CSE 546 — Project1 Report

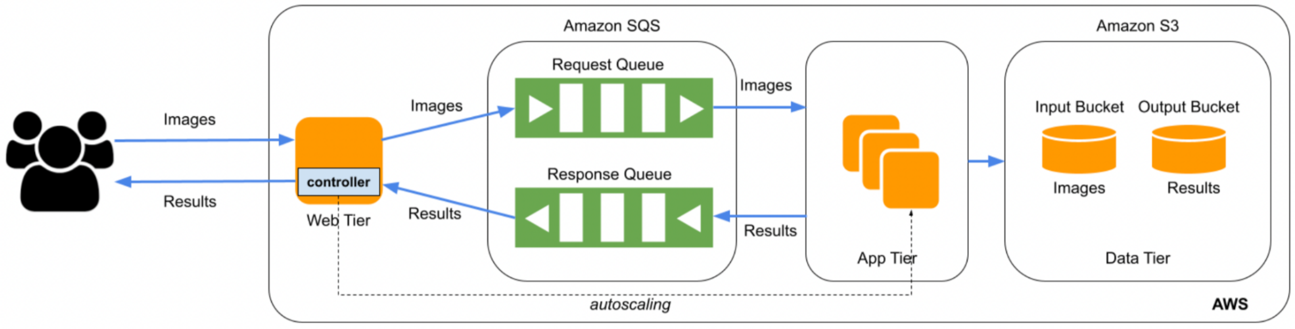
Aniket Devle (1220351418),
Fenny Zalavadia (1221443561),
Jason Jimenez (1215972966)

1. Problem statement

We need a service that uses cloud resources to take an image from the user as input and perform deep learning on this image and return the result of the image back to the user. This service should be an elastic application using IaaS resources from Amazon Web Services(AWS) to provide the facial recognition service to the users while also using cloud resources to perform deep learning on images provided by users. The application should automatically scale in and scale-out on-demand and will be able to handle multiple requests concurrently. AWS is the most widely used IaaS provider and offers a variety of computing, storage, and messaging services which is why it is important to use this for solving the problem. This is an important problem to solve because it will give us hands-on experience with various cloud computing tools to provide a full-stack solution to the problem at hand. It is also important to solve this problem using IaaS resources because it's easily scalable, cost-effective, and usable. Many other applications can be built in the future using technology and techniques used in this application.

2. Design and implementation

2.1 Architecture



2.1.1 Architecture Explanation

- **Web-tier (EC2)**
    - Our architecture begins with the Web-tier which in our case is a user-facing EC2 instance running an Apache Server with PHP. The web tier accepts the image classification requests from the users via an HTTP POST request method.
    - A script constantly running on the web tier takes care of ensuring multiple **application tier EC2 instances** are created based on the length of the SQS request queue. Initially,

    we will only have web-tier running and it will start up the app tier instances based on demand.
  - After accepting the request, the web tier creates a message for SQS containing the name of the file and the base64 encoding of the image and sends this image to the SQS request queue.
  - Web tier receives messages from the **request SQS queue** that were sent from the **application tier** and once the classification is extracted from the message, the classification is sent back to the user that initiated the request.
- **(Amazon SQS)**
  - **Request Queue**
    - **Web tier** is responsible for sending the request as a message through SQS via this request-queue.
    - **Visibility timeout** for this queue sets the length of time that the message will not be visible to other message consumers after being read, and this is set to 5 seconds for this queue.
  - **Response Queue**
    - **Application tier** is responsible for sending the response to the image classification request back as a message via this response queue
    - **Visibility timeout** for this queue sets the length of time that the message will not be visible to other message consumers after being read, and since this takes longer to process than the request messages this is set to 5 minutes for this queue.
- **Application Tier**
  - Once the appropriate number of Application Tier EC2 instances have been started each individual Application Tier will read a single message containing the image name and the base64 string encoding of said message from the SQS request queue and will run the deep learning program, then this image and the classified label from the deep learning program will be stored in S3 for persistent use. Next, a message is created using the output label and the name of the image is into the **SQS response queue**. After finishing these required tasks, the application tier will erase the message from the request queue. The application tier will then repeat this process until there are no more messages available to process.
  - Once the prediction is made, the output label and name of the image is sent into the **SQS response queue**.
  - The above steps are repeated in each application instance until there are no more messages
- **S3**
  - **Application tier** sends image to the Input Bucket which is in charge of holding all images that have been uploaded to the service
  - **Application tier** also sends the image name and classification to the results bucket.

