CSE 546 - Project 1 Report (Group 18)

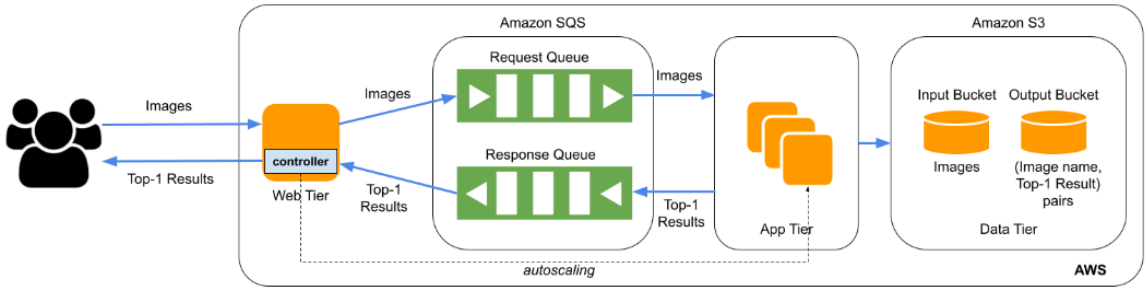
Simran Kharpude(1223158989)
Dharmit Prajapati(1219597132)
Pranav Agrawal(1222300690)

1. Problem statement

We have been given an AMI(Amazon Machine Image) that contains an image recognition model which classifies the input image. If there are multiple images to be processed it would be done serially, thus taking a lot of time. To solve this problem of high latency, we are using AWS resources so that we can scale up and down depending on the number of images/requests. Multiple images can then be processed parallely thus providing a better response time.

2. Design and implementation

2.1 Architecture



We extensively used 3 Amazon services: Elastic Compute Cloud (EC2), Simple Queue Service (SQS), Simple Storage Service (S3).

There are 4 components:

1. Web Tier (EC2)
2. Response and Request Queues (Amazon SQS)
3. Application Tier (EC2 and AMI)
4. Input and Output bucket (Amazon S3)

Web Tier:

The Web Tier had the following components:

a) A Flask application (appTier.py) that acted as the host server and received images from the user. For each image, a unique ID is generated. Image along with its ID is pushed to the

Request queue as a message. Once the message is pushed into the queue this code polls a folder for a text file with the Unique ID as a part of its name.

b) A Response Queue Listener which will listen to the messages from the response and generates a text file classifying the image and the unique ID in its name

c) A Scaling logic that keeps track of the request queue size and scales out or scales in by creating and terminating instances.

Application Tier:

The application tier contains the AMI that has the image recognition model. Along with the model, it has an image classification and listener file that is instantiated when any instance with the AMI is created. We passed the necessary Ubuntu commands while creating the instance with the AMI to trigger the image recognition file that classifies the images. The classified label along with its unique ID is pushed to the Response queue and the image and the classification result is put in the input and output S3 buckets respectively. The number of instances generated of the App Tier is based on the number of requests received from the Web Tier

Response and request Queues:

These queues are used as buffers to accommodate the delay of messages caused by the processing time of the image recognition logic. Usage of the queues helps us to decouple the web and the application tier thus providing the ability to scale.

Input and Output Bucket:

This utilizes S3 (Simple Storage Service) which is highly durable. The Input bucket stores the images sent by the user through the requests and the Output bucket stores the classified label of the corresponding image.

2.2 Autoscaling

In this project our aim was to use one of the most important characteristics of Cloud Computing ie. Autoscaling. As per the design and architecture mentioned above, we need to create and terminate instances based on the input image/requests. So if the number of requests increases, we increase the number of instances and decrease the number of instances when the number of requests decreases.

1) First step is getting the number of requests that are present in the input queue. This is done by using the parameters ‘ApproximateNumberOfMessages’,‘ApproximateNumberOfMessagesNotVisible’,‘ApproximateNumberOfMessagesDelayed’. The sum of the two parameters will give us the size of the queue.

2) Next step is to retrieve the number of instances that are in running or pending state. We can use the Instances Filters command, we can filter the instances using the Application Tier name and providing the states: running and pending. This will give us the count of instances(number of instances in pending or running state), which are running or about to run

3) The current variable contains the number of instances that are currently running. This will also help us in determining creation and termination of instances, as number of instances should not exceed 19 as we are only permitted to use 20 instances.

