# CSE 546 — Project1 Report

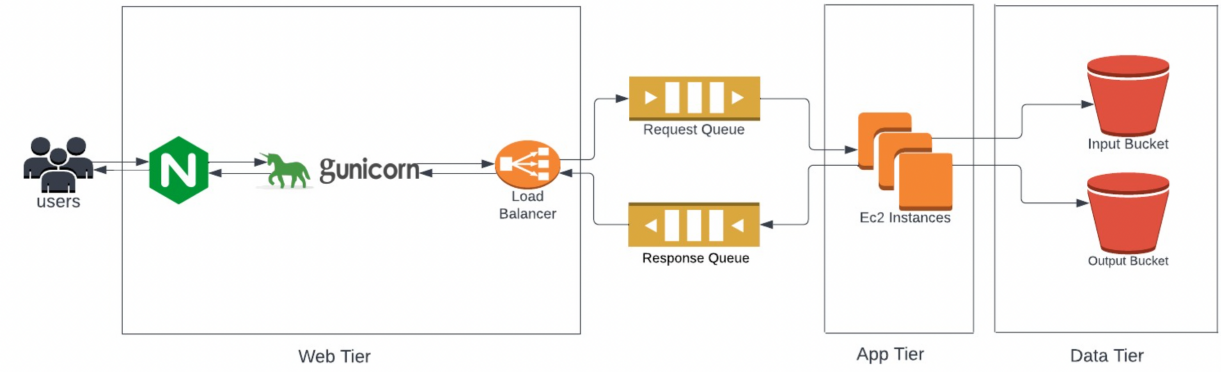
*Gompa Divya (1221003043)*
Nithin Jayakar Padala (1220061427)
Sai Varun Vaka (1223499368)

## 1. Problem statement

A cloud application which provides face recognition as a service to users who gives images as input. The application uses deep learning model to classify the images by using cloud resources. The application uses IaaS resources from Amazon Web Services (AWS) such as EC2 Instance, Simple Queue Services (SQS), Simple Storage Services (S3). The application hosted on cloud is Autoscaling application which automatically scale-out and scale-in as per the demand. By using cloud resources and Autoscaling, the application is cost-effective with better performance and high availability.

## 2. Design and implementation

### 2.1 Architecture




Architecture diagram of implementation with major components

### 2.2 Implementation

- **Web Tier:**

Web Tier includes the project's user interface, which transfers images to the App Tier and accumulates the predictions from the face recognition backend. The web tier end to end flow:

  - Users will upload images to the website.
  - The images are routed to the Nginx web server via a POST request. Plus, Nginx serves as a reverse proxy for Web Server Gateway Interface called Gunicorn, and requests are routed through the HTTP server.
  - Plus, the load balancer is also hosted with a web tier with the help of Nginx and Gunicorn.

    - The web tier uploads the images to the standard SQS queue, and the load balancer scales out or scales in according to the demand.
    - The web tier retrieves the final prediction from the App Tier and portrays it into the user interface.
- **Request Queue:**
    - It is a simple standard queue service offered by Amazon Web Services to handle the requests coming from the user. Autoscaling of EC2 instances to run the application is based on the requests in this Request Queue.
- **Response Queue:**
    - It is a simple standard queue service offered by Amazon Web Services to handle the results coming from the Deep learning model. The results in this queue used by the controller to display these results on application user Interface.
- **App Tier:**
    - App tier instances gets created when there are messages in SQS queues.
    - It receives the encoded string output from the request queue. After taking the encoded string, we convert it into decoded bytes and save it as image file.
    - This image file is stored in the ec2 instance and face recognition service is called by sending the image path.
    - The output of the service is then decoded and stored as string and sent to the response SQS queue and the message which we received in the request queue is deleted.
    - In the App tier, we send the image name, image file and predicted output to the Data Tier layer where results will be stored in the S3 buckets accordingly.
    - When all the messages are processed and the SQS is empty then individual instances get terminated automatically or else the next messages will be processed in the App tier.
- **Data Tier:**
    - Input Bucket:
        - Simple Storage Service (S3) offered by Amazon Web Services to store the user uploaded images. It stores as Image name Key and Image file as file object in S3.
    - Output Bucket:
        - Simple Storage Service (S3) offered by Amazon Web Services to store the prediction results of the uploaded images. It stores Image name as Key and prediction result as object in S3.

## 2.2 Autoscaling

Autoscaling of EC2 instances are completely based on the number of requests we have in Request SQS queue. Our application uses Master – Slave architecture to implement Autoscaling. When users send requests, Request SQS queue receives the requests to process. When requests queue has some request to process, Master will create a slave EC2 instance to process the request. When slave completes the process of one request, it will check for another request SQS queue to process more requests. When no more requests available in the queue to process, slave will terminate itself. If requests come after all slave instances terminated, Master creates slaves as per the requests again. Our application is built in a

way to create at most to 19 slaves by the master. To scale-up, Master creates instances, and it runs in the Web-Tier. To Scale-down, Slave instances terminate itself, and it runs in App-Tier.

