

# **CSE 546 - Cloud Computing**

## **Project 1 - Image Recognition as a Service**

### **Detailed Report**

#### **Group Members:**

- (1) Mit Patel (1217114054 - [mpatel42@asu.edu](mailto:mpatel42@asu.edu))
- (2) Darshil Shah (1218490169 - [dshah27@asu.edu](mailto:dshah27@asu.edu))
- (3) Saumya Gangwar (1219377939 - [sgangwa4@asu.edu](mailto:sgangwa4@asu.edu))

## 1. Problem Statement:

The aim of this project is to build an elastic-application using IaaS Cloud (IaaS services of AWS) that can automatically scale in and scale out on demand and cost-effectively. The cloud application will provide image recognition services to users by performing deep learning on images provided by users. The deep learning model was provided in an AWS image (ID: ami-0ee8cf7b8a34448a6, Name: app-tier, Region: us-east-1). This application invokes a deep learning model to handle concurrent requests. In order to achieve this we have used Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Simple Queue Service (SQS) AWS services. Using cloud resources we can build scalable applications to handle demand surges as well as improve the overall performance and overall cost-effectiveness of the system.

## 2. Design and Implementation:

### 2.1 Architecture:

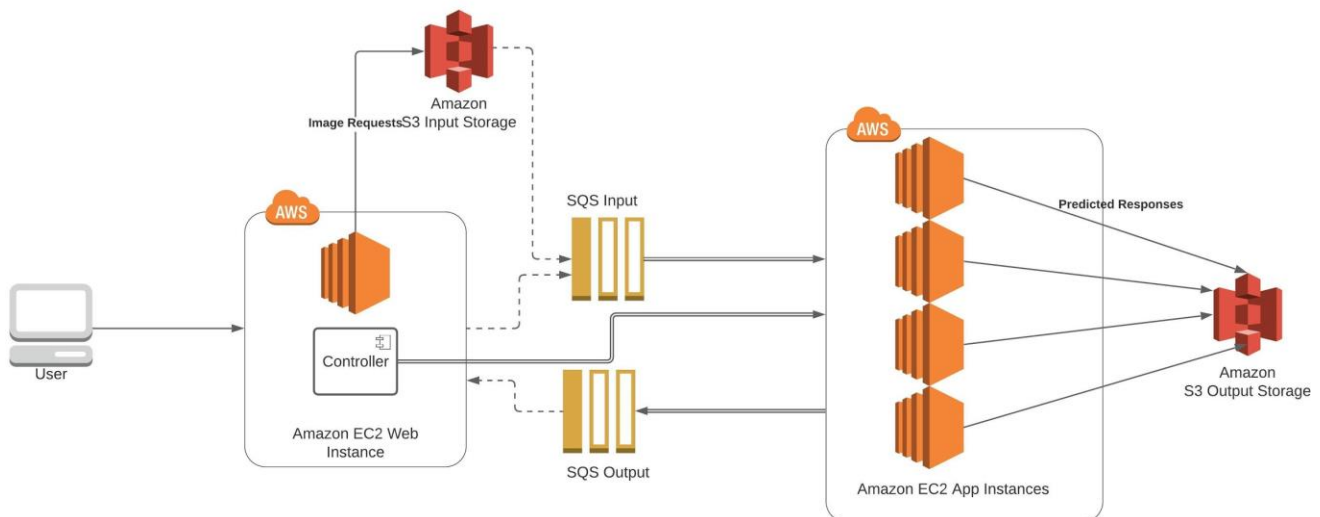


Figure:1 Architecture of image-classification elastic system

### **2.1.1 AWS Services used in project:**

#### **1. AWS EC2**

EC2 provides scalable computing .We can launch EC2 instances (virtual machines based on demand ,configure security ,networking and manage storage.In our implementation we are using EC2(Ubuntu images with deep learning environment) for app-tier and EC2(Ubuntu image with node application) for web-tier.

#### **2. Amazon S3**

We have used S3 in our project for persistent storage. We have configured two buckets respectively for storing uploaded images and predicted output received from the deep learning model. In S3, data is stored in buckets in the form of objects(key, value) pairs.

#### **3. Amazon SQS**

SQS decouples the microservices of cloud applications. We are using SQS (Standard) for incoming and outgoing messages. SQS helps in making our application scalable as the controller spawns new instances based on the number of running instances and the queue length(number of messages in input queue) at the moment.