4) Scaling Out:

From step 2 we already have the number of instances in running/pending state. To estimate the number of instances needed to process the queue we used the ceiling value of (queue_size/2). If the number of instances needed is greater than the number of instances in running/pending state, it means we scale out i.e. create more instances. Also at the same time we have to keep a track the number of instances are not more than 19 , which can be achieved the using the current value in step 3

The current variable is then increased by 1.

5) Scaling In:

To estimate the number of instances needed to process the queue we used the ceiling value of (queue_size/10).

In order to ensure that the termination of instances only occurs when the required number of instances remains constant for an extended period of time, we additionally utilize a counter. To prevent the deletion from occurring unintentionally, this is done. If the queue fluctuates, instances might be generated right away, but by the time they're produced, the queue size will have shrunk and the deletion process will have begun.

If the number of instances of needed is less than the number of instances in running/pending state, it means we scale in ie terminate the instances as the demand has reduced. The no of instances to be terminated is calculated as the absolute difference between the number of instances needed to process the queue and number of instances in running/pending state.

The current variable is then decreased by 1.

For both scaling in and out operations, we have used the boto3 python library.

2.3 Member Tasks

Dharmit:

1. Design:-

   Before diving deep into the project, I wanted to understand how different components of AWS work. So I understood the working of EC2 instances and how they can communicate with SQS and Buckets. In the design stage, I came up with the design strategy on how applications are going to be scaled up and down. This was a big challenge as the timing of the creation and termination of instances was crucial to make sure they don't shut down without the completion of the recognition task. We used the tracker logic to keep a track of the message count.

2. Development:-

   My first task was to write the scaling logic for the instances. I needed to first figure out how to find out the running instances. I had to poll the queue to retrieve the messages and use a counter to appropriately name the instances. For the shutdown/termination logic I had to use the tracker array to keep a count. Creation and termination of instances was carried out with the help of boto3 library. My next task included pushing the objects to the S3 buckets. Finally, to trigger the recognition task for the images in the instances I had to write the bin bash code in the user data to make sure the images are classified for each instance.

3. Testing:-

   As a part of testing, I performed unit tests on my scaling logic to make sure that the instances are created and destroyed in an expected manner, images are pushed to the input bucket, the recognition model gets triggered for each instance created. I also ensured that the objects created in the S3 buckets had the correct classification label .

Pranav:

1. Design:-

   I, along with my teammates started with creating the AWS resources to establish the architecture mentioned in the project. For this, we had to go through the documentation of AWS SQS, EC2 and S3 to understand the different types of settings and what would be optimal for our project. I had the idea of creating text files with the unique ID in its name so that our Flask application can fetch the correct file when it waits for the response. We did this in order to ensure that every request returns the correct output corresponding to the image it represents.

2. Development:-

My main task was to develop the code for listening to the output queue (outputQueueListener.py) and generate the response which the flask application could read. The Flask application and Scaling Logic, which were created by my teammates, coexist on the web tier instance as independent processes along with the outSqsListener. I utilized the capabilities of the boto3 library to access the Response queue. Upon going through the boto3 documentation I came across the receive_messages() function which was used to fetch the contents in the response queue.

The file works in such a manner that it continuously picks up messages from the queue but a maximum of 10 messages at a time. Next, for every message, a text file is created having the classification result in it and the Unique ID as its name. This file is then read by the appTier.py file for returning the classification result to the user.

3. Testing:-

   I was involved in testing the file I wrote(outputQueueListener.py) as well as the whole application along with my team. To test my file for basic inputs, I used unit testing. I made sure that every input test image was mapped with the correct classification result and that the parallel processing did not mess up the mappings.

<u>Simran:</u>

1. Design:-

   I started off by understanding the requirements of the project and the architecture diagram. Then, I looked into AWS documentation to understand how to use the resources mentioned in the architecture diagram. I also did the setup for both Python and Java as provided in the Professor's slides. I read both the AWS SDK as well the SDK for Python (Boto3). Since the team had no problem using either of the languages for the project, I suggested that we use Python along with Boto3 as I found a lot of helpful resources and documentation that could guide us through the project.
   I then discussed my understanding of the problem and how we can approach the solution with other members of the team. We then sat down and brainstormed the appropriate approach and usage of AWS resources for the project.

