CSE 546 — Project1 Report

Ashish Kumar Rambhatla *1215350552*
Chaitanya Prakash Poluri *1223158937*
Reethu C Vattikunta *1222619619*

1. Problem statement

Our project's main motto is to build an Elastic cloud application where we are providing the services for Image recognition by using IaaS (Infrastructure as a Service) resources provided by Amazon Web Services (AWS). The Application's functionality is to take images as input from the users, use a deep learning model to classify the images, and display the results on the user interface(or website). The main advantage of the Application is that we can handle multiple user requests concurrently. Another advantage is that when the client demand increases, the Application automatically scales out, and when the demand decreases, it automatically scales in. This autoscaling will utilize the limited resources cost-efficiently and quickly. The following AWS services were used to create this Application: Elastic Compute Cloud EC2, Simple Queue Service SQS, and Simple Storage Service S3.

2. Design and implementation

2.1 Architecture:

Commented [1]: Provide an architecture diagram including all the major components, and explain the design and implementation of each component in detail.

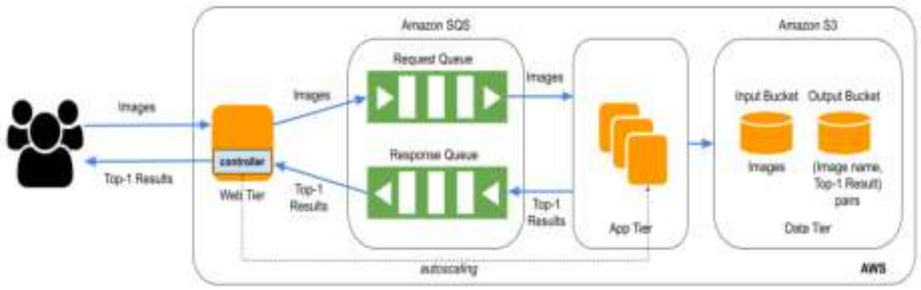


Figure 1: Image Recognition Service Architecture

The Application Architecture mainly consists of three tiers:

- Web-Tier (controller is inside Web-Tier)
- App-Tier
- Data-Tier

Web Tier:

The Application takes images provided by the users as input. That step will redirect to the web tier from the HTML interface. When the app starts running, request SQS queue and response SQS queue are created with distinct names. The obtained images are encoded into a string using the base64 method in the Web Tier. The image file name and encoded string are stored as key-value pairs. Web Tier is implemented by utilizing AWS EC2(Elastic Cloud Compute) instances. A web interface for users to upload images is made using the Python flask application. A web tier is designed so that users may upload an endless amount of pictures to the app. Each picture that the user uploads is treated as a request to the web-tier instance. As the images are uploaded, each image is encoded as a dictionary with key as image name and value. These encoded key-value pairs are sent to the App Tier through an AWS service called Simple Queue Service (SQS).

App Tier:

The App tier receives the encoded key-value pairs from the Web Tier through the AWS SQS service. To efficiently utilize the AWS free tier resources, the AWS services are split between two different AWS accounts and the access to the respective resources are assigned by assuming specific roles that provide temporary credentials. To accomplish this, an amazon service called Security token service (STS) is integrated into the App Tier.

The App Tier uses a ThreadPool mechanism to continuously receive the messages broadcast by the SQS service and assign them to the respective worker threads. The worker threads then run the operations to convert the incoming key-value JSON messages back into a dictionary The key-value pairs are extracted from the dictionary and the encoded image is decoded to retrieve the byte stream of the image file. The retrieved image file names and images are used as key-value pairs to store them in the Data tier in a AWS Simple Storage Service (S3) input bucket.

Post saving the incoming data in data tier, the worker threads will spawn a subprocess to run the image classification task and wait for the results. The output of the image classification script is a string representing the name of the person present in the image file. An output dictionary is prepared with its key as the image filename and the output as the classification result. This dictionary is used to store the results in Data Tier’s output S3 bucket. Post the data tier operations, the dictionary is converted into a json format to put it back into the SQS response queue.

The App Tier implements the scale-down logic to efficiently utilize the AWS resources. The following section discusses in detail the implementation details of the auto-scaling logic.