### **2.1.2 Framework used:**

We have used the express node js framework for our REST application on web-tier and java maven tool for controller and app-tier. We also have configured cron jobs to bring in some automation.

### **2.1.3 Architecture Explanation:**

Our architecture consists of the app-tier and the web-tier. The web-tier takes in requests from the user. The requests received from the web-tier are pipelined into input queue SQS and fed into app-tier EC2 instances (19 maximum , 1 minimum). These instances will be scaled using the controller which is deployed web-tier instance.The controller will decide how many EC2 instances are required at a given moment.We can have a maximum of 19 app-tier EC2 instances for scaling out.The app-tier instance will take the image name from SQS and download the image with the same key from S3 bucket and process the image using the deep learning model provided and output the predicted label. It will also store the subsequent results in S3 for persistency as well as output queue SQS.

Initially we have one web-tier instance and 1 app-tier instance running to handle quick responses. Scaling out is handled by the controller and scaling in is handled by the app-tier instances themselves as they auto-terminate if they do not find any message requests on sqs. Long polling is implemented on the app-tier EC2 instance that is persistently running. The controller spawns the EC2 instances based on number of messages in SQS (length of SQS) and current number of running EC2 instances. The controller can spawn a maximum of 18 instances to handle requests while scaling out. For scaling in the individual instances auto-terminate and short polling is implemented on individual instances hence if there are no visible messages in SQS input queue the EC2 instances auto-terminate.

Once the output has been obtained from the deep learning model on the EC2 instance, it feeds the predicted value in output SQS which will be retrieved by the web tier instance and displayed on the frontend whenever the user requests for the results of the uploaded images so far. This simplifies the architecture and improves the overall performance of the system.

## **2.2. Autoscaling:**

Scaling-in and scaling out are done at app-tier. We ensure scaling in our application using the following components.

**1. Amazon SQS-** SQS is the heart of our auto scaling algorithm. The controller uses the queue length to determine how many EC2 instances are required and current load and performs the necessary amount of scaling out. Scaling-in is done automatically by the app-tier instances if they cannot find any messages in the SQS input queue then EC2 app-tier instances auto-terminate.

**2. EC2 Controller-** This is really essential for scaling out in our project. The main task of the controller is to analyze the load based on the current running EC2 instances and length of input queue SQS and spawn EC2 instances based on this and scale out.

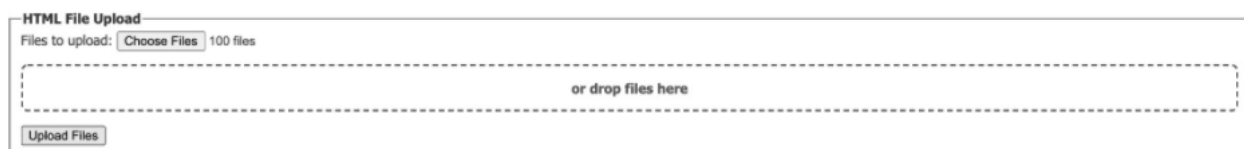
**3. Compute Module-** On initializing every EC2 instance executes the compute module which is the listener. Here our custom AMI which we are using to create the app instances has the capability of running the java listener application using cron job every time it boots up. Then the app instance will start doing the work that it needs to do. Here short polling is implemented and scaling in is done automatically. After an app-tier predicts the output of a given request it feeds the output to the output queue SQS and puts the key-value object pair in S3 bucket it

terminates. We can also use one of the SQS services that is wait time. If the wait time is 10 seconds, then the app instance will wait for 10 seconds looking for messages in SQS in single request. But here we have implemented short polling keeping the wait time as 0. So in essence application instances will run in a loop till they are able to find any request in SQS but if they do not find any message they automatically terminate hence scaling in the entire application. But one of the app-tier instance will continuously keep running we have implemented long polling in it keeping the wait time as 20 seconds to continuously serve requests for quick responses.

