CSE 546 — Project1 Report

Arindam Jain — *ajain243@asu.edu* — 1223458106

Panemangalore Srikanth Kini — *pkini2@asu.edu* — 1222310674

Bhavani Mahalakshmi Gowri Sankar — *bgowrisa@asu.edu* — 1222186719

1. Problem statement

A web-application service receiving a large number of requests from one or more clients needs to be able to process them quickly and efficiently. These requests may take a long time to execute and return results to the consumer of the application when there is a single server operating with limited resources. To solve this problem, we use Amazon Web Services' IaaS resources such as EC2 instances, Simple Storage Service(S3), Simple Queue Service(SQS), and the AWS SDK(boto3) that provide access to these resources and the capability to manipulate them, to meet the real-time demands of the application, handle multiple requests concurrently and return the results as rapidly as possible and without losing any requests. We use it to provide image-recognition to the consumers of our application.

2. Design and implementation

2.1 Architecture

Amazon Web Services (AWS) services used in the project:

1. **AWS Elastic Cloud Compute (EC2)**

EC2 allows customers to establish a virtual computer in the Amazon cloud to execute any application. In our project, we deploy an application consisting of the web-tier and app-tier that utilize EC2 resources. App-tier instances are invoked using an Amazon Machine Image (Linux images which are pre-configured with customizations and required code).

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

As a part of our project architecture, we use SQS for buffering outgoing and incoming messages in transit between the web-tier and the app-tier. This helps our application scale seamlessly as SQS helps in buffering requests and decoupling the web-tier from the app-tier.

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

For consistency, we use S3 to store results from App instances. Through application programming interfaces, S3 enables scalable storage. Any type of files may be stored in S3 buckets using the (key, value) object format.

Frameworks utilized:

Our web-tier consists of a Flask application server which uses the Python boto3 framework to programmatically access SQS. A controller script written in Python also runs in the web-tier that implements the auto-scaling logic and scales out app-tier instances by using the length of the request SQS queue. The app-tier instances use the torch library along with a deep-learning model (MTCNN) present in the deep-learning framework obtained through a community AMI.

Architecture Explanation

An EC2 web-tier instance (1 instance) is initialized to handle incoming requests from users of the application. When the user makes a POST request to upload the image, these queries are funneled through an SQS request queue and consumed by EC2 machines at the application tier. The web instance controller script scales out the app instances when the request queue length increases. To determine whether more application layer instances are necessary, a load balancing algorithm is applied. For scaling out, we have upto 19 EC2 instances in the free tier at our disposal . The application tier instances use a deep learning model to classify images from the request queue and return an output label. For the sake of persistence, these input images and their corresponding results are saved in S3 storage in two separate buckets in the data tier. To manage speedy responses, we have one web instance running while the rest of the resources are dedicated to the back-end. The controller in the web-tier is in charge of launching the additional app instances in proportion to the load. When app instances do not see messages in the request queue, they shut down automatically after 60 sec. Therefore, scaling-in is handled by the app instances themselves. Once the resulting classifications are obtained for the images, they are pushed into a second SQS response queue, which is consumed by the web instance. The web instance reads messages from the queue, identifies the response for the request obtained using an identifier, deletes the message from the queue and returns the result back to the user.

Fault Tolerance:

The image is encoded as a base64 string along with a unique identifier for each request and sent to the request queue by the web instance. It is picked up by an app instance. When an app instance reads a message from the queue, a visibility timeout (30 seconds) prevents that message from being read by other app instances. We set the visibility timeout setting to account for the longest duration an app instance may take to return the classification of the image. So, if anything catastrophic happens to the app instance and it terminates before processing the message, that message in SQS will be available to other app instances after the visibility timeout. If the deep learning model is successfully run on the current app instance, the message is deleted from SQS.

