Library [JavaSE-1.8]
Maven Dependencies
src/main/resources
```

Files

- ➔ WebTier
  - ◆ Configurations
    - These files contain the configuration beans that the spring boot application needs.
      - AWSConfigurations.java
      - WebTierConfig.java

- Constants
  - ProjectConstants.java
    - This file contains all the values of the credentials and other important parameter contents, required for the connections and successful execution of the application.
- Controllers
  - These files contain all the required API handler functions, to host and handle the requests to the website of the application.
    - ImageRecognitionAPIs.java
    - ImageRecognitionRest.java
- Service
  - AWSService.java
    - This file contains the methods that create a queue, accepts all the inputs from the user through the controller and sends these inputs as requests onto the queue. There are also other methods that are used to maintain the load balancing of the application and to delete the output messages from the output queue.
  - ImageRecognitionWebTierService.java
    - This file contains the methods to collect the output and display it by mapping the keys to their appropriate values.
- Store
  - OutputResponse.java
    - This class contains the output hashMap and the method to get the length of output.
- WebTierApplication.java
  - This is the main method or starting point of the web tier application.



AppTier-Image_recognition_Iaas [AWS_Image_recognition_Iaas main]
src/main/java
com.aws.cse546.aws_Iaas_image_recognition.appTier
AppTierApplication.java
com.aws.cse546.aws_Iaas_image_recognition.appTier.configurations
AppTierConfig.java
AWSConfigurations.java
com.aws.cse546.aws_Iaas_image_recognition.appTier.constants
ProjectConstants.java
com.aws.cse546.aws_Iaas_image_recognition.appTier.repositories
AWSS3Repository.java
AWSS3RepositoryImpl.java
com.aws.cse546.aws_Iaas_image_recognition.appTier.services
AWSService.java
JRE System Library [JavaSE-1.8]
Maven Dependencies
src/main/resources
src
target
pom.xml

➔ AppTier

- AppTierApplication.java
  - This is the main method or starting point of the App tier application
- Configurations
  - These files contain the configuration beans that the spring boot application needs.
    - AppTierConfig.java
    - AWSConfigurations.java
- Constants
  - ProjectConstants.java
    - This file contains all the values of the credentials and other important parameter contents, required for the connections and successful execution of the application.
- Repositories
  - These classes are used to upload the input images from the appTier instances to the S3 input bucket and also stores the classified images output in the S3 output bucket.
    - AWSS3Repository.java
    - AWSS3RepositoryImpl.java
- Services
  - AWSService.java

- This class contains the methods which implements the scaling in of the app instances, calling the S3 implementation method to store the values, method to create the queue request, method to delete the message in the request queue after the response queue starts processing and terminating the instances.

Steps to execute the code:

➔ Update the project constants in the ProjectConstants.java file as per your AWS set up in both modules (web tier and app-tier).

➔ Using Maven clean and install, create a jar file for the AppTier.

➔ Customize the given AMI by creating an EC2 instance and upload the AppTier .jar file using WinSCP, then install the java environment in that instance and finally create an image (say X) of this instance. This image X will be used to create the app instances.

➔ Create an instance that runs continuously until the application is closed.

➔ Transfer the WebTier .jar file using WinSCP to the created EC2 Instance.

➔ Setup the java environment in the java using appropriate install commands in the same instance.

➔ Execute the web .jar file in the instance.

➔ The HTTP requests are sent to the below URL
http://ec2-52-87-139-44.compute-1.amazonaws.com:8080/getfacerecognizationperImage

➔ To test the application run the following command:

```
Python3 multithread_workload_generator_verify_results_updated.py --num_request 100 --url http://ec2-52-87-139-44.compute-1.amazonaws.com:8080/getfacerecognizationperImage --image_folder face_images_100/
```

Individual Contributions:

Saicharan Papani (ASU ID: 1222323895):

- ➔ Design
  - ◆ The whole design and architecture were progressively made based on all the team members' ideas. I have majorly contributed to the Web Tier architecture, implementation of the code, scale out functionalities and integrations, etc. I also pitched in a few ideas in the app tier to make the application work the way it is working now.
- ➔ Implementation
  - ◆ The complete application was developed using Spring Boot, MVC, Thymeleaf technologies. We already know that the Spring boot is quite popular for developing microservices with features like loose coupling, IoC, dependency injection and concurrent request handling, etc. I have implemented the auto scaling code in a multithreaded fashion, the reason for using multi-threading is for reducing the load on the main thread (It avoids interference). Instead of creating stereotyped singleton service objects in the ApplicationContext or BeanFactory, I have implemented the services like AWSService (deals with SQS, creating of EC2 instances with Scale-out functionality, etc.), ImageRecognitionWebTierService (deals with the listening to response SQS, etc.) which are run as threads. This implementation will boost the performance of the application and the embed tomcat service can smoothly handle the concurrent requests without any interference. The controller will use the services that I have created for putting the messages in the request queue and for retrieving the face recognition results out of output HashMap. AWSService continuously monitors and loads the results from the response queue to output Map using SQS services. Based on the number of images in the input queue (put by the controller), AWSService scales out the App tier instances by creating the required number of EC2 instances from an AMI which has java pre-installed, App tier jar loaded and face recognition related code in it (This pre-configuration will reduce the efforts that we need to put in for explicit configurations). Regarding Images, I proposed and implemented the logic of converting the images into base64 encoding and sending it to SQS by using the SQS services. Encoding and decoding using base64 came into the picture as we require to transfer binary data to the App tier for processing. Decoding is done in the App Tier for getting the image. Apart from these, I have helped in a few other implementations like how to run a python script and extract results in the App tier, etc.