Data Tier:

The Data tier consists of two AWS S3 buckets to persistently store the input and output information of the App Tier classification task.

The input bucket stores the incoming image files from the web tier as key - value pairs files. The key represents the file name of the image while the value is the actual image file in its original compression format.

The input bucket stores the results of the image classification task as key - value pairs files. The key represents the file name of the image, without the file extension, while the value is the classification result, say Paul, stored in a plain text format.

Output:

The output is stored in S3 bucket with the key as the image name given(like test_25) and the value as the classification result obtained(like Paul). Thus when the user clicks on the "Result" button on the web interface, the output will appear.

2.2 Autoscaling

Commented [2]: Explain in detail how your design and implementation support automatic scale-up and -down on demand.

The Auto Scaling functionality is crucial to the elastic behavior of the application to effectively use the AWS resources. Auto scaling involves scaling up and down the resources based on the user demand. While the AWS provides the auto-scaling service to automatically do the scale up and scale down, this application implements this functionality by delegating the scale down functionality to app tier and the scale up functionality to the controller component.

Scale Up

Scale up means increasing the number of app tier instances if the demand from the users increases. The controller component hosted on the web tier is monitoring the number of messages in the AWS request SQS queue. The load per app tier instance is set to a threshold after multiple trial runs to maximize the performance. If the number of requests in the queue exceeds the load that can be handled by the number of running app tier instances, the controller calculates the required number of instances to balance the additional load and starts them concurrently using boto3. ec2 service.

Scale Down

Scale down means decreasing the number of app tier instances if the demand from the users decreases. The app tier implements a heartbeat counter that runs in parallel with the classification tasks. The heartbeat counter keeps track of the number of minutes the app tier is idle without processing any messages from the input SQS queue. A threshold is set to 3 minutes of idle time per instance from our trial runs to minimize the processing latency taking the instance up/down times into account. If the heartbeat counter reaches the pre-defined threshold, the app tier shuts down itself by fetching its private dns-name and using it as filter for the boto3.ec2 service API to stop the instances.

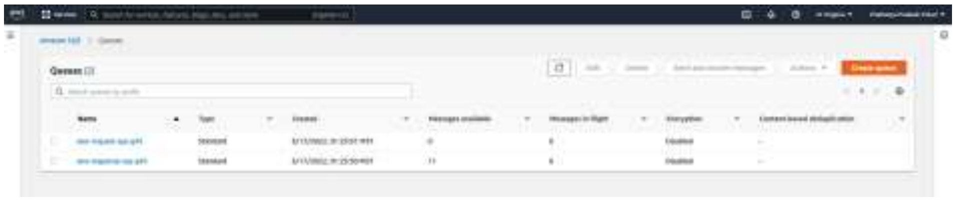
- **Testing and evaluation**

Commented [3]: Explain in detail how you tested and evaluated your application.
Discuss in detail the testing and evaluation results.

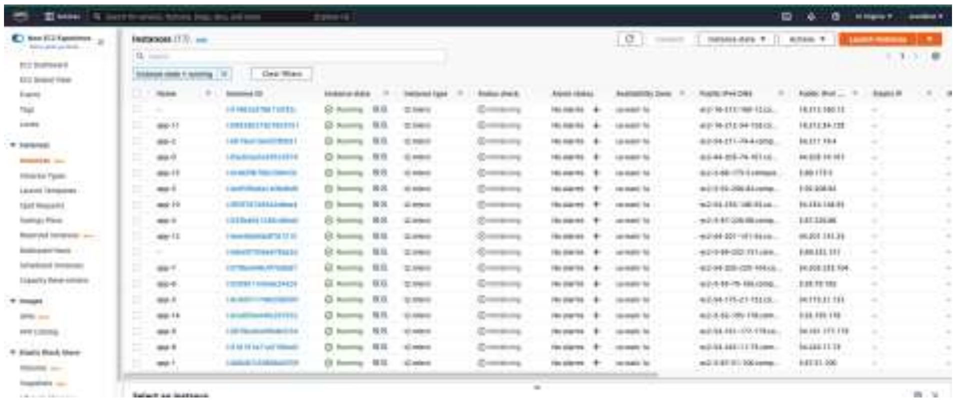
We are testing the app by uploading photographs from the web interface. The image classification is evaluated by comparing the outputs to the original results given, which are obtained in a reasonable amount of time.