2. Development:-

   I created the key pairs for two EC2 instances- web tier and app tier. I researched a lot on using scp and other approaches like FileZilla to transfer our local code into ec2 instances. We chose to go ahead with FileZilla as the team was already aware of the tool.
   I contributed to the code on the app tier. I looked up how we can use SQS efficiently and wrote code to consume messages from it continuously, process these messages and push them into the output queue. I also worked on the queue listener part and sat with

the team to brainstorm some ways to map the response with the request so that we send the correct output message.
I also helped in merging everyone's code together and reviewed my teammates' code. I helped my teammates debug issues they were facing while also taking their help to debug errors in the code that I wrote.

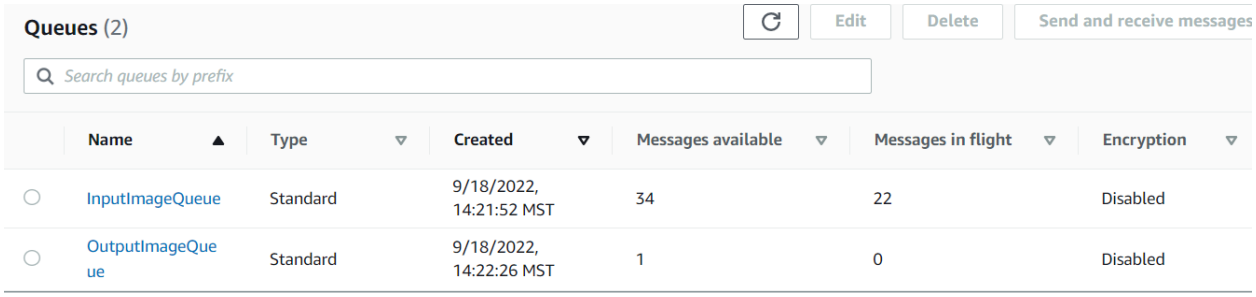
3. Testing:-

   After merging the code, I spent a lot of time testing the functionality of the code I had written and how it was working end-to-end. The workload generator included in the project requirements was helpful while testing several parallel queries. Regression testing was another area on which I concentrated to make sure the incremental code changes were stable and functioning.

3. Testing and evaluation

- We started testing our setup first by seeing if the code files that we transferred to our ec2 instances are running individually.
- Then we tested the output from the classification code provided by the Professor by sending a single image.
- Once these sanity checks were done, we checked if the images were flowing correctly end-to-end without any errors. For this we first sent one image, and checked if it was being consumed by the input queue and then being sent to the output queue and we also checked if the classification result was getting stored in the correct format in the S3 output bucket.
- In order to test the scaling logic, we initially sent 10 requests from the workloadGenerator file which was creating 5 EC2 instances.
- We then tested the complete load by sending 100 requests from the workloadGenerator and monitored the flow. We also checked if the number of running instances was crossing 20 or not.
- Once this check was successful, we verified the prediction results by opening the file objects stored in the S3 bucket.
- We also logged the time taken by our code to complete the entire process to check if our logic was within the time limit of 10 minutes as per the requirement.

Input and output queues getting populated:



Queues (2)

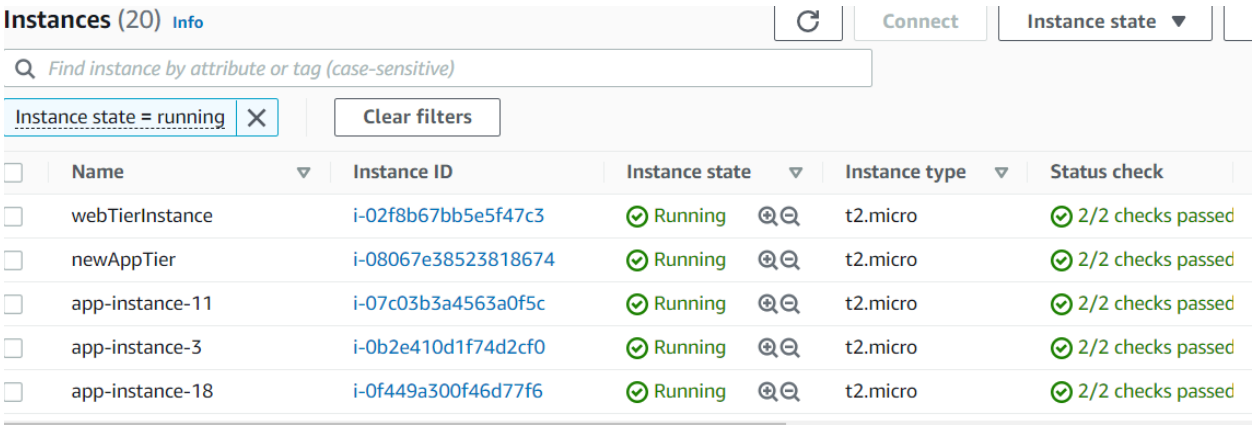
Edit | Delete | Send and receive messages

