CSE 546 — Project1 Report

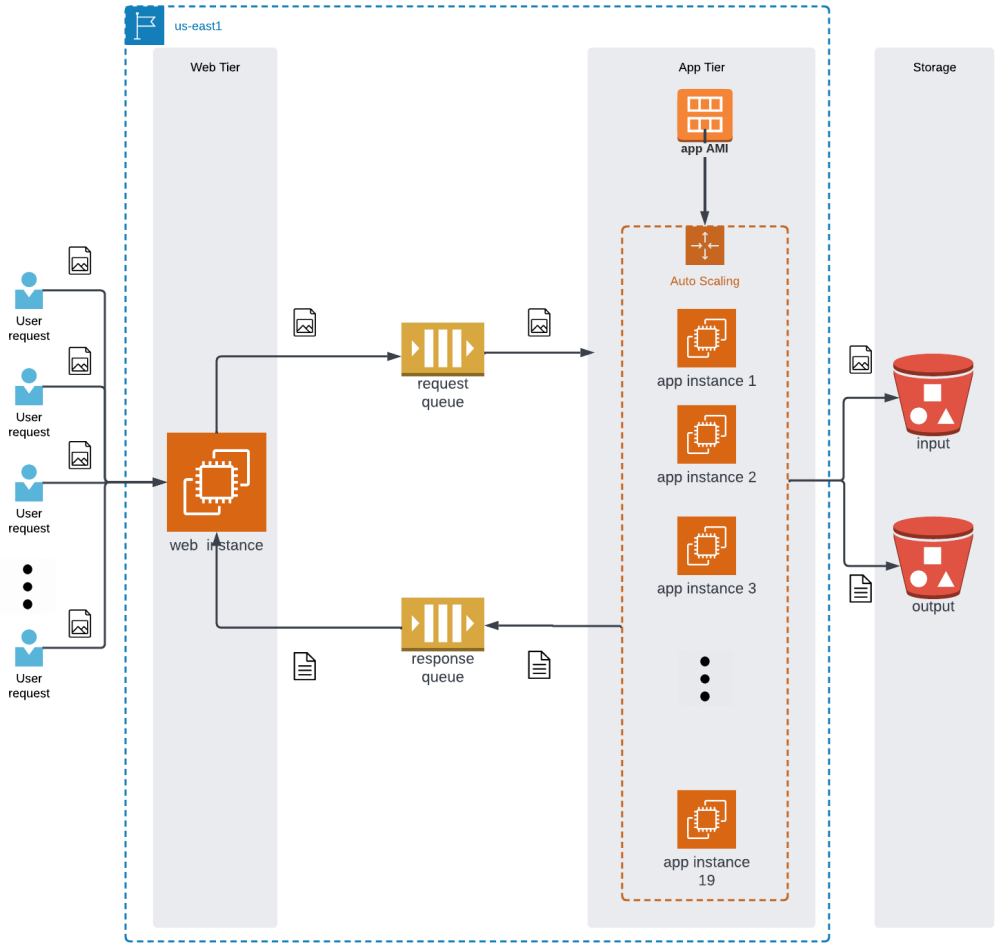
Krima Doshi (1222303329)
Nisarg Shah (1222257699)
Pankti Mehta (1223202305)

1. Problem statement

The aim of this project is to build an elastic web application, which utilizes cloud computing services Amazon Web Services (AWS), with objectives of reliability, system elasticity, cost-effectiveness and real-time usage. An image-recognition application has been created as a REST service on the Cloud for the clients to easily access. The application takes images as inputs and returns the correct outputs through a deep learning model using AWS resources for all the processes. The basic tasks included in the system are the RESTful API, an implemented load balancer that scales in and scales out EC2 instances at App tier according to the demand of the user.

2. Design and implementation

2.1 Architecture



The architecture consists of using three major AWS components, namely, EC2, SQS and S3.

EC2

Amazon Elastic Compute Cloud is a resizable computing service provided to consumers. Consumers are charged per hourly use of the computing resource depending on the instance size and resources. It provides a large variety of virtual machine environments like Ubuntu, Linux, etc. to build the instance. It also provides an option to create an Amazon Machine Image(AMI) from an ongoing EC2 instance and instantiate an EC2 instance from a given AMI. [Note: EC2 and AMI are regional, i.e. resources on the same region are accessible to each other.]

In our project, EC2 is used in both the web tier and app tier.