<u>Architecture Diagram:-</u>

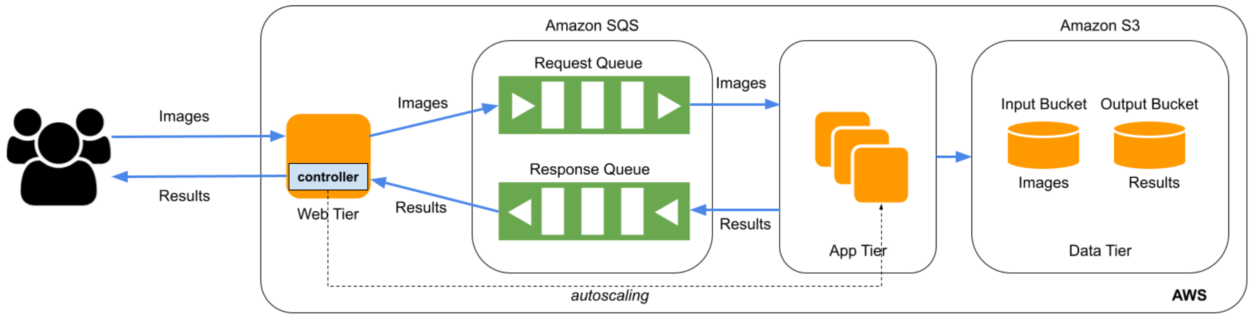

Figure 1.

2.2 Autoscaling

Scaling in and out is done at the application tier and web tier respectively. When the web instance receives a new request, it puts it into the request queue as shown in Figure 1. The most difficult part of auto-scaling was to create an environment that is thread-safe. As in our web application there can be concurrent requests served by different threads, and they individually cannot do auto-scaling as they don't share a memory, so keeping a shared variable that keeps track of running instances is not possible.

The first thing we needed to ensure was that our scale-out logic does not try to create instances beyond the number 20. When there is a request on the web-tier instance, it is pushed to the request queue and on the basis of the length of the request queue, we check every 5 sec to perform auto-upscaling of app-tier instances and spawn ec2 instances with custom AMI containing app-tier code.

Auto Scale-out logic:

Message in Input SQS	App tier - Instance
1-5	1
6-10	2
11-15	3
16-20	4
21-25	5
26-30	6
31-35	7
36-40	8
41-45	9
46-50	10
50+	19

Table 1.

During app-tier instance scale-out, we first calculate the required number of instances as given in Table 1. After getting the required number of needed instances, we subtract it from the number of already running instances, check if any stopped instances can be started or create new instances when there are no stopped instances.

As part of scaling out, our custom AMI which we use to create the app instances has the capability of running the python application using a cron-job every time it boots up.

Auto Scale-in logic:

The logic for scaling-in works at the application tier. After an application tier instance predicts the output label of a given image, it pushes the output to the SQS response queue (unique Identifier and the classification) and also puts the classified key-value object pair in the S3 result bucket. The app instance repeatedly polls the request queue for messages and processes them. If there are no messages in the queue, the app-tier instances will wait for 60 sec using sleep time before polling the queue one last time and if there are no messages, it shuts down. So app instances run a loop till they are able to find messages in the request queue, and if not, then automatically stop itself. Thus the app instances which are spawned by the web-tier scale in by themselves when there is no load.

3. Testing and evaluation

Message in Request Queue	Instances Running	Time
4	1	73 sec
10	2	66 sec
50	10	126 sec
100	19	170 sec

Table 2.

We performed intensive testing by initially sending 4 requests to the request queue which was able to finish processing in 73 sec with a single instance. For 10 requests, according to our auto-scaling logic, two app-tier instances need to be running and we observed the same and it was able to complete the request in 66 sec, faster than sending 4 requests as there are two instances running. Then we tested with 50 requests, for this case, we have two app instances in a stopped state and we observed that these instances started running along with 8 new instances and it finished processing in 126 sec. Finally, we tested with 100 requests, for this we already have 10 instances in the stopped state, these began running along with 9 new instances, so a total of 19 instances which completed the task in 170 sec. This procedure helped us in testing and evaluating the application to get the best-optimized state.