2.1.2. AWS Services used in the project:

1. **AWS EC2**

   It allows users to create Virtual machines on the cloud to run any application. In our project, we are using EC2 instances for implementing the Web-tier and App-tier.

2. **AWS SQS (Simple Queue Services)**

It manages to send and receive messages in the queue. We have used 2 standard SQS (loosely coupled FIFO) in our project; Request Queue and Response Queue namely **face-recogntion-request-queue** and **face-recogntion-response-queue.** SQS also makes it easier to scale in and scale out microservices.

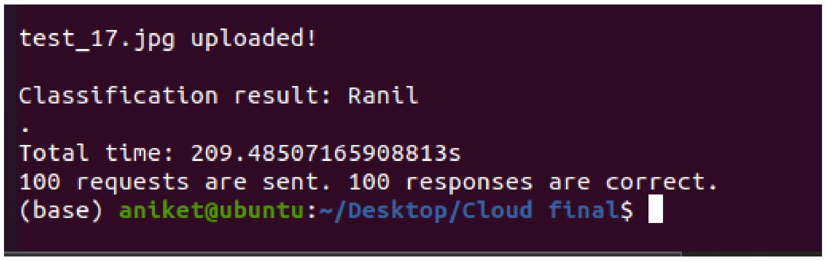
3. **AWS S3 (Simple Storage Service)**

   It is used to store and retrieve any amount of data at any time from anywhere. In our project, we are using 2 buckets that store images and responses in a key-value format namely **face-recognition-s3-img** and **face-recognition-s3-name.**

2.2 Autoscaling

We have implemented scaling in and scaling out as a script that runs as a Linux cron job that runs every 10 seconds and is located in the Web-tier EC2 instance. When the web instance gets new requests, it sends these requests to the SQS **request queue**. The scaling script monitors the **SQS request queue message count** constantly to ensure that there is the optimal number of instances running based on the length of the SQS queue. The way the script determines how many instances should be running is based on dividing the number of messages by 5 and starting these many instances unless it is more than 19. The scaling down is implemented by having the script notice if there should be less instances running based on the number of messages that are in the SQS request queue, if there are it begins scaling down by terminated running instances.

3. Testing and evaluation



```
test_17.jpg uploaded!

Classification result: Ranil
.
Total time: 209.48507165908813s
100 requests are sent. 100 responses are correct.
(base) aniket@ubuntu:~/Desktop/Cloud final$
```

We tested the application by running the provided python script that sends requests to our server and prints out the results. We added the arguments for 100 requests and our ec2 instance's web address. After completing this script, we observed that the time for testing 100 images would vary between 165 - 240 seconds depending on the firing time of instances. The tests would always return the correct evaluation for each image that was requested to the application. We approached the auto-scaling feature by setting a Cronjob on the web server, running every 10 seconds. The time required for processing the queries will be less when the difference between the cronjob run and the request to the server is less. The average response time for our system was around 190 seconds for 100 images.

4. Code

Explain in detail the functionality of every program included in the submission zip file.

Explain in detail how to install your programs and how to run them.

