CSE 546 — Project 1 Report

By

Mohamad Mudassar Shekh 1211247229

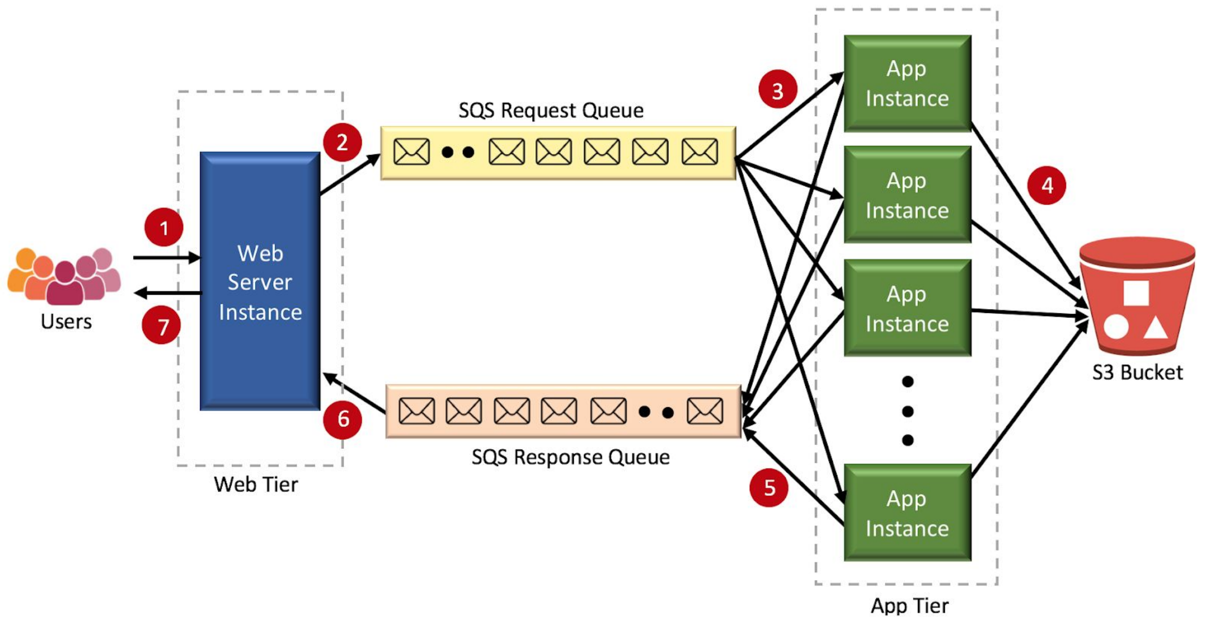
Mrunal Bodhe 1213134843

Nishi Shah 1211180916

Anandavaishnavi Ardhanari Shanmugam 1211256043

1. Architecture

The architecture followed in our application involves five main components - Web Tier, SQS Request Queue, App Tier, S3 Bucket and SQS Response Queue. The design of the architecture and the basic implementation of the five components are illustrated in the figure below.



1 Users send the image URLs to the application which are caught by the web server instance

2 Web server instance sends the message to the SQS request queue and performs load balancing

3 The message from the SQS request queue is picked up by one of the running app instances

4 The app instance puts the image name and output from the image recognition module into S3 bucket as key-value pair

5 The app instance deletes the message from SQS request queue and sends the image name and output to the SQS response queue

6 The web server instance picks up the responses and deletes it once sent to the user

7 The web server sends the response back to the user

Figure 1: Architecture diagram of the entire application with steps

The design and implementation of each component is described as follows:

- **Web Tier:** The web tier is implemented using one EC2 instance (Web Server Instance) which executes the web server program. It gets the URLs requested by the users as concurrent requests via the REST API and sends the URLs as messages to the SQS request queue.

  It houses a load balancer which constantly compares the approximate number of messages in the SQS request queue and number of running EC2 instances (App instances) in the App tier to help in scaling up the application by increasing the number of app instances as the number of messages in the SQS request queue increases or increase the app instances to a maximum count of 19 if there are 19 or more requests in the SQS request queue.

  The web server instance listens to the SQS response queue for responses and sends the respective output to the user based on the URL requested initially. If the response is not received within four minutes, the web server issues a time out message.

  The web server instance has been assigned an IAM role which grants full access to EC2 and SQS services in order to perform the above mentioned operations.

- **SQS Request Queue:** This queue houses all the user requests sent by the web server instance. Each message has a visibility timeout limit as 30 seconds. This ensures that once a message is picked up by the app instance, it no longer will be visible to the other app instances for 30 seconds and after which will be made visible to all running app instances unless deleted by the app instance that picked it up.