- ➔ Testing

- I have thoroughly tested the functionalities at various phases of application development by performing unit, regression, and integration testing. The scale-out logic was tested multiple times with a different combination of the number of images as that is the main functionality of my contribution. At last, we as a team tested the whole application which performed well and was able to serve the concurrent 100 requests in a short duration of under 4-5 minutes.

Sarath Chandra Mahamkali (1222281736):

➔ Design :

- As mentioned in the description of the project flow, I have implemented the handler functions that are required to handle the requests made to the hosted web application and process the requests accordingly, to return outputs to the Web UI. The developed handler functions act as the backbone of the webtier part of the application by defining the structure of the application. The design of the functions and their modularity were all finalized after a series of discussions among the group members ,converging to a common solution.

➔ Implementation :

- The files that are generated to store all these handler functions are together named as Controllers. These controllers have some basic responsibilities in order to maintain the connectivity between the backend that is where the processing and lookup of the image inputs is done and the frontend part of the application that is the user interface developed for the hosted application. Firstly, the controllers monitor the request queue to maintain the files required. If a request queue is not present then the controllers will create one, else the ImageRecognitionWebTireService will be used to create a unique file. The controllers collect the created multipart files and then convert them into JavaFileObjects. The content present in these fileobjects are converted into base64 content. This base 64 content contains the information in the following format: imagename + –filename– + the actual input test file name. Subsequently, all this parsed base 64 data is dropped into the request queue using the ImageRecognitionWebTireService. One of the main and important functionality of the controllers is to continuously monitor the hashmap named as output response for the termination and return of the result key. The above mentioned hashmap contains a key as the output file name. These key values are continuously looked up with a time complexity of O(1) and in the runtime, if the desired key is found the, that output filename is returned as an output that is sent into the WebUI for generating the final value present in the output file.

➔ Testing

- I have tested the controller functionalities thoroughly in various scenarios with various combinations of workload generators with different numbers of requests. Later on, I have performed unit testing, regression and integration testing. At last, As a whole we have tested the application and it was able to handle the required load successfully.

Jayanth Kumar Tumuluri (1221727325):

- ➔ Design
  - ◆ As per the architecture of the project, I have handled the scaling-in function of the AppTier. This script runs in each app instance. Used Multithreading to handle the load and accordingly scale the app instances which improves the performance.

- ➔ Implementation
  - ◆ I have implemented the application using the same technologies as the web tier. I have predominantly made contributions to the scale-in functionality. In the scaling-in implementation, I have implemented a function in which if the number of requests are more than the maximum number of app instances, each instance will invoke a thread depending on the demand. Also an instance is created for continuously polling the request queue to know the demand. Once the SQS queue starts receiving requests from the WebTier, the AppTier instance which monitors the queue will read the number of requests, then trigger the threads and finally delete the requests which it reads from the queue. So for example, let the number of requests be 100, the number of threads which process would be (total number of requests) / (total number of max instances) => 100/19 = 5. If each thread processes 2 requests in parallel, then the number of requests served would be 10. So the instance which monitors the SQS request queue will now see that there are 90 messages in the request queue and the same process continues. Once these threads handle the requests and process the output, it will trigger the instances to terminate. Additionally, I have implemented the function which sends the output of each app instance to the response queue after the requests are served.

- ➔ Test
  - ◆ After the final integration of the project, I have tested the application by varying the number of requests. I have tested for 2, 10, 50, 100 and 1000 requests respectively. Because of the multithreading implementation which is very efficient, the execution time was within the expected time which was around 4 - 5 minutes for 100 requests.

Lakshmi Venu Raghu Ram Makkapati (1222296439):

- ➔ Design
    - ◆ There are several different modules present in this application. Among them, I have suggested the use of HashMap for the storage of the output and also worked on the module which contained the usage of Amazon Simple Storage Service. Additionally, I have also worked on the creation of a custom AMI which contained additional dependencies. The final plan of all the involved modules was finished solely after a large number of conversations and conceptualizing with all the Teammates.
- ➔ Implementation
    - ◆ As explained earlier there are two main parts to the application namely the app tier and the web tier. For the web tier, I have helped in the development of a method that would peruse the response queue for an indefinite amount of time. Later the result of this method is stored in a hashmap. This has diminished the handling time of the request considerably. I have also put in an additional parameter in the app tier so that in the event of the application not receiving any messages from the response SQS queue, the thread would sleep for an additional ten seconds so that the amount of cost billed is reduced. Also, I have carried out the implementation of the S3 related utilities. These included connecting to the Amazon Simple Storage Service, and also the creation of the input and the output buckets. I have contributed to the methods which are utilized in placing the input images onto the input bucket once we get a request in the request queue and also placing the output values onto the output bucket once the threads present in each app tier instance process the input and send the output.
- ➔ Testing
    - ◆ I have participated and contributed to the joining of different modules in the project and for the Test phase. Individually, I have checked with various test scenarios, all the different methods created by me and made sure that all of them worked as per the requirements. On top of that, I also took part in the final testing of the module and also after hosting the application on AWS.