The web tier is created in an Ubuntu t2.micro instance with python installed. The web tier consists of a flask service for the front end to receive requests. EC2 by default has a Public IP assigned to it when it is instantiated, to which all requests from the client are sent. After receiving the request from the client, the program then sends the request to the SQS request queue. We have implemented the Memcached system for maintaining the states of the results for all images. Filename is the key and the output of that image is a string value. Every request will wait to get a result from the cache for the image provided to it as input, and as soon as the result is found in the cache key value store, it will return the output label to the client.

A separate thread is also created in the web tier, in which responses are fetched from the SQS response queue continuously and the cached key value store is updated with the results. These results are being used and sent back to the client as the result.

In another program on the web tier, an ec2 instantiation script is run in the background, using which app tier ec2 instances are dynamically instantiated (scaled up) on demand. Here, the number of instances are created as per the number of requests received in the SQS queue in such a way that a maximum of 19 instances can be active at a time, i.e. if I have 10 requests in SQS request queue, 10 app instances are created but if I have 50 requests in the SQS queue, there will be only 19 such instances active. This script also terminates all stopped app instances if requests have not been received for 2 minutes.

The app tier is created using an AMI created on a LInux instance. The AMI has all python dependencies and the face recognition script pre-built. The app instances are created by the web tier and configured with AMI, security group, pem key name and USER DATA. The USER DATA runs in the root mode on startup. Our USER DATA pulls the latest python script to run from s3 and executes it. The python script reads messages from the SQS request queue until there are no more messages in the SQS queue or all of the messages are in flight (being read by other app instances). When a message is read, the image passed as a binary value in the message is stored on a file in both s3 and on the local file system of the EC2. The face recognition script is then executed with the local image and the output string is saved to S3. The answer label is also sent to the SQS response queue along with the image name. When no more messages are found, the EC2 instance calls a function to shut down itself.

SQS

Amazon Simple Queue Service is a simple distributed message queue service provided by AWS. It is mainly used to decouple the sending and receiving components in an architecture, which in our case are the web tier and app tier. Message requests are sent from the web tier to the SQS request queue and then read from the SQS request queue by the app tier. SImilarly, messages are written to the SQS response queue from the app tier and then read by the web tier.

SQS queues have two types: Standard and FIFO. The FIFO queue is an exact delivery queue, which has a limited throughput of 300 transactions per second and no duplicates. Standard queue on the other hand has nearly unlimited transactions per second, at-least once delivery queue and possible duplication. In our use case where duplicates are allowable but transactions need to be fast, a standard queue seems to be a better choice for both queues.

SQS queues in AWS have various features like visibility timeout, delay seconds, etc. We use the visibility timeout in our program to read our message and wait for a maximum of 15s for it to be deleted before the message becomes readable to other resources. Since the face_recognition script is able to complete one run in seconds, 15 seconds can be set as a visibility timeout for the SQS request queue. Since 2 seconds has been kept as a wait time between two reads of the SQS response queue in the web tier, its visibility timeout can be anything greater than that. If any process takes longer than the visibility timeout, we consider it to have failed and the message is made available to be re-ingested.

S3

Amazon Simple Storage Service is a global storage service provided by Amazon. It stores data in the form of buckets, files and folders. Access, versioning and archiving of buckets can be set and modified. The service is charged by the amount of storage used and the number of PUTS and GETS of the files. We use the S3 to store the input images and output files in their respective folders on the same bucket.

2.2 Autoscaling

The autoscaling logic is generated in two parts:

1. Scale up logic: The web tier creates app tier instances using the following conditions:
    a. When number of requests in SQS queue (Q) > number of active app tier instances(A):
        i. For S number of existing stopped instances, if S>0, start min(S, Q-A) instances.
        ii. If S+A<19, create min(Q,19)-S-A new app instances.
        iii. Given S+A==19, do nothing.
    b. For a given time interval T after the last request time T0, if A>=Q and S>0 and current time T1-T0>T, terminate all stopped instances
    c. No action required
2. Scale down logic: The app tier instances shut down themselves when no further requests are found in the SQS request queue.

