CSE 546: Cloud Computing

Image Classification Application Project Report

Team: Cloud Nine (15)

Uma Maheshwar Reddy Malay
1222304759
umalay@asu.edu

Aayush Dewan
1222336453
adewan3@asu.edu

Venkata Lakshmi Narayana Obillaneni
1222713739
vobillan@asu.edu

1. Problem Statement

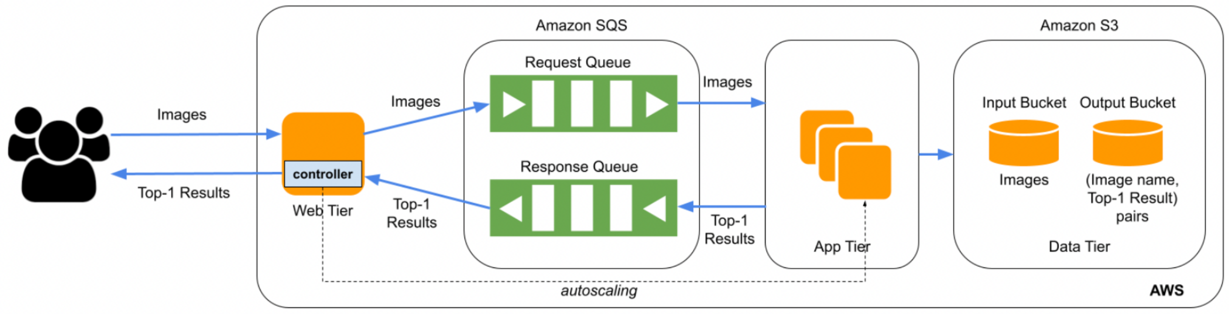
In this project the goal is to build an elastic application that can scale out and in dynamically, quickly, and affordably that takes in images as the input and runs a deep learning model on the images which processes and classifies them. The application should make use of various AWS services like EC2 for computation, S3 for storage, SQS for sending messages between different components.

This application will provide a service to users for sending an image and obtaining the classification result. This application which we build using this project will help us get a good understanding of AWS IAAS resources and help us learn technologies and techniques which will help us build many such applications.

2. Design and Implementation

2.1 Architecture

As we discussed, in this project we are building a cloud application which accepts images from a user, runs a deep learning model to classify the images and returns the result back to the user. We are limiting ourselves to only use AWS IAAS resources to build our application. Here are the different components in the architecture of the application.



a. Users

b. Web Tier
c. Input and Output Queue
d. App Tier
e. Storage

We'll have a look into what each of the components in the architecture does and then see the overall architecture and how it works.

a. **Users**: The users send images to our application by creating a file pointer to the image we need to read and send the file pointer as paremeter in their request (or API call) to the API which is exposed by Web Tier. Then they wait for the response and parse the classification result from the response and print it as the output to console.

b. **Web Tier**: The web tier needs expose an API to accept the images form the user of the application. Then it needs to queue the image into an input SQS queue and poll for results on the output SQS queue. We also need to keep track of the messages we are reading from the output queue and send the correct result corresponding to the image which this API call thread is handling. Web tier machine also has logic to implement auto-scaling for the service. It keeps monitoring the number of messages that need to be processed in the input queue and based on it, it will create additional App Tier EC2 instances where maximum instances can be less than equal to 20.

c. **Input & Output Queue**: These queues are used for communication between the Web Tier and the App tier. The Web Tier sends the images it gets as messages into the input queue and from here the App tier reads the images. Similarly, after processing the images the App tier sends the classification result as a message to the output queue from where the web tier gets the result and returns it to the user.

d. **App Tier**: App tier EC2 instances are created form an AMI image and all of them have the required python script to classify the given image and have our scripts which runs as a daemon. It first reads the image from the input queue and passes it to the deep learning model to get the classification result and sends the result to the output queue. It also makes requests to S3 buckets and sends the image and result which needs to be stored in S3. Also, after processing the images, it deletes the message from the input queue.

e. **Storage**: We created two buckets input bucket and output bucket. The app tier machines send two requests per image one to save the input image to the input bucket and send the result in the form (image name, classification label) to the output bucket.

