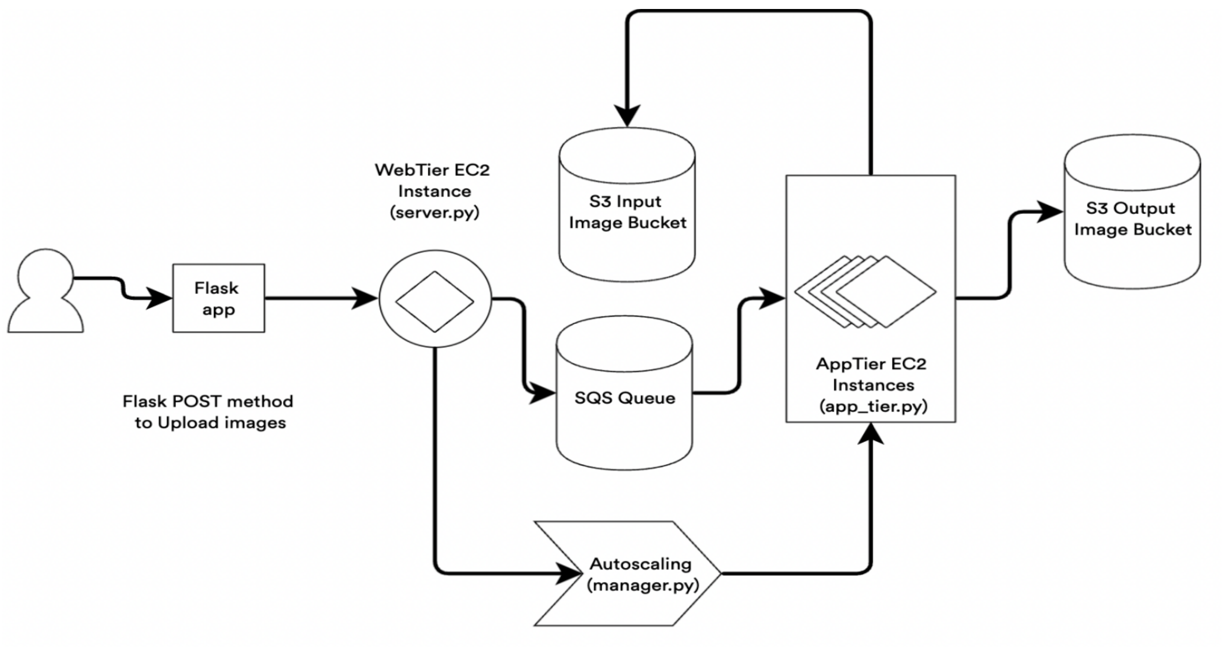
CSE 546 - Project 1 Report

Adwait Patil
Dhaval Shah
Akul Rishi

1. **Problem Statement**:
   The problem at hand has to be approached by building an IaaS based image recognition service on AWS EC2. The provided AMI consists of a pre-loaded model to perform image classification. The EC2 application should have auto scaling feature depending upon the number of requests in the input queue, this helps achieve rapid elasticity. In this case, the image classification is achieved using the AMI that we launch on an EC2 instance. Multiple test cases are provided to run the benchmarks on, where they run as a workload using the provided workload generator. The idea behind using this workload generator is to create a request sequence pool that is fed to the input queue in SQS. Depending upon the number of requests in the input queue, the IaaS application should automatically scale up and down. The provided input images and output results are stored in separate S3 buckets. At the end, the output queue shall return the output of the machine learning model.

2. Design and Implementation

a. Architecture


WebTier EC2 Instance (server.py)
S3 Input Image Bucket
S3 Output Image Bucket
Flask app
Flask POST method to Upload images
SQS Queue
AppTier EC2 Instances (app_tier.py)
Autoscaling (manager.py)

b. Auto-Scaling:

In order to implement the auto-scaling feature, we implemented an on-demand instance provisioning logic that will use the number of messages present in the input queue. When the number of messages in the input queue crosses a set threshold, a new instance is added and starts working on the load. The threshold is determined using a variable ‘scaling parameter’ that uses intervals of 5 messages to scale out; if this value is exceeded, a new instance is added.