3. Testing and evaluation

1. In our initial testing, we tested for 10, 25, 100 images. For 10 and 25 images, the output response was perfect, but in 100 images we were missing some of the responses back to the workload generator. The implementation on which this was tested had a global variable which was mapping the request and the output and we had a thread which was fetching the responses from the SQS. But in python, the threads were not able to update the shared variable in a correct way. Process of testing was as follows.
    a. Testing:
        i. Send a request using the workload generator using 100 images provided to us.
        ii. Checked whether the web tier is able to accept all 100 image requests.
        iii. Checked the number of messages in the SQS request queue and the input/output stored in the S3 bucket.
        iv. Based on the number of SQS messages, the load balancer was tested for starting and stopping the app-tier instances.
    b. Evaluation:
        i. Missed some responses in the workload generator (local), although flask got all 100 outputs and the whole pipeline completed successfully.
        ii. S3 buckets had all 100 input and output images.
        iii. SQS queue message count was perfectly maintained and the web-tier was able to fetch responses.
        iv. The web tier was able to scale up app ec2 instances when queue length increased until it reached 19.
        v. The web tier successfully terminated stopped instances after 2 minutes of inactivity.
        vi. The app tier was able to stop itself after the request queue became empty.

2. Due to the above issue, we changed the approach and decided to use the cache key value store for the mapping. Due to the single persistence storage, we were able to get all the responses back to the workload generator in a correct manner. Testing process was the same as the above.
    a. Testing:
        i. Send a request using the workload generator using 100 images provided to us.
        ii. Checked whether the web tier is able to accept all 100 image requests.
        iii. Checked the number of messages in the SQS request queue and the input/output stored in the S3 bucket.
        iv. Based on the number of SQS messages, the load balancer was tested for starting and stopping the app-tier instances.
    b. Evaluation:
        i. Got all 100 responses back to the workload generator and output was correct.
        ii. The execution time for 100 requests scaled down to <200 s.
        iii. S3 buckets had all 100 input and output images.
        iv. The web tier was able to scale up app ec2 instances when queue length increased until it reached 19.
        v. The web tier successfully terminated stopped instances after 2 minutes of inactivity.
        vi. The app tier was able to stop itself after the request queue became empty.

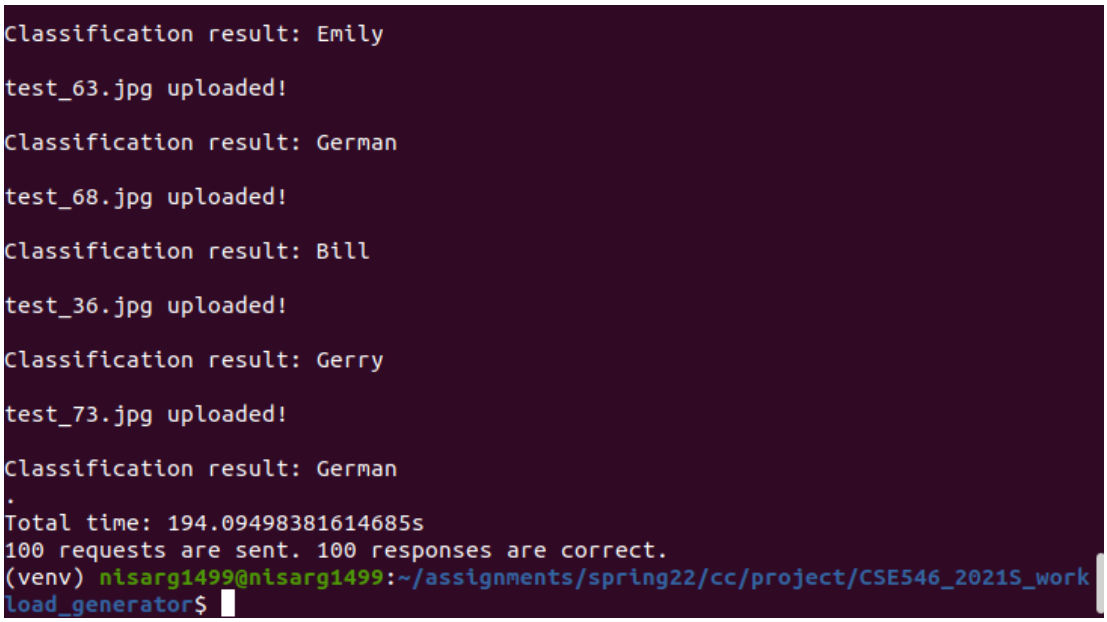


Figure: Successful output screenshot of workload generator

4. Code

4.1 Files

4.1.1 Web Tier

4.1.1.1 main.py:

This file contains the flask web server code and a thread created for fetching the messages from the SQS response queue.