Here is the end-to-end data flow. First, the user sends an image as input to the web tier. The web tier API accepts the image and sends it to the input queue as a message. From there the app tier machine gets the image and stores it on the EC2 instance and passes it as an argument to the *image_classification.py* script. The app tier also gets the output of this script and sends the classification label to the output queue and from there the web tier polls for this response and returns it to the user. The App tier also saves the input image and classification result to the AWS S3 buckets. The details of the resources created are included in the ReadMe file.

2.2 Autoscaling

Auto-scaling is a feature which helps applications to improve performance and reduce latency and this is very useful handle the requests when multiple requests are send concurrently. In this project we use horizontal scaling where we increase the number of instances in the App tier which work on processing and classifying the input images. To do this we need to scale-out and scale-in the number of app tier instances. The scaling logic is written in the Web Tier in a scaling.py script which runs as a daemon service on the web tier EC2 instance. Here is a detailed explanation of the scaling logic.

a. Firstly, we set up one EC2 instance with the AMI image provided in the project description (ami-0bb1040fdb5a076bc) which has the deep learning packages and script created. On this machine we also installed the required packages and setup the App tier script (serverapp.py, shellscript.sh and CRON job). Then we created a master AMI (ID: ami-0a156aab071a4bdd7) from this EC2 instance which has all the required packages and scripts for App tier to function.
b. In the web tier the scaling.py script takes care of auto-scaling, and it runs as a daemon service on the web tier machine. Periodically for every 5 seconds we monitor the number of messages in the input queue (both messages available and messages in flight).
c. Based on the number of messages we come up with the number of app-tier instances required. Then we check if the number of active instances is less than

the number required, we create the difference number of VMs and add their IDs to the set of active VMs. We use the boto3 ec2 create_instances API to do this. If the number of messages available is less (0), we scale down the app tier machines by terminating the instances we created using the ec2_client terminate_instances API.

2.3 Member Tasks

Member 1: *Uma Maheshwar Reddy*: Developed the web tier API using flask and python which accepts a post request from the workload generator, encodes the images and passes the encoded image to the SQS queue. Created a cronjob and a bash script shellscript.sh that automatically runs the serverapp.py script on the background as a daemon. Implemented the logic to properly get the results from SQS and store them so that appropriate results would be returned to user. Contributed to the implementation of auto scaling by making an AMI image from the first setup EC2 instance and investigated the get_queue_attributes API of SQS, create_instances API for EC2 and terminate_instances API of ec2 client.

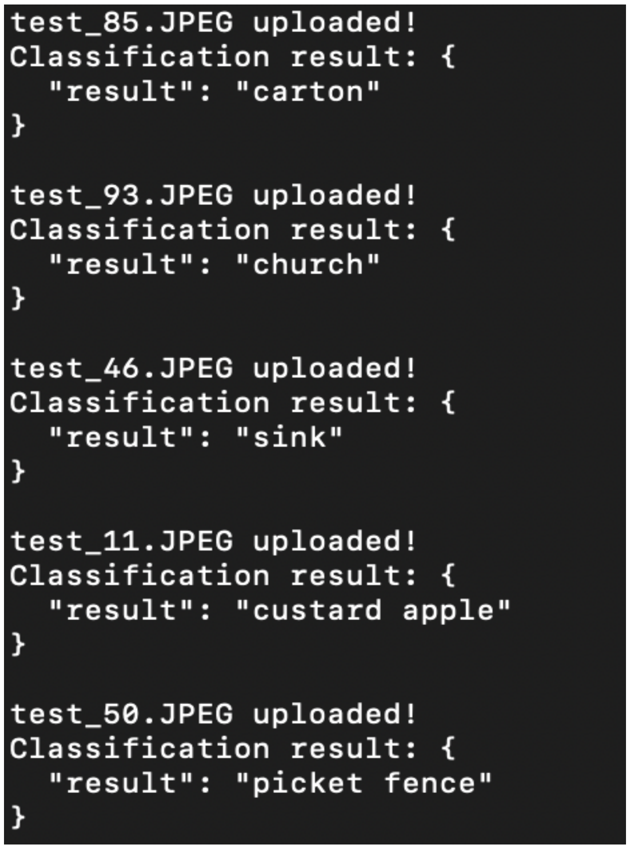
Member 2: *Aayush Dewan*: Setup the initial App tier machine and developed the code and logic in App tier using python that runs on App tier EC2 instances which accepts the data from the SQS, decodes the input images, and runs the image classification script which gets the output as image_name, label. Wrote logic to invoke the classification python script, parsing the output and deleting the image on the EC2 machine. Contributed to the implementation of auto scaling by implementing the scale-up logic and testing that the instances are properly scaled up when the number of messages in the queue increases.