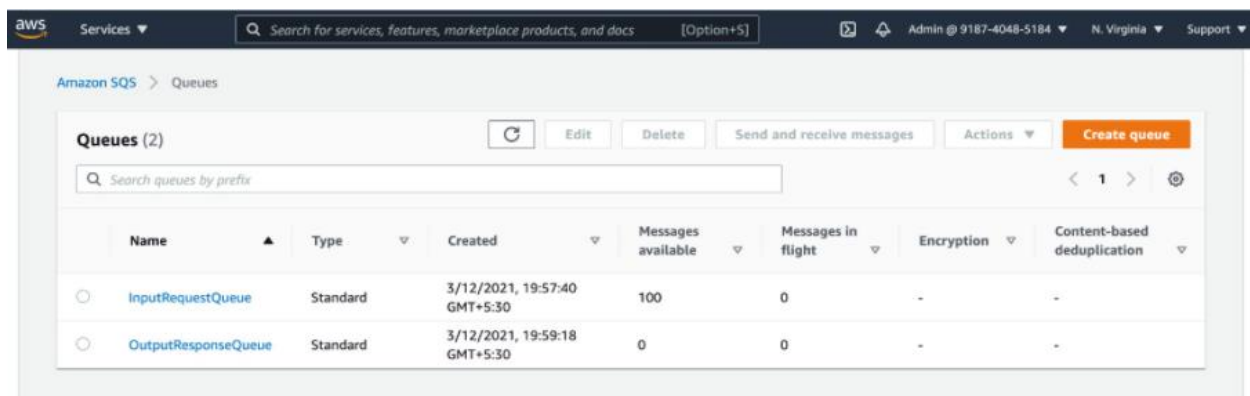
### 3. Testing and Evaluation:

We performed testing on a set of 100 images.

**Step 1:** We uploaded multiple images together. (100 in total) through the upload module page.



**Step 2:** After uploading the images they were stored in input bucket S3 as well as the input queue SQS



Name	Type	Created	Messages available	Messages in flight	Encryption	Content-based deduplication
InputRequestQueue	Standard	3/12/2021, 19:57:40 GMT+5:30	100	0	-	-
OutputResponseQueue	Standard	3/12/2021, 19:59:18 GMT+5:30	0	0	-	-

**Step 3:** After this the controller spawns the instances based on the number of SQS requests in input queue SQS. Here we observed that the controller spawns instances from 1 to 18 and creates these instances to read the 100 messages in input queue SQS. So these 18 instances along with the persistent app-instance that is continuously running handle these 100 requests.

The screenshot shows the AWS Management Console for the EC2 service. The left sidebar contains navigation links for EC2 Dashboard, Events, Tags, Limits, Instances, Instance Types, Launch Templates, Spot Requests, Savings Plans, Reserved Instances, Dedicated Hosts, Scheduled Instances, Capacity Reservations, and Images. The main content area is titled 'Instances (20)' and shows a list of instances. The instances are filtered by 'Instance state: running'. The table lists the following instances:

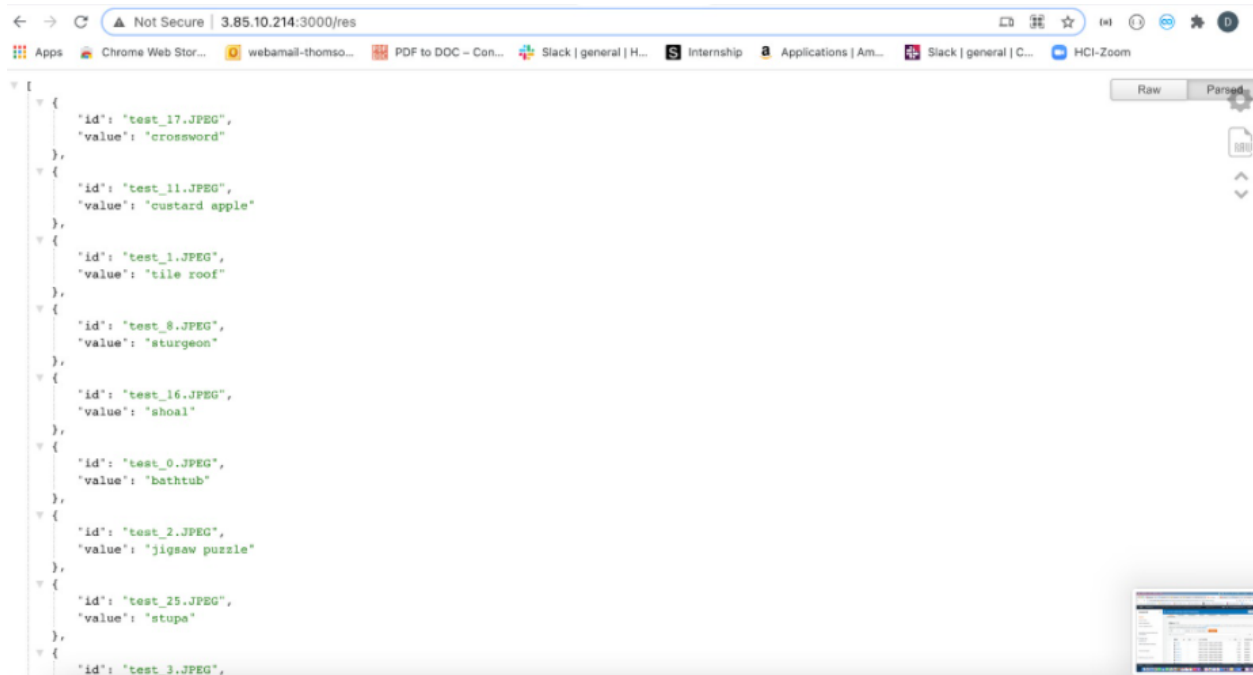
Name	Instance ID	Instance state	Instance type	Status check	Alarm status	Availability zone
app-instance3	i-003d2f1d67a546f76	Running	t2.micro	Initializing	No alarms	us-east-1
app-instance2	i-055066279b3e5e52b	Running	t2.micro	Initializing	No alarms	us-east-1
app-instance12	i-0bdf3700d42dc3b53	Running	t2.micro	Initializing	No alarms	us-east-1
app-instance18	i-03d8914052b1b3f92	Running	t2.micro	Initializing	No alarms	us-east-1
app-instance10	i-0e3277ac1bbc294a0	Running	t2.micro	Initializing	No alarms	us-east-1
app-instance9	i-067f87e8c5656272e	Running	t2.micro	Initializing	No alarms	us-east-1
app-instance-running	i-02a2ca2889f0f8683	Running	t2.micro	2/2 checks passed	No alarms	us-east-1
web-instance1	i-081abe15350d585ca	Running	t2.micro	2/2 checks passed	No alarms	us-east-1
app-instance6	i-0ddf1ae813e8c336c	Running	t2.micro	Initializing	No alarms	us-east-1

**Step 4:** The created instances process the request and store the predicted value in S3 output bucket. So here when we tested we observed that all 100 images were stored with their predicted values in S3 output bucket and Output SQS queue.

The screenshot shows the AWS Management Console for the S3 service. The left sidebar contains navigation links for Buckets, Access Points, Batch Operations, Access analyzer for S3, Block Public Access settings for this account, Storage Lens, Dashboards, AWS Organizations settings, Feature spotlight, and AWS Marketplace for S3. The main content area is titled 'Objects (100)' and shows a list of objects. The objects are named test\_40.JPG through test\_50.JPG. The table lists the following objects:

Name	Type	Last modified	Size	Storage class
test_40.JPG	JPEG	March 12, 2021, 23:02:52 (UTC+05:30)	1.9 KB	Standard
test_41.JPG	JPEG	March 12, 2021, 23:02:32 (UTC+05:30)	2.1 KB	Standard
test_42.JPG	JPEG	March 12, 2021, 23:02:32 (UTC+05:30)	1.9 KB	Standard
test_43.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	1.7 KB	Standard
test_44.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	2.4 KB	Standard
test_45.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	1.7 KB	Standard
test_46.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	2.0 KB	Standard
test_47.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	1.8 KB	Standard
test_48.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	1.6 KB	Standard
test_49.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	2.1 KB	Standard
test_5.JPG	JPEG	March 12, 2021, 23:02:32 (UTC+05:30)	1.8 KB	Standard
test_50.JPG	JPEG	March 12, 2021, 23:02:33 (UTC+05:30)	1.8 KB	Standard

**Step 5:** The predicted result is fetched from the output queue SQS and all 100 requests with their results are displayed on the frontend.