- Flask Server
  - API for uploading the image sent from the workload generator, sending images to the SQS request queue.
  - Check for the output of the request image. If the output is available in the key-value store then send that as a response to the workload generator.
- Threading
  - Continuously fetch responses from the SQS response queue and update the key-value store for the image and the output.

4.1.1.2 ec2_instantiate.py:

This script implements the autoscaling logic. It gets the size of the request queue and number of instances in the running/pending state[Active] as well as stopped instances[Inactive]. Then it keeps track of not creating more than 19 instances at a time using the logic specified in section 2.2. It also terminates the stopped instances if the last request was received from the client to the web tier more than 2 minutes before.

4.1.1.3 USER DATA:

The USER DATA script contains commands to be executed on bootup of the app EC2 instance. It is set in the ec2_instantiate.py file.

4.1.1.4 requirements.txt:

To be installed as-is when creating a new web tier as it has the compatibility of flask and mem cache configured.

4.1.2 App Tier

4.1.2.1 receive_messages.py:

This file contains the logic to run the app ec2 logic. It makes use of boto3 to connect with and use AWS services. We create boto3 resource handlers for the S3 bucket, the SQS request queue and the SQS response queue.

Looping until the SQS request queue is empty, we first check if the face_recognition.py file exists in the specified location. If it does not, it prints that the "EC2 is still initializing". If it exists, our EC2 is properly booted up and we are ready to execute our code. Firstly, we receive one message from the SQS request queue which would contain the image name and binary value. We save the binary value as an image file on the local as well as on s3. Then we pass the image path as an argument to execute our python face recognition program. The output is stored in a file on s3 and transmitted to the SQS response queue.

When there are no more messages in the queue to read, the instance executes the command to stop itself.

4.1.2.2 initialize.sh (init.sh):

This script would be run to set up the python environment initially from the deep learning instance provided. We only use it to set up our AMI initially from the deep learning instance. It is not used after an AMI has been created.

4.1.3 Workload generator.py:

This script is provided by the TAs and this is used for sending concurrent requests to the application.

4.2 Setup and Execution

4.2.1 Setup

- Web Tier
    - Create and start an EC2 instance using the ubuntu image.
    - Download the code of the flask and ec2_instantiate code in the instance.
    - Install the dependencies using requirements.txt
    - Install python memcache using pip command.

    - Assign an IAM role to it with permissions to SQS and EC2.
- App Tier
    - Create an EC2 instance using the image given by the professor.
    - Download init.sh and execute it.
    - Create an AMI from this app-tier and set the ami-id in ec2_instantiate.py
    - Assign an IAM role to it with permissions to S3, SQS and EC2.
- S3
    - Create a bucket with an ‘input’ and ‘output’ folder
- SQS
    - Create two queues, one for request and one for response

4.2.2 Execution

- Web Tier
    - Start the flask server using `python3 main.py`.
    - Also start the ec2_instantiate script using `nohup python3 ec2_instantiate.py &`
- Workload generator
    - Execution command from local:

```
python3 multithread_workload_generator_verify_results_updated.py
--num_request NUM_REQUEST  --url URL --image_folder IMAGE_PATH
```

5. Individual contributions

5.1 Krima Doshi (1222303329)

Design

- The primary architecture and workflow of AWS services, technologies used, script distribution of each task was presided by me. The design of modules for the app-tier and development of the scaling logic for the app-tier was done by me along with the communication of the scripts in the instances.
- In the app tier, I developed an automation pipeline to spin up app ec2 instances on demand according to the number of requests received by the web tier. The app EC2 instances were configured to execute the program on startup and shut down the instance when completed.
- I implemented the auto-scaling of the app instances to control the app EC2 lifecycle, i.e. the scaling up/creation of app EC2 using AMIs, starting existing stopped EC2 instances and terminating idle stopped EC2 instances when no new requests have been obtained in the web tier.

Implementation

- Brainstormed the upper level technologies, libraries and workflow of resources in AWS
- Setup of AWS resources required for project.
- Creation of IAM roles for resources, attachment of access policies to them and attaching roles to their respective resources.
- Web tier load balancer:
    - Wrote a script for controlling the scale up and scale down of app-tiers EC2 instances along with the start and stop of those instances.
- Automation in the app-tier.
    - Create App AMI from App EC2
    - Configured USER DATA script to execute the program automatically and instantly on boot up.
    - Added the script to close the EC2 instance when there are no more messages available in the SQS queue at the end of the app-tier EC2 script.