Search queues by prefix

Name	Type	Created	Messages available	Messages in flight	Encryption
InputImageQueue	Standard	9/18/2022, 14:21:52 MST	34	22	Disabled
OutputImageQueue	Standard	9/18/2022, 14:22:26 MST	1	0	Disabled

Scaling out as per queue size:

```
Creating  app-instance-1
Creating  app-instance-2
Creating  app-instance-3
Creating  app-instance-4
Creating  app-instance-5
Creating  app-instance-6
Creating  app-instance-7
Creating  app-instance-8
Creating  app-instance-9
Creating  app-instance-10
Creating  app-instance-11
Creating  app-instance-12
Creating  app-instance-13
Creating  app-instance-14
Creating  app-instance-15
Creating  app-instance-16
Current queue size : 90
16
Inside create_apptier_instances()
Creating  app-instance-17
Creating  app-instance-18
Current queue size : 84
```



Instances (20) Info

Connect | Instance state

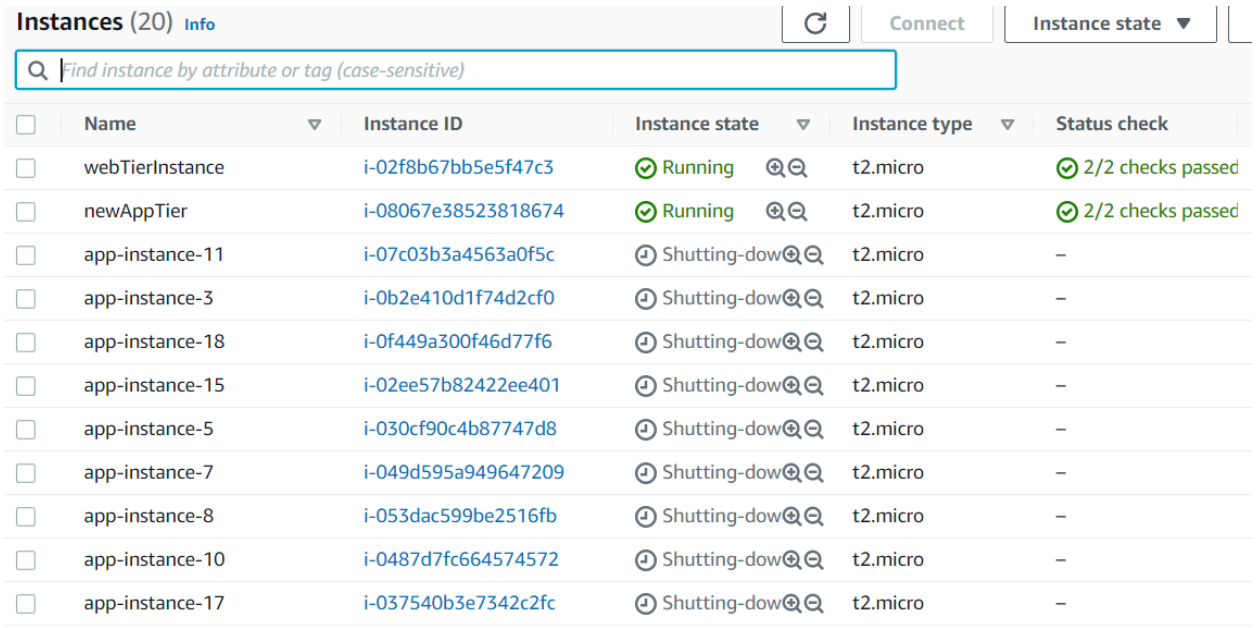
Find instance by attribute or tag (case-sensitive)