Scale-up and scale-down works as below:

Case 1:

Number of Requests to process is 1 <= Requests < 20.

Number of new slave instances created = (Number of Requests – Current Running slave instances).

Case 2:

Number of Requests to process >= 20.

Number of new slave instances created = (19 – Current Running slave instances).

Case 3:

Number of Requests to process < Current Running Instances.

Slave instances terminates itself gradually when no other request to process.

If all the requests are processed and currently no requests in Request SQS Queue, Slaves gradually terminates itself and reach to zero.

In simple terms, when there are requests to process, App-Tier instances gets created and when no requests to process App-Tier instances gets terminated itself gradually. With this, we can achieve best performance with cost effectiveness and high availability.

## 3. Testing and evaluation

All the modules (web tier, app tier, data tier) were tested thoroughly in the development stages. Every time we integrate two modules, we test the flow of the modules to make sure it works according to the requirement.

- Web-Tier Testing:
    - Tested Django project with images upload and with form submission.
    - Tested each image request went successfully to Request SQS queue.
    - Tested Master instance running all the time and creating App-tier instances as per the requirement.
- App-Tier Testing:
    - Tested App-tier instances created successfully as per the demand based on Request queue.
    - Tested sending input uploaded images and prediction results to s3 buckets.
    - Tested App-tier instances terminated gradually itself when no request to process.

- Data Tier:
    - Tested uploaded input images are same as user uploaded images.
    - Tested image predictions in S3 bucket with classification result for accuracy.

**Manual Testing:**

All the flow of the project is tested with the 100 images:

| **Test Case** | **Average Time in Seconds** |
|---|---|
| Upload images to request SQS queue | 10 |
| Scale-out App-Tier instances | 120 |
| Face Recognition algorithm process and send results to response queue and S3 and Scale in automatically | 150 |
| Display results to Application User Interface | 30 |
| Total Time | 310 |

All the flow of the project is tested with the 10 images:

| **Test Case** | **Average Time in Seconds** |
|---|---|
| Upload images to request SQS queue | 10 |
| Scale-out App-Tier instances | 60 |
| Face Recognition algorithm process and send results to response queue and S3 and Scale in automatically | 120 |
| Display results to Application User Interface | 30 |
| Total Time | 220 |

**Multi-thread Workload generator Testing:**

10 Multi threads with one image in each thread: 250 seconds

100 Multi threads with one image in each thread: 309 seconds

## 4. Code

**Web Tier:**

- Web Tier consists of the following files:
- face_recognition.html: The Html file that renders the user interface in the browser
- queue_handler.py: The queue operations are handled in this file, such as sending and receiving messages to the queue and the number of messages in the queue, etc.
- views.py: This file sends the messages to the request queue and receives predictions from the output queue.

**App Tier:**

App Tier consists of the following files:

- Slave.py: Here we check the messages present in a queue if messages are present, we process them to service file else the instance is terminated by taking instance id.
- Service.py: Here messages are received from the request queue. As the received message is in encoded format it is decoded and saved as image file. This file is sent to the provided face_recognition algorithm which in turn produces prediction result. This result is sent to response queue and output bucket.
- Ec2_handler.py: All instance related operations which depend on ec2 resources are performed here.
- Queue_handler.py: All queues related operations like sending and receiving messages are done here.
- Literals.py: Constants related to App Tier are declared here.

**Data Tier:**

- Data tier does process for storage into S3 buckets.
- It stores the image name as Key and image file as Value in Input S3 bucket.
- It stores the image name as Key and predicted output as Value in Output S3 bucket.
- S3_handler.py: Storing the data in the s3 input bucket and s3 output bucket is done here. Both the operations performed are called from service.py which is in the App Tier.

**Steps to Execute:**

Setup and Execution

Web Tier Instance

Setup

- Instance is created with the ubuntu AMI
- This instance consists of the web tier package. Or else send the code through any file transferring applications.

- Name it as Web-instance1.
- Instance also consists of a service which invokes the scaling up of instances method.
- Install the requirements with below commands:
- pip install boto3==1.21.14
- pip install botocore==1.24.15
- pip install Django==4.0.2
- pip install ec2_metadata==1.1.0
- pip install Pillow==9.0.1

Execution

- Launch the instance with the Web Tier AMI
- Start the server by the below command:
- Python3 manage.py runserver 0.0.0.0:8000

App Tier Instance

Setup

- Launch the instance from the Linux AMI with the custom AMI which consists of the app tier package.
- Due to the auto script present for running the instances in the AMI, instances will run according to the scaling logic present in the web tier.

Execution using workload-generator.py file:

Enter the below command:

- In order to run 10 requests at a time. Use the below command:

python workload_generator.py --num_request 10 --url "http://23.23.17.134:8000/" --image_folder "C:/Users/Projects/face_images_100/face_images_100/"

- In order to run 100 requests at a time. Use the below command:

python workload_generator.py --num_request 100 --url "http://23.23.17.134:8000/" --image_folder "C:/Users/Projects/face_images_100/face_images_100/"