The following screenshots show the different aws resources and their respective state machines in the respective stages of the application execution and user demand.

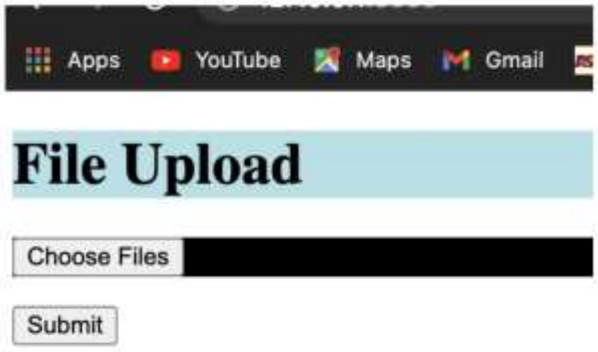
The following screenshot show the the input and output SQS queues :



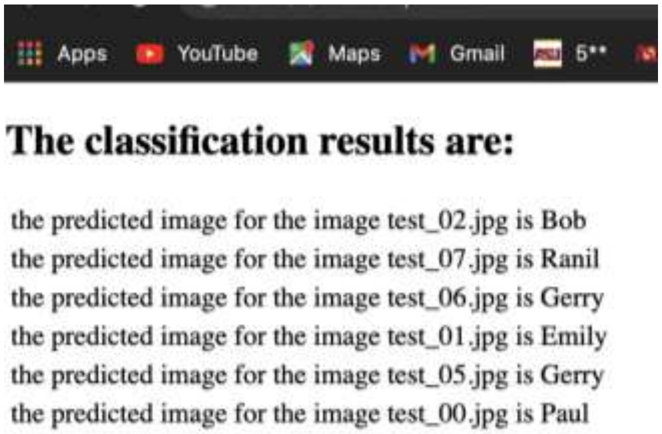
The following screenshot shows that the scale-out is happening when demand is high but doesn't exceed 19 instances:



The image below shows our web interface to upload the image



The screenshot below shows the image classification results displayed on the website.



The classification results are:

the predicted image for the image test_02.jpg is Bob
the predicted image for the image test_07.jpg is Ranil
the predicted image for the image test_06.jpg is Gerry
the predicted image for the image test_01.jpg is Emily
the predicted image for the image test_05.jpg is Gerry
the predicted image for the image test_00.jpg is Paul

Results are obtained as follows :

Bill	//is shown when when user uploaded test_07.jpg and clicked on submit button
Gerry	//is shown when when user uploaded test_26.jpg and clicked on submit button

3. Code

Commented [4]: 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.

Our project have the following 4 files:

1. WebTier.py
2. AppTier.py
3. Controller.py
4. Index.html

WebTier.py:

We created an upload button for uploading the image on the index.html page using the post request. We hosted the website using flask server. Here we used three functions: one to create a new sqs queue, next to upload an image to the sql queue, and next one to get the result of the image uploaded. Here we are taking the file uploaded and decoding the image to UTF-8 unicode string and passing it to the created aws queue along with the image name. Then we are redirecting the page to the result page where it can access the response queue, it has the classification result passed by the app tier. When the submit button is clicked, it redirects to the results page showing the name of the person enclosed in the image, which is processed in the app tier.

AppTier.py:

The AppTier.py hosts the code to upload the data to the S3 input buckets, process the images using the provided **"**face_recognition.py" file and send the responses back to the response SQS queue. The AppTier implements a *threadPoolExecutor* for parallel consumption of the incoming messages. The following are the different functions to implement different aspects of the App Tier.

getS3Bucket(bucketParams): To get the buckets with the provided name. It also has the logic to create these buckets on the first app instance launch.

getSqsQueue(queueParams): To get the sqs queue objects with the specified name. This function also implements the logic to get the temporary security credentials to access the SQS queue owned by a different AWS account by assuming the pre-defined roles. It calls the *AssumeRole* API to elevate its permissions for fetching the specified sps queue object.