Instance state = running | Clear filters

Name	Instance ID	Instance state	Instance type	Status check
webTierInstance	i-02f8b67bb5e5f47c3	Running	t2.micro	2/2 checks passed
newAppTier	i-08067e38523818674	Running	t2.micro	2/2 checks passed
app-instance-11	i-07c03b3a4563a0f5c	Running	t2.micro	2/2 checks passed
app-instance-3	i-0b2e410d1f74d2cf0	Running	t2.micro	2/2 checks passed
app-instance-18	i-0f449a300f46d77f6	Running	t2.micro	2/2 checks passed

Scaling in as per reduced demand:

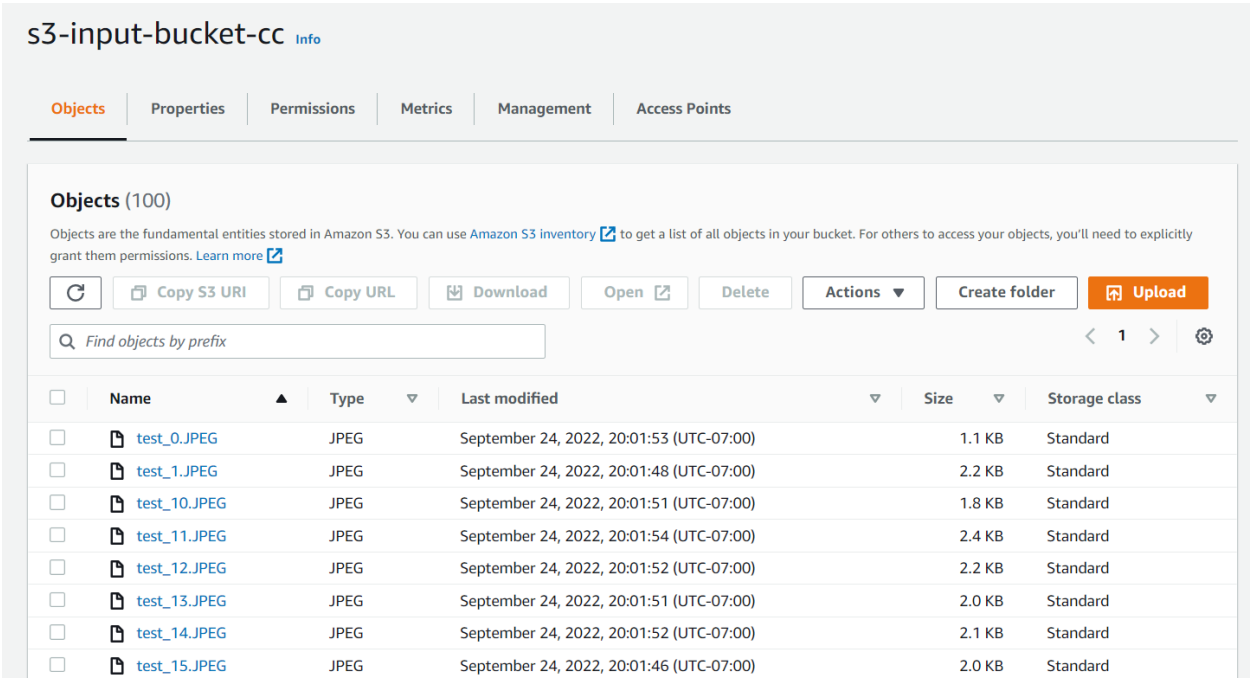
```
Inside terminate_apptier_instances()
Terminated app-instance-18
Terminated app-instance-17
Terminated app-instance-16
Terminated app-instance-15
Terminated app-instance-14
Terminated app-instance-13
Terminated app-instance-12
Terminated app-instance-11
Terminated app-instance-10
Terminated app-instance-9
Terminated app-instance-8
Terminated app-instance-7
Terminated app-instance-6
Terminated app-instance-5
Terminated app-instance-4
Terminated app-instance-3
Terminated app-instance-2
Terminated app-instance-1
Current queue size : 0
0
Current queue size : 0
0
```