4.1 Code Modules

1. **Web-tier module**

   a. ***Controller.py:*** This controller will scale in and scale out the EC2 instances by monitoring the SQS queue length. This will keep track of all running instances and keep a count of started and pending state instances. It will take the length of the queue from get_queue_ length from Sqs_utils and create new instances using create_new_instance() from ec2_utils, with a max limit of 19 instances. When there are no messages left in Queue, it will auto-terminate the running instances using terminate_instance() from ec2_utils.
   b. ***ec2_utils.py :*** This consists of 2 main functions; one which creates new EC2 instances i.e create_new_instance() and other to terminate the instances , i.e terminate_instance(). The instances will be created based on the ami application tier screenshot.
   c. **imageToSqs.php** : This file contains a PHP function sendimage() used by index.php that takes a file and a name as a parameter. It is responsible for encoding the jpg image and sending the image and name of the image as a message to the SQS request queue. After this it will check the folder that holds the image classifications as a file that derived from the SQS response queue from sqsToFile.php and and returns the result back to index.php. The file is erased once we are finished with using its information.
   d. **sqsToFile.php** : Constantly checks the SQS response queue for responses from the app tier. Once a response/set of responses has been received (up to 10), it saves the message name, classification and SQS message receipt handle and creates an array with the classification and image name. After this, it makes a json encoding of this array and writes it to a file for imageToSqs.php to read and return to index.php. After we are done with the file we erase it from the SQS response queue.
   e. **index.php :** User-facing logic of the web server, responsible for taking the file from the user as input and sending the file and its name imageToSqs.php to get the classification.
2. **App-tier module**
   a. ***app_tier.py:*** The main function of this module is to process the image received by first decoding it using generate_image(), which generates the image from the base64 encoded string. After that, this image is passed to the get_face() from the recognition_face module which processes the image by applying a deep learning program and generates the output label of an image. This is then pushed into S3 buckets by upload_to_s3(), which consists of 2 separate buckets, one which contains images with the image name and another having an image with the predicted name.
3. **SQS module**
   a. ***Sqs_utils.py:*** It is present in both web-tier and app-tier modules. The web tier will push the requests into the Queue and App-tier will push back the response into the Queue. This will establish the connection with SQS queue using queue_url and it consists of different functions like get_message() to receive a message to SQS queue i.e an image; send_message() returning predicted image name into the SQS queue; Once the image is in the request queue is processed, delete_recent() will delete the message from the queue. Lastly, it had get_queue_length() which measures the total number of requests in the queue.

4.2 Instructions to set up and run the project:

   a. **Set up of web-tier EC2**
      i. Web tier is already set up and constantly running but if need to start follow these instructions:
         1. Sign into AWS console
         2. Ensure region is us-east-1

3. Go to the EC2 dashboard.
4. Select Facial Recognition Web tier instance
5. Press insurance state and start

ii. Server and required will now startup automatically with no further setup.

iii. Follow the instructions from the AWS Instruction file, to set up aws configuration by setting the default location as “us-east-1” and image id as mentioned in the Project 1 file for creating instances.

b. Set up of Application-tier EC2

i. Follow the instructions from the AWS Instruction file to setup aws configuration by setting the default location as “us-east-1” and image id as mentioned in the Project 1 file for creating instances.

ii. We need to have boto3, pip, torchvision installed in each of the application tier instances created.

iii. Our application by default runs all the ec2 instances in ‘us-east-1’ region and the names of all the created instances vary between worker-1 through worker 20.

iv. We set up a cronjob to run app-tier.py script on reboot for a created instance. And we created a new AMI out of this instance to make sure that the Cronjob runs for each auto scaling instance.

v. We used the AMI created in the previous step to fire new app-tier instances in the scaling phase**.**

c. Steps to Execute

i. Web server should be started automatically already but if it isn’t we need to start the web tier from the aws console so that it can listen and respond to the https requests from the user.

ii. Ensure that you are running this command in the terminal in the same directory as the images folder and the provided workload generator

iii. Run the command, substituting the number of requests if you want to do less than zero, changing the ec2 instance public domain name, and image folder if necessary:

iv.
```
python3 multithread_workload_generator_verify_results_updated.py --num_request 100 --url "http://ec2-18-207-105-167.compute-1.amazonaws.com/index.php" --image_folder "face_images_100/"
```

v. Now you can wait and the application should do the rest, sending messages to SQS, starting and scaling application tier instances, sending results to s3 and also returning the results. After all this is finished and you have received your classifications. The application will begin to scale down and remove the instances that it created if they aren’t needed anymore.