Example: If there are 8 messages present in the input queue, 2 instances shall be created (they can handle up to 10 messages). For 16 messages in the queue, 4 instances are created which can handle up to 20 messages.

When the total number of messages decreases in the multiples of 5, then the instances are dropped accordingly. When the input queue becomes empty, the auto-scaling manager stops all the running app-tier instances. When an app-tier instance stops itself, it ensures that the auto-scaling manager does not shut any instance while an image is still being processed.
Hence, the total number of instances that the manager starts is MAX(scaling_requirement - running_count, max_instance-running_count) {It ensures that the number of instances started do not exceed the number of instances present in the app-tier}

c. Member Tasks

Adwait Patil

1. Implemented the web-tier of the project. [Virtual environment for Python and Flask application]
2. Used the Gunicorn server for hosting the Flask application as the web-tier through which the client can send requests.
3. Created helper files pertaining to the SQS which contained functions that sent the incoming requests to the input queue and pulled the responses from the output queue.
4. Worked on integrating our application with EC2 deployment and instances on AWS. The code needed to be refactored in order to be placed to the cloud once the project had been tested locally.
5. Ensured that the code functioned as intended and that no extra, useless methods had been developed in the past to test functionality or try out alternative ideas.
6. Implemented SQS utility functions. [encoded image strings into the input queue, getting the length of the input and output queues]
7. Implemented a function that uploads the output returned from the image recognition model to the output bucket.

Dhaval Shah

1. Created a ‘systemd’ service, which started as soon as the App tier an instance was started.This service called “startup” executed a shell script called “startup.sh”, which in turn started the App tier part of the application.
2. Solved the request timeout issue by configuring the timeout value to 300 seconds in the Gunicorn server. [Sufficient for web-tier to receive outputs from the output queue]
3. Performed testing on the application by varying the timeout period [seconds] of the web-tier and optimized the timeout period to get the response for its respective request.
4. Worked on putting the project report together, making sure that every requirement of the report template was met.
5. Implemented sessions for each of the resources SQS and S3 that are accessed by many threads.
6. Implemented functions to upload the image files to the S3 buckets in the form of binary objects.

Akul Rishi

1. Created all the resources on AWS specified in the given architecture of the project including the app-tier and web-tier instances and their respective keys, SQS input and output queues, and S3 input and output buckets.
2. To make the processing faster, used multithreading by configuring the Gunicorn server to take multiple requests [this helps ensure processing of simultaneous requests provided by the client]
3. Varied the thread count to determine the right number of threads to optimize the application's overall operating time and handle the entire workload. This caused the processing time for the entire application to decrease and fall under the necessary time frame.
4. Designed the architecture diagram.
5. Implemented EC2 utility functions that help in determining the exact state of instances. (states, number of instances running and stopped etc.)
6. Contributed to the system's documentation while following the outlined specifications and formatting rules.

3. Testing and evaluation:

a. First, we used Postman to send a single picture as part of an HTTP request in order to test the web-tier endpoint. This was done to see if the output for the testing images was what we wanted. We correctly identified each response.
b. The URL was then tested with 10, 20, 30, and ultimately 100 concurrent queries using a single app-tier instance. This was accomplished without using the mechanism for automatic scaling. 100 queries might be correctly answered by a single instance in roughly 300 seconds.

c. Finally, we put the auto-scaling logic into place, which starts the right number of instances based on how many images are in the input queue. The auto-scaling was discussed in the section before this one. During this testing phase, we were able to receive accurate answers to the 100 concurrent requests in just under a minute.