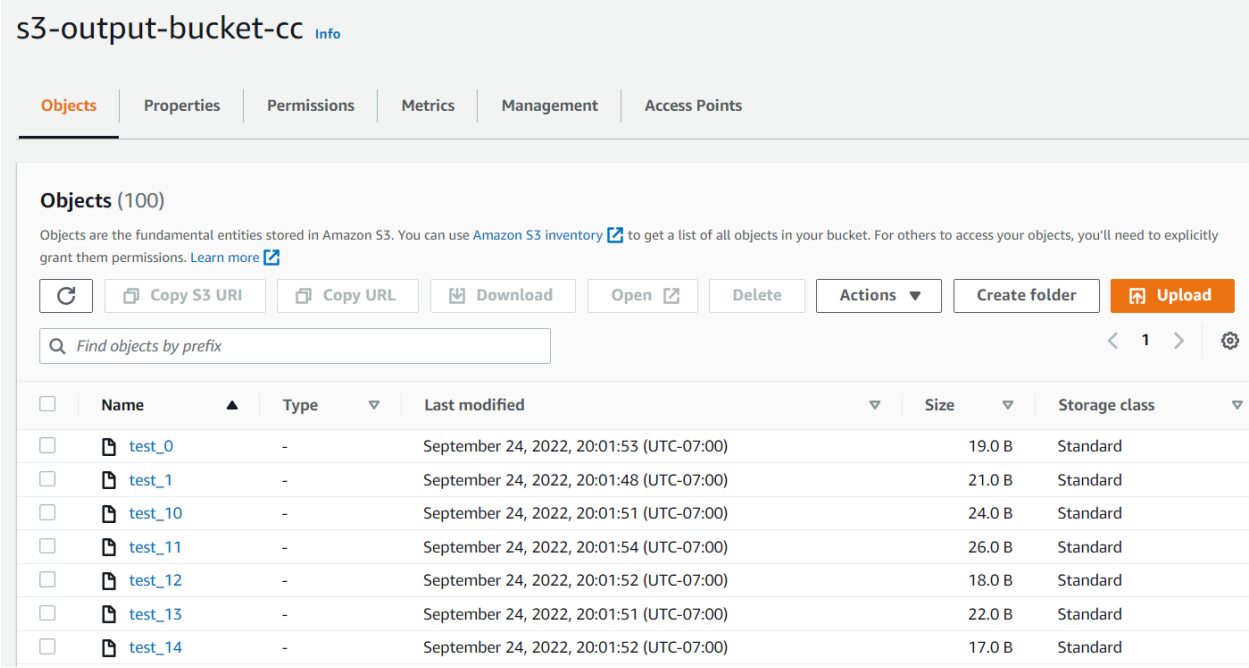
Messages received in the output SQS queue:

```
============================
Message received at :  2022-09-26 00:48:11.139977
Message Body : test_92.JPEG,mosque
Message ID : 260989cd-86a3-4134-a27e-2a80600037cc
Image Name : test_92.JPEG
UID : 07ce7e35-187a-4283-a9cb-a6a481afe54d
============================
============================
Message received at :  2022-09-26 00:48:22.257248
Message Body : test_99.JPEG,alp
Message ID : edb73bd6-f286-4861-8d2f-b9786a4cdc65
Image Name : test_99.JPEG
UID : 14a5a626-1a7a-4721-a45e-fcbb67a9ca42
============================
============================
Message received at :  2022-09-26 00:48:22.270241
Message Body : test_98.JPEG,yawl
Message ID : 351f9d0f-d003-4556-ac44-1b3a968f816b
Image Name : test_98.JPEG
UID : d1ec6c0e-98ef-4e17-ae1c-cd39158c1b5c
============================
```

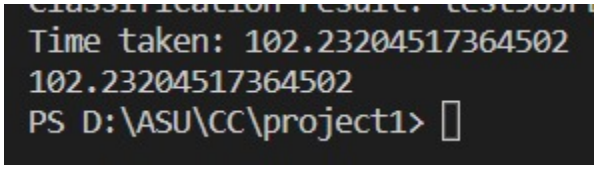
Images being populated into input S3 bucket:



Predicted output files populated in S3 output bucket:



Time taken to complete the processing and store the results for all 100 images:



4. Code

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.

Steps to run the program:-

- We started with creating the Web tier ec2 instance of type t2.micro and kept the remaining settings as default.
- As the Web tier would have the code to catch the API hit, we set some inbound rules.