- **App Tier:** The app tier houses the app instances which facilitate the image recognition process. Each running app instance constantly listens to the SQS request queue for messages and performs one of the following:

  - If the instance receives a message, it picks it up and runs the deep learning script. It then fetches the output and puts the image name from the URL and output as key-value pair in the S3 bucket ("mykeyvaluebucket"). After which, it deletes the message from the SQS request queue and sends the key-value pair to the SQS response queue.

  - If the instance does not receive any message, it self terminates. As such, the application scales down when the idle instances are terminated.

  The app instances have been assigned an IAM role which grants full access to the SQS, S3 and EC2 services to perform the above mentioned operations.

- **S3 bucket:** The S3 bucket named "mykeyvaluebucket" holds all the image names and the corresponding output as key-value pairs. Any object that comes in with an already existing key will have its content overwritten so the bucket holds unique keys at all times.

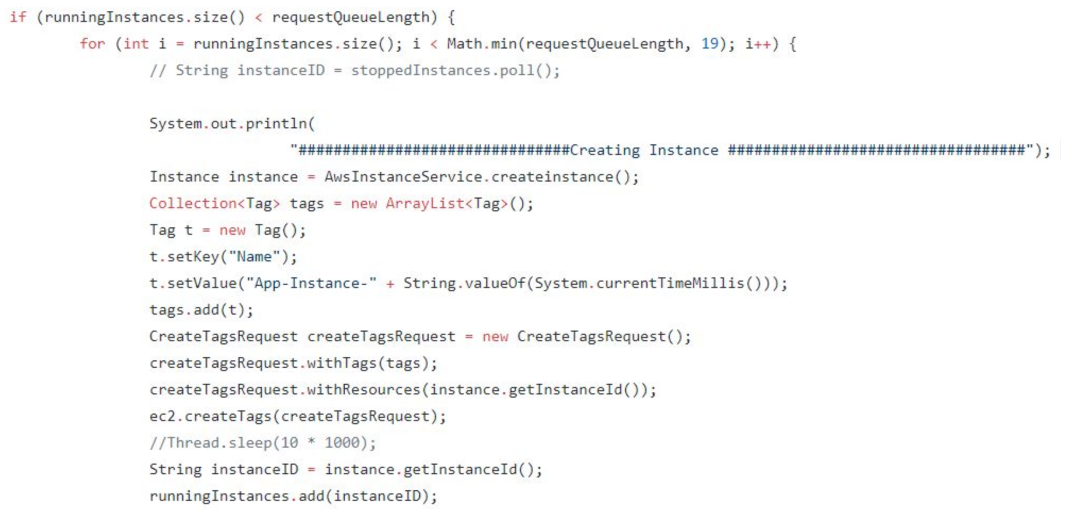
- **SQS Response Queue:** This queue houses all the image names and their corresponding output as responses sent by the app instances. Each message has a visibility timeout of zero seconds since the web server instance is the only one listening to the queue for messages and any message that comes in must be promptly send to the user.

2. Autoscaling

Autoscaling thread monitors our application constantly. We are starting the LoadBalancer thread when the application gets started on web tier.

- **Design**: The Load Balancer thread checks the status of running and pending instances at each interval of 3 seconds. At the same time, we are also checking the total number of messages in the Request SQS. If this number exceeds the total number of running instances, we are creating more instances on demand. As limit for maximum running instances is 20 including web-tier instance, In the case when there are more than 20 messages in the Request SQS, we will create 19 instances at max . This is our design for Scale-up. The app-tier instances continuously receive the messages from the Request SQS and process them for image Recognition. For Scale- down, the instances will terminate themselves at the moment when they will not find any message in the queue. Because when there are no messages in the queue, it will be waste of resources to keep the app instance running.