5. Individual contributions (optional)

Aniket Devle (ASU ID: 1220351418)

1. **Brief description of Project**

   Using AWS IaaS services, this project attempts to create an elastic application that can recognize the Individual in an image using a pre-trained model. We were asked to follow a particular architecture to maintain consistency with our peers. The application accepts uploaded pictures sent in a request form to a web server. The web-server will then process the request, extract the image and image name and push the image in a request queue. Then the app-tier instance will read the image and image name from the SQS queue and classify the name of the Individual using a pre-trained model. Upon classification, the app-tier instances will send the images to two separate s3 buckets, one for the image and name of the image and the second for name and classification. The app-tier instance will also send the classification result back to the web-tier using the response SQS queue. The waiting web tier will read the classification from the response queue and send it back to the user. We also had to scale the system based on the number of requests made to the web-tier instance.

2. **Contribution in the Design phase**

   I contributed to making design decisions on app-tier and data tier in our project. I also made design decisions with handling the data while transferring from web-tier to app-tier(emphasizing on encoding and decoding the image using base64). I also contributed in making design decisions on the data tier, including how and in which format we will save the images and results. I also worked on designing the logic for auto-scaling the app-tier instances upon an increase in requests on the web tier. I also contributed in creating the logic to speed up the web tier as we are only able to get 10 responses from sqs at a time. Moreover, I was also responsible for maintaining the proper communication between all the team members and stepped up when either of them was stuck in some task.

3. **Contribution in the Implementation phase**

   Most of my contribution in the implementation phase was in implementing the app-tier, data-tier, and auto-scaling. I built upon some of the smaller components implemented by my teammate and orchestrated most of the process in app-tier. I built the components to read the message from SQS and convert them to image format. Moreover, I made changes to the existing face-recognition.py file provided to make it return the name instead of printing it on the console. I also implemented the script for transferring the files from app-tier instances and storing them in S3. Upon completion of one app-tier instance, I created an AMI of the created app-tier instance. Finally, I implemented the Auto-scaling script using the AMI created in the previous step. I was also responsible for setting the Cronjob(used for scheduling executions) on both app-tier and web-tier instances.

4. **Contribution in Testing phase**

   During the testing phase, I was responsible for unit testing and verifying results. As most of my implementation was in-app tier I took the responsibility to unit test each and every component of the app tier. Moreover, I made sure that the results that we were getting were correct and were consistent with the predictions provided. In the later stages of the projects, I also helped with testing the cronjobs by stopping the cronjob and manually running the script using the command-line interface.

Fenny Zalavadia (ASU ID: 1221443561)

1. **Brief description of Project**

   This project aims at building an elastic application to recognize the names of an image provided by a user using IaaS services on AWS. The user will first send the request through web-tier which will be pushed into the request Queue of SQS and based on the number of requests, new EC2 instances will be created which will process the request and send the results back to the queue and push the images with its label name into 2 separate buckets of S3. Based on the length of the SQS Queue, the app-tier will scale in and scale out automatically. Different tasks involved in this project were to design the architecture, set up the web-tier and app-tier, and logic to scale in and scale out efficiently and test it. Below is my contribution to each of the phases of the project.