Instance: i-02f8b67bb5e5f47c3 (webTierInstance)

▼ Inbound rules

Filter rules

Security group rule ID	Port range	Protocol	Source	Security groups
sgr-00b123a7c1cff8149	5000 - 5005	TCP	::/0	launch-wizard-2
sgr-0fc1ee47b8fec608e	22	TCP	0.0.0.0/0	launch-wizard-2
sgr-0fca223ca1a6b4f9c	80	TCP	::/0	launch-wizard-2
sgr-0667e860b9539d7fe	443	TCP	::/0	launch-wizard-2
sgr-0443e709f04d8e6fa	5000 - 5005	TCP	0.0.0.0/0	launch-wizard-2
sgr-0af541b4660d3a785	80	TCP	0.0.0.0/0	launch-wizard-2
sgr-0e707c3240c26235c	443	TCP	0.0.0.0/0	launch-wizard-2

- Next, install flask and boto3 in the web tier instance using the pip3 command.
    - pip3 install boto3
    - pip3 install flask
- Next using fileZilla, we transferred the appTier.py, scalingCode.py and outputQueueListener.py files to the web tier instance inside the ec2-user folder.
- Then, create the app tier ec2 instance using the AMI provided by the professor.
- Once created, connect via ubuntu and install the facenet_pytorch library using the pip3 command and transfer the recognition.py file to the app tier instance using fileZilla.
- Create an AMI with the changes in the app tier and copy that AMI id in the scalingCode.py file so that this snapshot is used for scaling out the app tier instances.
- Next, launch 3 terminals of the web tier.
    - In the first one, we run the flask application using the following commands
        - export FLASK_APP=appTier.py
        - flask run --host=0.0.0.0
    - In the second terminal of web tier, we run the scalingCode.py file via the command : python3 -m scalingCode
    - Finally in the 3rd WebTier Terminal, run the command : python3 -m outputQueueListener to start the listener.
- Now that all the components of the web tier are running, we can send the requests from our local terminal using the workload generator file provided by the professor.
    - We run the command python workload-generator-byProf.py --num_request 10 --url "http://54.147.84.220:5000/" --image_folder "D:/ASU/CC/imagenet-100/" where the url is the URL of the web tier and the image_folder parameter has the path for the folder containing the images.

appTier.py (Web Tier):

- Using the AWS credentials like the access key id, secret access key and input queue url we get access to the input queue using the BOTO3 library.
- This file contains the code for the flask application and is mainly used to bridge the gap between user and the input queue.

- We catch the API request sent and get the image that was sent by the user for classification. If the filename is not empty, it is saved in a folder called requests_files.
- We then generate a unique ID for the input image and encode the image before sending it to the input queue. Encoding is done because the SQS is incapable of storing images in JPEG format.
- Once the file is uploaded to the queue, we remove the input image from the requests_files and wait for the classification results.

recognition.py (App Tier):

- This file consumes messages from the input queue by calling the sqs.receive_message API constantly to check for any new incoming messages. Each message from the response of the queue is processed for further classification in the process_message method
- The process_message method extracts the message ID (UID), image name, and the actual image. The image is then base64 decoded to save it in JPEG locally in the appTier instance. Once this image is processed, the path to these locally saved images is sent to the classifier method.
- The code for the classifier method is adopted from the classification file provided by the Professor. This predicts the label from the image provided to it and returns the label.
- Once the classification is done, the image file is converted to an Object and saved into the input S3 bucket. The predicted label along with the fileName of the current image is saved in the S3 output bucket. The result with the file name, predicted label and unique ID is returned from the process_message method.
- This result is sent to the output SQS by calling send_message method. Once processing is complete for an image, the message is deleted from the input and output queues. All these steps are repeated till there are no messages remaining in the queue.

outputQueueListener.py (Web Tier):

- Using the AWS credentials like the access key id, secret access key and output queue url we get access to the output queue using the BOTO3 library.
- This file polls the output queue for any new messages. If it finds a new message(a new UUID), the code opens the text file with the UUID as the name and writes the predicted label in the file.
- Once the message is processed successfully, it is deleted from the output queue.

scalingCode.py (Web Tier):

- Using the AWS credentials like the access key id and the secret access key we get access to the input queue using the BOTO3 library.
- The get_instance_count method returns the input queue size. It uses the parameters: ApproximateNumberOfMessages','ApproximateNumberOfMessagesNotVisible'.

- The current variable holds the value of the currently running instances, which also helps in the naming of the newly created instances.
- The create_apptier_instances creates the apptier instances with ‘apptier-’ prefix and instances are created with the given AMI id, user data(which contains the commands to recognition.py) and security group. The scale out logic is briefly explained in section 2.2
- The terminate_apptier_instances method uses the terminate method depending on the tracker array. If the track array is filled completely with the same number (x) over its entire size,then x number of instances are going to be deleted.The scale in logic is briefly explained in section 2.2