runImageclassification(message): This function is used as a thread entry function by the *threadPoolExecutor* to map the received messages from the request queue and run them in parallel. The message object is parsed using *json.loads()* API and stored in a dictionary. Then it fetches the input bucket and stores the received input image. It also makes a temporary local copy of the image to perform the classification task. It then spawns a *subprocess* to run the classification task by providing the locally stored image file as the input. The results of the classification are stored in the output bucket as well as returned to the caller of this function in a dictionary format.

processSqsMessages(): This function fetches the messages from the input queue and maps them to the worker threads using *ThreadPoolExecutor.* It then fetches the results and puts them back into the response queue.

main(): This function runs the *processSqsMessages()* function. It also implements the heartbeat counting logic for scaling down.

cloudAppTier.service: This systemd service file is used to run the appTier.py file automatically on bootup of the app Tier instance.

Controller.py:

within the while loop, the controller runs infinitely checking the length of SQS Queue. It also has the number of running and stopped instances. There are two distinct lists that are appended with running and stopped instances IDs respectively. As we only have 20 free tier instances, we assign the threshold to be 19 such that the maximum number of instances to run are set to be 19 (as the web-tier instance is always on). If the length of the queue is greater than the length of running instances, we start new instances according to the length of the queue. As mentioned, we stopped the instances based on the Queue length in the App Tier.

Index.html:

The HTML website is rendered when we run the Application using Python Flask. When we click on the “Upload” button(uploading image), the POST request will redirect to the WebTier file. When clicked on the “Submit” button, the GET method will redirect to the answers page that contains key-value pairs and displays the value as classification result(output).

Steps to install and run the programs:

➔ Install FLASK app, PyCharm IDE and BOTO3 Libraries
➔ Create AWS account
◆ Create two S3 buckets for storing input images and output key-value pairs namely “g45-input-bucket”, “g45-output-bucket” respectively
◆ Create two SQS queues which will store the input requests and output responses namely “aws-request-sqs-g45”, “aws-response-sqs-g45” respectively
➔ Set the FLASK_APP variable and execute flask run to start the web tier
➔ Once the application is run, user can upload the images on our HTML page
➔ As the process of image classification is done, the results can be displayed by clicking on the “Submit” button on the HTML page

Individual contributions (optional):
Reethu Chowdary Vattikunta (1222619619)
The goal of the project is to develop a cloud application using AWS IaaS Services(EC2, SQS, S3). The application auto scales according to the user traffic and thus works cost-effectively by using cloud resources (AWS). AWS is the most popular IaaS provider, with a wide range of computation, storage, and messaging cloud services. Our program takes users' photographs as input, predicts the deep-learning model's output, and shows the findings(image classification result).

- Design :

All of my teammates collaborated while designing the Architecture of our Application. I wholly contributed to designing and developing the logic for the controller. Developed the auto scale out logic within the controller file, whereas scale in logic is integrated in App-Tier according to the same SQS activity. In the controller file, I created EC2 instances according to the length of the input SQS Queue. This gives us information on how many app-tier instances to start as per length of the queue. I also help my teammates in hosting the website through DNS and also with putting input and getting output results from two separate S3 buckets.

- Implementation:

As the length of SQS Queue depends on the input given by users. I obtained the length of input SQS queue length using boto3. Controller code periodically checks the sqs length and starts a new ec2 instance based on the demand. The scale out logic provides the number of ec2 instances to start. I also made sure that the number of instances to create or start should not exceed 19, as we only have 20 limited resources. While processing each file, I checked to see whether the queue length was larger than the number of instances running; if it was, I added additional instances as required; if the number of instances was greater than the queue length, I reduced the number of instances that would be running. When one of the instances isn't connecting, I terminate it and try to connect to another. I halted the instances after they finished processing and restarted them if queue length was larger than the number of current instances.

- Testing:

I conducted a lot of manual testing with different input sizes such as 10,20,50,100 to examine how the autoscaling performs. The scale out was appropriate according to the logic that I have provided. I also ensured that the number of EC2 instances did not surpass the threshold limit of 19 instances at any moment. The scale in was also done as per the inactivity of input SQS queue. The accurate results are received after installing the deep learning model with expected value outputs. I also double-checked that the output SQS and output S3 bucket had the same values.