Member 3: *Narayana*: Investigated and wrote code to invoke the APIS of S3_client such as upload_file and put_object for storing images and results in S3. Created two buckets inputBuckerCloud9 and outputBucketCloud9 firstly for storing input images and secondly for storing the output images. Wrote a script for deleting messages from SQS from the app tier. Contributed to the implementation of auto scaling by writing the scaling down logic and ensuring the instances are deleted after the images are processed.

3. Testing and Evaluation

From the project description we got the scripts which can be run on the user side called workload_generator.py and multithread_workload_generator.py which can be used for testing our implementation of the application. Initially before setting up auto-scaling we only used one app server and tested the end-to-end flow using at most 20 images. We recorded the time for each run, and we saw that it took approximately 3.5 minutes for 20 images using only one instance.

Then we implemented the auto-scaling logic and tested our end-to-end flow for image set of size 10, 20, 30, 50 and 100. In each case we checked if the number of App tier machines created by the scaling logic is as expected and saw if the latency is not increasing too much on increasing the number of images to 50 and 100. We also checked that during the run the messages count in the input and output queues is increasing and then decreasing which is as expected. We then confirmed that the images we pass, and the classification results are successfully stored in the S3 input and output buckets respectively. Here are some of the output screenshots and the results we got.



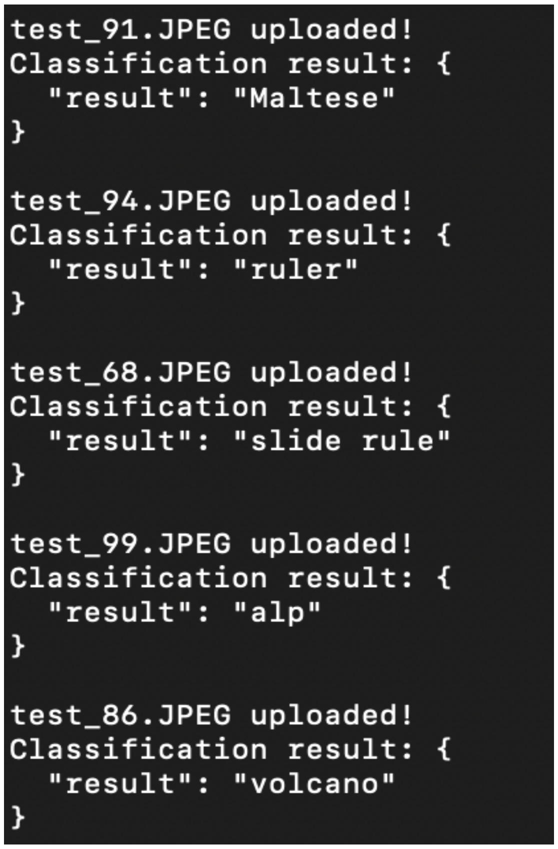
```
test_85.JPEG uploaded!
Classification result: {
  "result": "carton"
}

test_93.JPEG uploaded!
Classification result: {
  "result": "church"
}

test_46.JPEG uploaded!
Classification result: {
  "result": "sink"
}

test_11.JPEG uploaded!
Classification result: {
  "result": "custard apple"
}

test_50.JPEG uploaded!
Classification result: {
  "result": "picket fence"
}
```



```
test_91.JPEG uploaded!
Classification result: {
  "result": "Maltese"
}

test_94.JPEG uploaded!
Classification result: {
  "result": "ruler"
}

test_68.JPEG uploaded!
Classification result: {
  "result": "slide rule"
}

test_99.JPEG uploaded!
Classification result: {
  "result": "alp"
}

test_86.JPEG uploaded!
Classification result: {
  "result": "volcano"
}
```

Here is a screenshot from the AWS S3 showing storage of input and output.