2. **Contribution in the Design phase**

   The design phase involved deciding different components of the AWS to use, how the data will flow through these components based on the architecture, how they will interact with each other and where will the scaling logic be present, where will the deep learning image recognition model be called and so on. So, after having brainstorming sessions with other teammates, I contributed in the part where after having an image in a request-queue in the form of base64 encoded, how will the App-tier read this message and decode it back into jpg format and process it.

3. **Contribution in the Implementation phase**

   The implementation involved many small modules, the major part which I worked on was to make sure that images from the SQS queue are received successfully by the app-tier and it is able to decode and process it by running a deep learning image recognition model on it and generates correct output. To do this, first I created a dummy queue(standard) in SQS which will act as Request-queue, and using the boto3 client I established the connection using queue_url and received the message in the form of a string, which I decoded using the b64decode function. This resulted in generating images in ".jpg" format, which gets passed into the main function to make predictions.

4. **Contribution in Testing phase**

   We performed different types of testing, few involved testing to make sure that we are receiving the correct number of requests as requested by users, to test whether the EC2 instances are created correctly with proper scale-in and scale-out, to verify whether the images were processed on time and predicted correct name and finally the images were stored in S3 successfully and responses were returned back to the user in minimum time. We also performed end-to-end testing with 10 and 100 images. I contributed to this by making sure all the SQS, S3 buckets are empty and no instances are running except web-tier initially. I also helped in the manual testing of receiving an image from SQS and processing it to generate an image and predict the correct name.

Jason Jimenez (ASU ID: 1215972966)

1. **Brief description of Project**

   For this project, our goal was to create a full-stack cloud application that aims to provide a classification for an uploaded image. The application should follow the specified architectural design given to us. This design starts with the user. The user should be able to upload the image to an Amazon EC2 instance that they want to be classified as a JPG file and the application should send the file to an Amazon SQS request queue. Another Amazon EC2 instance that will run as the application tier should receive this message from SQS and run the classification on it. After classifying the image it should send the results to 2 different S3 buckets. It should then send the result back to a response Amazon SQS queue. The web tier running on the first EC2 instance will keep searching for the result of the image from the response SQS until it receives it. The web tier will then return the result back to the user. The application should scale-out by starting multiple EC2 instances whenever there are a lot of request messages sent to the queue.

2. **Contribution in the Design phase**

   I contributed to designing the web tier instance and how it would send the messages to the SQS. I also created the design of how the web tier should constantly check SQS to and write the results to a file on the server so that it can promptly be returned to the user. I also contributed in coming up with the design idea for the base64 encoding of the image sent by the user so that it would be able to be sent to SQS for the application to process.

3. **Contribution in the Implementation phase**

   For implementation, I contributed to several things. First, I created an Amazon account that would be used for the implementation of the project. I then created roles in this account so that the rest of the team would have access to the same resources. After this was set up, I created the Amazon EC2 instance that would be used for the web server and I set it up so that it would have an Apache Web Server running on it using PHP. I set this server up so that it would start every time the instance starts. After doing this, I set up the support for the server to receive a file from the user via an HTTP post method and would return an output whenever the web tier would know an answer. After this, it was time to upload the required functionality of sending the image file and name for SQS to process it. I installed the AWS PHP SDK so that I would be able to send the images from the web server to the SQS request message queue. After installing the required SDK, I implemented the functionality of sending the message to SQS for the app tier to process the message and reply with the classification. I implemented a script that will run every 10 minutes via a linux cronjob that will constantly check the queue and write the results to a folder that the web tier will read. I then implemented the logic for the web tier to ready this folder until it finds what it's looking for and returns this to the user.

4. **Contribution in Testing phase**

   When the team was finished with both web tier and the app tier, I contributed to testing the flow of the application. I did this by sending a request to the web tier and ensuring that it seamlessly flowed through the SQS queue to reach the app tier created by the other teammates. After this was tested, I contributed to testing the whole flow of the application to ensure that it runs as expected with no errors. Every time we changed a certain aspect of the application, I made sure that it was tested thoroughly.