4. Code:

The functionality of each program is explained in the zipped file provided for this project as follows:

a. **EC2.py:**
The purpose of this utility software is to retrieve EC2 instance-related parameters. These parameters can be obtained via a number of methods, like get running count, get instances, get instance state, etc.
get_running_count- this method returns the number of instances running on the console which have their status set to either running or pending.
● get_instances- this method returns a list of instances on the console.
● get_instance_state- this method uses the instances on the console, and then returns its state.
● start_instances- this method takes the number of instances to be started, and comes in handy when the task at hand is to start new instances for auto-scaling the model. From the list of instances that are already created, the state of each instance is noted. For the ones that are marked 'stopped', those instances are put into a list of instances to be started and are then set to run until all the instances are running.
● stop_all_instances- this method uses the list of instances and checks for the instance that is running, adds them to a list of instances to be stopped, and stops them. If the number of instances running is 0, it simply returns a message conveying the same.

b. **S3.py:**
This utility software manages the input and output functionality for our two S3 buckets. Here are a few strategies for managing uploads and downloads to and from the input and output buckets.
upload_file- this method makes use of the provisioned S3 buckets, specifically, the input bucket. The method takes two arguments in its parameters, the file, and its filename. Based on this, it tries to upload this file to the input bucket, and if any error occurs, it will say so.
● download_file- this method also uses the input S3 bucket, and uses a filename as its argument. Given that, it tries to download its file object and if any error occurs, it will say so.
● put_result- this method uses the output bucket and stores the output generated by the app tier. It takes the filename and the generated output as its arguments, and puts that result in the output bucket. The file name is the key.

c. **SQS.py:**
This utility software manages and displays significant queue parameters for the input queue and output queue, two queues that have been provisioned on SQS.Here are a few crucial strategies that facilitate effective communication between the web and app tiers.
get_queue_length- this method accesses the input queue and returns its length
● get_out_queue_length- this method accesses the output queue and returns its length
● enqueue- this method has a message as its argument, and it works by sending this message to the input queue and returning its message-id. This helps to enqueue messages in our input queue for the app tier to work on.
● enqueue_out- this method has a message as its argument, and it works by sending this message to the output queue and returning its message-id. This helps to enqueue messages as output from the app tier that can be later returned to the user via web tier.
● dequeue- this method checks if the queue is empty, if so, there's nothing to deque. Else, from the input queue, messages are scanned and each of the messages is returned as output, and then it is deleted from the input queue.
● get_result_to_send- this method is created to put the output in the output queue. The output queue is scanned, if not already empty, it will scan the messages in the output queue and for each message, it will return the filenames.
● get_messages- this method is used to maintain a hashed output queue on the web tier. The function accesses the input queue, iterates through the messages, and stores the key-value pairs such that key is message-id and value is its message body. Once stored in the dictionary, the message is deleted from the queue.

d. Server.py:
This application's web tier makes use of the flask framework. We've developed a way for you to upload files here. To send a file upload to our server, we employ the post mechanism. Go to the directory that contains this file i.e. web_tier in the source code. Run it by setting up a flask app using this file and execute the command gunicorn —bind 0.0.0.0:8000 server:app—timeout 180 —threads 30. Now, the end-point listens for requests.

e. App_tier.py:
This application's app tier makes advantage of the AMI that comes with the facial recognition machine learning model. The command that executes the image recognition model is run by the app tier using a subprocess. The output queue receives cleaned output from stdout.

f. **Manager.py:**
The logic for auto-scaling is managed by this file. It keeps an eye on the input queue to count the images being input. The upscaling and downscaling functions are then called in accordance with the auto scaling mechanism outlined in the preceding section. Every 30 seconds, it continues to poll the input queue to determine how long it is. The application is then scaled as necessary, either up or down.

To launch the manager, type python3 management.py into your terminal.

Run the files D and F together on the web-tier instance, create the desired number of web tier instances with the app-tier code on it along with a startup script to run the file E on startup of the instance and then use the workload generator to send the requests to the URL :

http://44.206.167.58:8000/upload

The files D and F run on the web tier. It has to be run as python files on the instance and the file E is run automatically as a service when an app-tier instance starts up. All other files are imported as utility modules to these three files which support the access to the AWS resources.