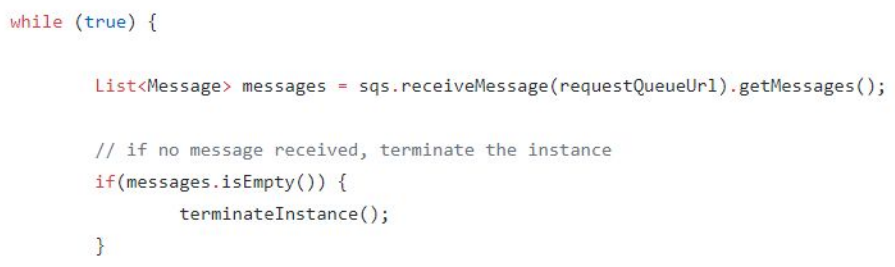
- **Implementation**: For unique names, we are attaching timestamp with app-instance tag name while creating new instance.
  Scale Up Implementation on Web Tier:



```
if (runningInstances.size() < requestQueueLength) {
        for (int i = runningInstances.size(); i < Math.min(requestQueueLength, 19); i++) {
                // String instanceID = stoppedInstances.poll();

                System.out.println(
                                "###############################Creating Instance ##################################");
                Instance instance = AwsInstanceService.createinstance();
                Collection<Tag> tags = new ArrayList<Tag>();
                Tag t = new Tag();
                t.setKey("Name");
                t.setValue("App-Instance-" + String.valueOf(System.currentTimeMillis()));
                tags.add(t);
                CreateTagsRequest createTagsRequest = new CreateTagsRequest();
                createTagsRequest.withTags(tags);
                createTagsRequest.withResources(instance.getInstanceId());
                ec2.createTags(createTagsRequest);
                //Thread.sleep(10 * 1000);
                String instanceID = instance.getInstanceId();
                runningInstances.add(instanceID);
```

  Scale Down Implementation on Receiver:



```
while (true) {

        List<Message> messages = sqs.receiveMessage(requestQueueUrl).getMessages();

        // if no message received, terminate the instance
        if(messages.isEmpty()) {
                terminateInstance();
        }
```

3. Code

The overall functionality required from the project are developed and split into two phases. The first phase consists of all the functionalities required from the web tier which will include the code for load balancer and the restful API. The second phase will consist of all the functionality required from the individual app instance which will be responsible for consuming the messages, executing the deep learning scripts, populating the output in S3 and SQS, and terminating the instance.

Web Tier: Web Server instance will consist the below code/functionalities and its main function lies in the RestfulWebServiceApplication Java file :

- Restful API: For creating the Restful API we install all the required Spring Boot dependencies and declare the required functionalities in the file ApiController.java. In the API controller we first declare the the get function using @GetMapping which routes all the incoming get Http requests with the mapping "/cloudimagerecognition.php" , to our given function getUrl. In the getUrl function we will first fetch the url from the Http request and put in into our sender queue using AWS Sdks SendMessageRequest function. We then subscribe to the receiver queue " cc_proj_receiver2" and check if the received message contains the url provided by the user in the Http request. If the url matches we return the rest of the data in the message to the user. This implementation of sending and receiving messages is implemented in the file *SqsServices*. If the request doesn't receive a response in the message queue for a given amount of time, we will return the string "Timed Out".

- Load Balancer: This part of the code will handle creation of new EC2 instances based on the number of messages in the queue and the number of running instance. The code is present in the java file *LoadBalancer*.java. This class object is run as a new thread when web tier is started in *RestfulWebServiceApplication*. Load Balancer first fetches all the instance using the function ec2.describeInstances(). We then extract the number of running instance by checking its state using the code instance.getState() .getName() and if the number of running instances is less than 20 and less than number of messages in the queue then we create the required amount of instances using the function runInstances in *runInstanceRequest* giving it a Tag name and assigning userdata. Creation and Management of instances are implemented in the java file *AwsInstanceService*. The Load balancers thread executes every 3 seconds to check for number of running instance and manage the creation of new instance.

App Tier: App tier instance will consist the below code/functionalities and its main function lies in the Receiver Java file

- Receiver: Each app Tier instance will have a Receiver Jar present inside and will start executing as the run command is given in the instances user data. Receiver will subscribe to the *cc_proj_sender1* queue and check if any message can be received. Once a receiver receives the message it will then use the url in the message to execute the deep learning using process and exec the bash commands. Once the deep learning image recognition code is executed the output is fetched using buffered reader and put into an S3 Bucket called "*mykeyvaluebucket*" using the SDK function *putObject*. We then delete the message from the queue using the function *deleteMessage*. The request Url and the deep learning Image recognition output is concatenated and pushed into the *cc_proj_sender1* queue so that it can be returned to the requester. If the receiver checks for messages and doesn't find any messages then it will call the function *terminateInstance*. This function will first fetch the current instances ID and terminate itself using *TerminateInstancesRequest*.

Steps for execution:

1. Start the web server instance named “Web_app” in the AWS GUI.
2. The code for web server instance is now running you can use the public IP generated to run the test scripts.