For autoscaling, our initial logic was to scale out app-tier instances as we receive the request in our request queue. We tested this logic, which was working fine, however, we further wanted to optimize on time. So instead of linearly increasing the app-tier instance (out-scaling) our team went ahead and followed the bound range approach i.e. for every 5 requests in the request queue, we utilized one instance. This significantly improved the overall performance. The performance of auto-scaling was benchmarked with the wait time to 1 min. For 10 requests it takes 66 sec and 100 requests, 170 secs.

Code

WEB TIER:

app.py	It contains the code for the Flask application with the endpoints for health check and image processing. It also has a route for the front end landing page of the application.
requirements.txt	It contains the packages required for running the web tier.
templates/	It is a folder containing HTML templates for the web tier instance.
auto-scaler/ec2_manager.py	It contains functions to create, stop, terminate and perform operations on EC2 instances.
auto-scaler/controller.py	It contains the logic of scaling out the app-tier.

Table 3.

To run the web tier code on an instance,

Install pip:- **sudo apt-get install python3-pip**

Install requirements: **pip3 install -r requirements.txt**

To run the flask application server, **flask run –host=0.0.0.0 –port=8080**

To run the auto-scaling script,

python3 auto-scaler/controller.py

APP TIER:

credentials.py	Contains constants such as AWS SQS queue names and S3 bucket names
data.pt	Required for Image processing
face_recogonition.py	Required for Image processing
facenet_pytorch	Required for Image processing
main.py	Contains the code to get the latest image from the queue, process it, store the input and result in S3 and push the result to the response queue. It also has the code to shut down the instance when the queue length is 0 for more than 1 minute.

requirements.txt	Contains the packages required for running the app tier.
wrapper_s3.py	Contains wrapper code with functions related to S3 like upload file, upload value, etc.
wrapper_sqs.py	Contains the wrapper code with functions related to SQS like get_queue_length, delete message from the queue, get the latest message, etc.

To run the app tier,

Install requirements: **pip3 install -r requirements.txt**

To run the python application, **python3 main.py**

Configuring cron to run main.py on startup of an instance,

-> crontab -e

-> @reboot /usr/bin/python3 /home/ec2-user/main.py

4. Individual contributions

Arindam Jain (ASU ID: 1223458106)

I. Design:

My responsibility for this project was to work on auto-scaling logic and controller files located in the web tier of the application and ensure the task is performed within the given time. I wrote code to perform auto-upscaling of app-tier instances using the length of the request queue, spin up ec2 instances with custom AMI containing app-tier code. The number of instances for scaling out as per number of messages is defined as:

Messages in Request Queue	Number of app-tier instances	Messages in Request Queue	Number of app-tier instances
1-5	1	26-30	6
6-10	2	31-35	7
11-15	3	36-40	8
16-20	4	41-45	9
21-25	5	46-50	10

For 50+ messages in the request queue, we start 19 app-tier instances and when there is no message in the request queue, the app instance shuts down.

II. Implementation:

My main responsibility for this group project was setting up the autoscaling logic written in the controller.py in the web-tier that would be ultimately used by the application for scaling. Our initial idea was to scale up the app tier instances as the request queue length increases, however it turned out to be time-consuming. So, we proceeded with a defined range auto-scaling approach and when the request queue length became zero, the app tier code shut down the instances within 60 sec. Now coming to the code implementation, we check the request queue length at an interval of 5 sec and calculate the number of running instances (running + pending state instances) and calculate the stopped instance (if any). The reason behind calculating the stopped instances is to optimize time and reuse an already created instance that is in a stopped state instead of creating a new one from scratch.

III. Testing:

Initially, I tested autoscaling by increasing messages in the request queue and spawning instances on the go, upto 19 instances. As requests decrease, the instances stop themselves. But this was quite time-consuming and not a good approach as starting and stopping an instance takes significant time. Later, I changed the logic of stopping by adding a wait time of 2 min i.e. even if there is no request in the queue, the app-tier instance will not shut down immediately but after 2 min. Further, instead of scaling out as the number of requests increases, I switched to a range system i.e. with every 20 messages in the queue, it will spawn one instance. This significantly improved the time consumption. Finally, the performance of auto scaling was benchmarked with the wait time changed to 1 min and for every 5 requests there will be one instance spawned upto 19 instances, these results were obtained by multiple testing with different numbers of requests. With all these modifications our team was able to run 100 concurrent requests within 5 mins