Testing

- The load balancer testing phase was conducted in three phases:
    - Unit testing: Instantiation of unit app ec2 modules from AMI and calculation of the scaling factor for instance spin up in web ec2 were tested. Tests among them included testing each boot time script execution output and auto self-stopping app instances.
    - Integration testing: The ec2 instance creation script was deployed on the web-tier and parallely tested along with the web-tier flask deployment script. The scaling of app instances was tested by changing the number of requests received and checking the number of terminated instances after 2 minutes of inactivity, The boot up script and shut down state of multiple app instances in parallel were checked.
    - End-to-End testing: Testing of the end-to-end pipeline was done in the team with different configurations like images, number of requests and checking the results of the face recognition sent by the app-tier and received by the web tier.

5.2 Nisarg Pranav Shah (1222257699)

Design

- I collaborated with my teammates to design the architecture of the system. I worked on the design of the web-tier and configuration of the instance for the web tier. Along with this I was involved in the design of the modules which will reside in web-tier as well as app-tier and the approach for integration of the load balancer module.
- Primarily, I contributed in the design, implementation and integration of the web tier. I decided the flow of the request, how it will be mapped to the output for sending correct responses back to the client and how to fetch the responses from the SQS queue. Furthermore, I have contributed in deciding the design required for interaction of scripts with web-tier.

Implementation

- Required AWS setup for web-tier security groups to serve requests on specific ports and specific IPs.
- Install all dependencies and figure out library compatibility for memcache.
- Implementation of Web tier as a flask service.
    - Implemented an API which will accept the concurrent request, send the image in the SQS request queue and then wait for the output of that image in the key-value cache store. As soon as the result is updated in the cache, the output will be sent as a response to the workload generator.
    - Delete the received message from the SQS response queue.
    - Implemented a thread which will run in parallel with the flask web service and a function will be called continuously to fetch responses from the SQS response queue. As soon as any response is received, an entry of <image, output> will be created in the cache which will be used by the flask API to retrieve the result of a specific image.

Testing

- I did testing of the web-tier locally as well as on the ec2 instance. The testing phase was conducted in three phases: unit testing, integration testing and end-to-end testing.
    - Unit testing: Every small module/code was tested on the local machine to check the functionality such as inserting request in SQS queue, uploading file to the server, fetching message from SQS response queue, delete message and threading in flask.
    - Integration testing: As soon as other modules were created, we integrated those and then tested the web-tier with some integrations such as working of ec2 instance create script along with flask service.
    - End-to-End testing: Testing of complete flow was done from sending requests to the web-tier to receiving the correct responses along with scale-in and scale-out of the instances as per requirements.

5.3 Pankti Sachinkumar Mehta (1223202305)

Design

- First of all, after brainstorming various ideas, my primary task was to come up with the designs for integrating the loosely-coupled App-tier and Web-tier, and create an algorithm for the load balancer.
- My main contribution was that I created the architecture of the app-tier and designed the flow of the mapping of the inputs and outputs by Web-tier and App-tier instances.

Implementation

- Required setup of AWS resources
    - Creation of a web-tier ec2 instance and IAM roles, Creating SQS queues, S3 buckets and understanding roles/security groups.
- Developed code/scripts for reading messages from the SQS queue along with the required information.
- App-tier modules implementation.
    - Developed code for storing the input image to the S3 bucket from the app-tier.
    - Execution of the face recognition script with the image read from the SQS request queue.
    - Save the output generated from the script to the S3 bucket for persistent storage.
    - Returning output to the SQS response queue along with the required parameters.
    - Calculate the number of messages present in the input SQS queue to decide whether to terminate the instance or not.
    - Terminate instance after retrieving instance-id.

Testing

- I tested the back-end and front-end extensively locally and in web-tier. I also tested a few functionalities of the app-tier.
    - Unit Testing: I tested each module independently (local server) to ensure the correctness of each functionalities. For instance, I tested functions like uploadFile (upload to S3), sendMessage (send images to input SQS queue), and receiveMessage separately.
    - Integration Testing: I tested the flow of the app tier after setting up all the code modules on the app tier for various APIs.
    - End to End Testing: We tested end-to-end flow by checking outputs in SQS queues and S3 buckets as well as the scale-in and scale-out functionality. We checked the application with different parameters.