4. Project status

No.	Specification provided	Status on meeting the specification
1.	The app should take images uploaded via HTTP and perform image recognition on these images using the provided deep learning model. It should also return the recognition result (the top 1 result from the provided model) to the users via HTTP.	**Requirement Met** - Our application takes image URLs as http requests http://<publicIP_of_webserverinstance>/cloudimagerecognitio n.php?input=[URL of the image] and runs the python script on them, retrieving the top 1 result from the deep learning model and returns the same to the user over HTTP.
2.	The deep learning model will provided to you in an AWS image (ID:ami-07303b67, Name: imageRecognition, Region: us-west-1a). Your application will invoke this model to perform image recognition on the received images.	**Requirement Met** - Our app instances were created using the provided AWS image.
3.	The application should be able to handle multiple requests concurrently.	**Requirement Met** - Our application handles concurrency in handling multiple requests with the help of the REST API.
4.	It should automatically scale out when the request demand increases, and automatically scale in when the demand drops.	**Requirement Met** - Our application handles this using the load balancer logic implemented in the web server instance.
5.	The application should use no more than 20 instances, and it should queue all the pending requests when it reaches this	**Requirement Met** - Our application only uses 1 web server instance and 19 app instances (at max) at any time. The requests are queued up in

	limit.	the SQS request queue when the number of incoming requests exceed the count of 19.
6.	When there is no request, the application should use the minimum number of instances.	**Requirement Met** - When there are no requests in the SQS request queue, all the app instances terminate and the web server will be the only running instance.
7.	Give your instances meaningful names, for example, a web-tier instance should be named in the fashion of“web-instance1” and an app-tier instance should be named “app-instance1”.	**Requirement Met** - Web server instance is named as “Web-app” and all the app instances are named “App-Instance-<current_timestamp>” where the current timestamp is in milliseconds.
8.	All the inputs (imagenames) and outputs (recognition results) should be stored in S3 for persistency. They should be stored in a bucket in the form of (input, output) pairs.	**Requirement Met** - Our application puts the image name and output as key-value pair in the S3 bucket in the format (image name, output) where image name is the key and output is the value.
9.	The application should handle all the requests as fast as possible, and it should not miss any request. The recognition requests should all be correct.	**Requirement Met** - Our application handled all requests once they hit the web server instance and no requests were missed and the correct response was sent to the user.
10.	For the sake of easy testing, use the resources from only the us-west-1 region for all your app's needs.	**Requirement Met** - All the AWS services used by the application are from the region us-west-1.

5. Individual contributions (by Anandavaishnavi Ardhanari Shanmugam 1211256043)

My contribution to the project was mainly on the logic and working of the app instances in the app tier. The details on the design, implementation and testing of this module are elaborated as follows.

Design: The app instances which are created by the web server instance constantly listen to the SQS request queue for messages. When a message is available, it picks it up and runs the deep learning script to obtain the recognition output for the respective image URL. It then takes the image name from the URL and stores it along with the image recognition output as key-value pairs in the S3 bucket. Upon placing the object in the S3 bucket, the app instance then deletes the respective URL processed from the SQS request queue and sends the image name and corresponding output to the SQS response queue to be picked up by the web server instance. If there were to be a failure in any of the above mentioned steps during execution, the instance terminates and makes the message visible to the remaining running app instances so as to be processed. Also, when there are no messages in the SQS request queue, the app instance then fetches its instance ID and then self terminates as part of the scaling down process.

Implementation: The logic for the app instance is present in the receiver.java file. This file was packaged into a jar and was set to run during the initial boot of the instance by including the command to run the jar in the user data of the instances created. There are two methods in the receiver class - main and terminateInstance. The main method contains the logic to process the URL and calls the terminateInstance method upon any failure or when the list of messages fetched from the SQS request queue is empty. The pseudo code of these two methods are as follows:

<u>In main():</u>

```
while (true) {
    Receive message from SQS request queue
    if(no messages received) { terminateInstance(); }
    for each message received {
        Obtain output from the deep learning module, call terminateInstance() on failure
        Put the image name and output into S3 bucket, call terminateInstance() on failure
        Delete message from SQS request queue, call terminateInstance() on failure
        Send image name and output to SQS response queue, call terminateInstance() on failure
}
```

<u>In terminateInstance():</u>

```
Get the instance ID of self
Execute termination request on the instance ID
```

Testing: In the testing phase, the jar was placed in a new manually created app instance using the scp command and then the jar was manually run using the "java -jar <jarfilename>" to check for issues. The app instance was also manually assigned an IAM role which grants full access to the SQS and S3 services since the app failed otherwise due to permission issue while manipulating these AWS services via code. A few messages were manually placed in the SQS request queue and it was constantly checked for consumption of messages and the response queue for incoming number of responses. Also the S3 bucket was checked for the presence of objects and if they came in the correct key-value pair format having the correct output values.