## 4. Code:

### 4.1 Code Modules:

#### 1. Web-tier Module:

- Launch app-module ( index.html ) : Provides users the usability to upload multiple images and redirects to upload.html page on successful upload.
- Check result and upload more images (upload.html) : The page displays two options to the users to upload more images or check the result of already uploaded images.
- Creating express node server (Index.js) : The file creates an express node server on listening port 3000. The server uses multer and multer-s3 apis to upload images to the S3 input bucket. We have configured event notification on the put action of S3 input bucket, so as soon as any new object is put into the bucket it triggers this event and sends notification to input request SQS.

- d. Return the result from output SQS : AWS sqs-consumer api fetches all the response messages from output SQS, pushes them into the result list and deletes them after processing.
- e. The Controller module runs on one same web-tier AWS EC2 instance to scale out based on the requests. The module uses AmazonSQS client to get the approximate number of messages in the input request queue.
- f. It also used AmazonEC2 client to get the total number of running EC2 instances.
- g. On the basis of the above two parameters it calculates the required number of EC2 instances needed to serve requests and spawns them. This fundamental ensures proper scaling-out of app instances. However, scaling-in is taken care of by individual EC2 instances it spawns.

## **2. SQS-Listener Module:**

- a. This module is responsible for processing the Input SQS messages using the deep learning algorithm provided by AWS AMI ID.
- b. Every EC2 instance which Controller spawns is going to run this module to process the message.
- c. It fetches the message (Image Name) from Input SQS Queue and then downloads that image from Input S3 bucket. Then it applies the deep learning algorithm on that Image and produces the output.
- d. Then, it sends back that output with the image name to Output SQS Queue and also stores the result in the Output S3 bucket for persistence.
- e. At the end, it terminates the EC2 instance in which it is running.

## **3. SQS-Continuous-Listener Module:**

- a. This module is the same as the above one except the difference would be that it won't terminate the EC2 instance in which it is running.
- b. It will continuously check for the inputs in the Input SQS Queue using Long Polling that means, if there are no messages in the SQS Queue, it would wait for some time before closing out. It facilitates the quick response to the user.

## **4.2 Instructions for Project Execution:**

### **1. Setup Modules:**

- a. **Setup of Web-tier-EC2:**



- i. While setting up EC2 instance, please refer to the free tier policy of AWS. We have set up an ubuntu based free tier instance on us-east-1 region for the web tier.
- ii. In order to install node js and relevant libraries please use the following commands.
  1. *sudo apt-get update*
  2. *sudo apt-get install nodejs*
  3. *sudo apt-get install npm*
  4. *npm install aws-sdk*
  5. *npm install express*
  6. *npm install multer-s3*
  7. *npm install multer*
  8. *npm install sqs-consumer*
- iii. Upload our node application on the server by using git clone or nano editor.
- iv. Install Java on this web-tier instance by using the following commands.
  1. *sudo yum update*
  2. *sudo yum install java-1.8.0*
- v. Upload the jar file of Controller Module using the scp command mentioned below.
  1. *scp -i "path-to-pem-file" "path-to-jar-file" ubuntu@ec2-address.com:~*
- vi. Create a .aws folder in /home/ubuntu directory and put the access keys into the "credentials" file.

**b. Setup of Application-tier-EC2:**

- i. To set up an app-tier EC2 (SQS-Listener), we have used the given AMI ID. We initially launched an instance using the given AMI ID.
- ii. Upload the jar file of SQS-Listener module on to this instance using the same command mentioned above.
- iii. Create a .aws folder in /home/ubuntu directory and put the access keys into the "credentials" file.
- iv. Add the following command into the crontab so that whenever an ec2 is launched of this type, it will run the jar file automatically.

1. \* \* \* \* \* /usr/bin/java -jar path-to-jar-file.jar
- v. Create the Amazon Machine Image out of this instance and use this AMI ID in the controller module.
- vi. For SQS-Continuous-Listener, Launch a separate EC2 instance using the same provided AMI ID and name that instance as app-instance-running.
- vii. Upload the jar file of SQS-Continuous-Listener on this instance and set up the .aws folder along with the crontab so that when this instance starts running, it automatically starts running the jar file inside.

## 2. Execution Steps:

- a. First, start the web-tier instance and app-tier-running instance from the AWS console.
- b. Now, connect the web-tier instance using the given ssh command to the local terminal and start the node application and controller jar file by using the following command respectively.
  - i. *node index.js*
  - ii. *Java -jar controller.jar*
- c. In order to debug the execution process we have implemented this manually. However, we can configure cron jobs to run both commands automatically as soon as instance starts.