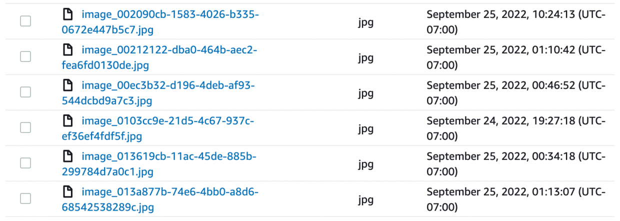
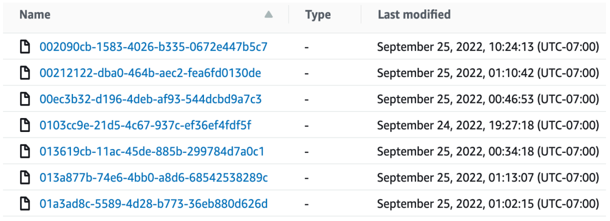


image_002090cb-1583-4026-b335-0672e447b5c7.jpg	jpg	September 25, 2022, 10:24:13 (UTC-07:00)
image_00212122-dba0-464b-aec2-fea6fd0130de.jpg	jpg	September 25, 2022, 01:10:42 (UTC-07:00)
image_00ec3b32-d196-4deb-af93-544dcbd9a7c3.jpg	jpg	September 25, 2022, 00:46:52 (UTC-07:00)
image_0103cc9e-21d5-4c67-937c-ef36ef4fdf5f.jpg	jpg	September 24, 2022, 19:27:18 (UTC-07:00)
image_013619cb-11ac-45de-885b-299784d7a0c1.jpg	jpg	September 25, 2022, 00:34:18 (UTC-07:00)
image_013a877b-74e6-4bb0-a8d6-68542538289c.jpg	jpg	September 25, 2022, 01:13:07 (UTC-07:00)



Name	Type	Last modified
002090cb-1583-4026-b335-0672e447b5c7	-	September 25, 2022, 10:24:13 (UTC-07:00)
00212122-dba0-464b-aec2-fea6fd0130de	-	September 25, 2022, 01:10:42 (UTC-07:00)
00ec3b32-d196-4deb-af93-544dcbd9a7c3	-	September 25, 2022, 00:46:53 (UTC-07:00)
0103cc9e-21d5-4c67-937c-ef36ef4fdf5f	-	September 24, 2022, 19:27:18 (UTC-07:00)
013619cb-11ac-45de-885b-299784d7a0c1	-	September 25, 2022, 00:34:18 (UTC-07:00)
013a877b-74e6-4bb0-a8d6-68542538289c	-	September 25, 2022, 01:13:07 (UTC-07:00)
01a3ad8c-5589-4d28-b773-36eb880d626d	-	September 25, 2022, 01:02:15 (UTC-07:00)

4. Code:

The code for this project mainly includes three different python scripts, a shell script, two Linux daemon service files and a cronjob file. We'll first explain in detail what each of the python script is for and how it is used. The python scripts are *webapp.py*, *scaling.py* and *serverapp.py*. The scaling.py script is for auto-scaling and is already explained in the Auto-Scaling section.

The *webapp.py* is run in the Web Tier to accept images from the user. The *scaling.py* script runs on the Web Tier which is for auto-scaling the App-Tier based on the number of requests and the *serverapp.py* file runs on the App-Tier to process the image. Here is a detailed explanation of the logic used in each of the scripts.

<u>WebApp.py:</u>

The webApp.py script runs on the Web Tier EC2 instance, and it exposes an API endpoint */queueImage* which the user will directly hit. This python script has an API called *queueImage.* We used python Flask framework to expose this API endpoint. Here is the step-by-step explanation of what the API does.