Panemangalore Srikanth Kini (ASU ID: 1222310674)

I. Design:

I worked on building the web-tier part of our project. As I had prior experience with Flask, I built the front-end using Python and Flask with endpoints to upload files to the application and wait for the response after the application finished processing. To process the image, I used the boto3 library to push messages into the request queue and used uuid to create a unique identifier for each message. After pushing the image to the request queue, the code runs a loop to repeatedly poll the response queue and identify the corresponding message, delete the message from the queue and return the classification as a string.

II. Implementation:

The flask application includes end-points to the HTML index page which contains a nice form to upload an image and the HTML results page to get the result in a tabular format. An end-point to process an uploaded file is also included, that converts the file into a base64 encoded string so that it can be put in the request queue. Each input file also has a unique identifier associated with it to easily identify the classification result from the response queue. Once the web tier places the input image onto the request queue, it keeps polling the SQS response queue, reading 10 messages at once to check if the classification result for the input image is obtained. This is done by checking the unique identifier associated with each classification in the output queue with the identifier of the input image. Once there is a match, the message is deleted from the output queue and the result is sent back to the user. If there is no match, the code polls the response queue for 10 more messages.

III. Testing:

I tested the front-end code by hosting the flask server on my local machine as well as the web tier EC2 instance and repeatedly making requests to the different endpoints. Each of the endpoints were tested in my local machine and fixed for any existing bugs. Later, the code was pushed to an EC2 instance and the flask application was run from the instance. The endpoints were accessed via the instance's public IP and tested for handling concurrent requests. After checking the correctness of the web tier, it was integrated with the app tier and the auto scaling code to test the end to end working of the application extensively. The results were benchmarked and are included in the Testing and Evaluation section of the report.

Bhavani Mahalakshmi Gowri Sankar (ASU ID: 1222186719)

I. Design:

The app tier consists of models to read messages from the SQS queue, store the image file in s3 input bucket, perform the classification and write results into the output SQS queue as well as into the s3 output bucket. The app tier code also had to be converted into an AMI image that is used by the auto-scaling code to spin up new app tier instances when there is a need to scale up owing to a large number of requests.

II. Implementation:

The app tier code was implemented in Python3 using the boto3 package in order to connect to, and use the AWS services (S3, SQS). Each AWS service has its own model where all its functions are implemented. The main module(main.py) checks the input SQS queue, specified by its URL, reads the latest message from the queue (if any) and starts processing the message. The message is decoded and converted back into a JPEG file. The image is put in the S3 input bucket. The image is then run through the classification algorithm provided and the results obtained are put in the output SQS queue and the output S3 bucket. When there are no new messages in the input SQS queue, the app tier waits for a minute to see if it receives any new messages in the queue. If it doesn't, it shuts down, moving the instance to Stopped state.

The second phase of App tier implementation involved creating an AMI image from the app tier code. The app tier code, along with the classification model provided by the Professor, was copied to an EC2 instance. The required packages for running the app tier and the classification model were installed on the instance. A cron job was added to run the app tier script on every reboot to make sure the app tier code starts running automatically in the background when a new instance is created or a stopped instance is restarted. Finally, the AMI ID was used by the auto-scaling code to spawn new app tier instances based on the number of requests. I also created an AWS Role that gives full access to EC2, S3 and SQS. The role is given to both web and app tier instances.

III. Testing:

The models for S3 and SQS were tested individually before integrating with the rest of the app-tier logic. Once the entire app-tier logic was completed, it was tested extensively on my local machine. Later, it was put in an ec2 instance and tested with sqs, s3 and a web tier ec2 instance. The performance of a single app tier instance was benchmarked with the architecture for 100 images. After thoroughly testing the app tier instance and configuring the cron to make the logic run automatically when the instance is rebooted, an AMI was created from the instance. The AMI ID was used in the auto-scaling logic and verified to have the same contents of the original app tier along with the cron configuration.