1. ***Accept Image from User***: First accept images from a user. Based on the input which is sent by the user as '*myfile*' argument in the request body we get the file pointer and read the contents of the image file.
2. ***Encode the Image***: Then we encode the content using base64 encoding and convert the image read into a string object.
3. ***Send image to InputQueue***: After that we create an sqs_client for the *inputQueue* and using the client we send the encoded string as a message into the queue. We also generate a random uuid for the image and send it as message attribute which can be used to uniquely identify the image (ImageID).
4. ***Poll for result in outputQueue***: Then we need to poll the *outputQueue* to get the classification label result for the image we sent. Since we don't have control over the order in which the results arrive, we maintain a global dictionary called *classification_results* to which every API call stores the label with ImageID as the key.
5. ***Processing and returning results***: In a while loop we first check if the result for the image, we are processing is in the *classification_results* dictionary. If it is present, we return the result stored in dictionary as the response. If it is not present, we poll the output queue get a message, parse the ImageID, label from the message and add it to the dictionary and delete the message from queue.

Storing into a global dictionary ensures that all the API threads will have access to the messages retrieved by any of the threads.

serverapp.py

The ServerApp.py is a python script and it runs on the App-Tier machines which basically accepts the images from the *inputQueue*, classifies them using our deep learning model and the queues the result to the *outputQueue*. Here is the step-by-step explanation of what the script does.

1. ***Get image from inputQueue***: First, we poll for a message from the inputQueue. If there are no images, we exit. If there are we get a response and parse the message and its attributes.
2. ***Decode and store image***: We then decode the message body and get the ImageID from the message attributes. We store this image in the format image_<IMAGEID>.jpg at /home/ubuntu/classifier path on the EC2 instance.
3. ***Call the script to classify image***: Then we run the command to classify the image using the image_classification.py script by passing the path to image (/home/ubuntu/classifier/ image_<IMAGEID>.jpg) to the script and get the output which is in the format (Image Name, Classification label).
4. ***Send result to outputQueue***: From the output we got, we parse the classification label and send it as a message to the outputQueue with the ImageID in the message attributes.
5. ***Store input and output to S3***: After that we upload the image from the image path to the S3 input bucket and the classification result to S3 output bucket in the form (image name, label). For this we create a S3_client use the boto3 APIs upload_file and put_object. We then delete the .jpg file on the EC2 instance using "rm -f <Image Name>.jpg" command.
6. ***Delete message from inputQueue***: Then we call the delete message API on the inputQueue to delete the image from the input queue using the receipt handle which we previously stored.

Shellscript.sh and CRON job:

To automate the serverapp.py execution we set up a shell script and a CRON job so that the script is automatically invoked in the background on Linux. The shell script runs a simple while loop 30 times with a sleep time of 2 seconds between two iterations. The command 'python3 serverapp.py' is invoked in each execution. Now we need to invoke the shell script periodically for which we use the Linux

CRON jobs. We set up a CRON job to run every minute which invokes the command "bash shellscript.sh". This invokes the shell script which then invokes the serverapp.py script.

<u>Scaling.py and Scaling.service:</u>

The logic of auto-scaling and the code in this section is explained in detail in the auto-scaling section. To summarize this script monitors the number of messages in the inputQueue and based on that it creates additional VMs from AMI image (ami-0a156aab071a4bdd7) which we created using the first EC2 instance that we created. An additional feature we added was to run this scaling script automatically as a daemon in the background on the Web Tier so that the user need not manually execute it each time. The scaling.service configures to run the command "python3 scaling.py" automatically as a daemon and restarts it if it stops due to any reason such as system restart or failure.

<u>Installation and Execution:</u>

To run the cloud application which we created we need to first login to the AWS console and start the EC2 instances WebTierServer and AppTierServer from the EC2 dashboard. Since the services in AppTier are automated to run, they will automatically start in the background on starting the instance. Even the scaling.py service on WebTier also runs automatically. We only need to run the WebApp.py script. For that we need to enter a virtual environment, go to the folder with webApp.py and run it. Here are the steps.

a. Connect to WebTierServer as ec2-user using the WebTierKey.pem file.
b. Run command "*source env/bin/activate*" to enter the virtual environment.
c. Run "*cd webTier*" and then run "*python3 webApp.py*" to start the flask API service.

Then run the workload_generator.py or multithread_workload_generator.py to Communicate with the web tier as given in the GitHub link of workload generator.

Conclusion: This report summarizes the end-to-end implementation details of the image classification application project and doing this project we learned the concepts and technologies such as AWS, Python, Flask, Rest APIs, CRON jobs, Bash scripts and Linux daemon jobs. With the project we got a good understanding of AWS IAAS resources such as EC2, SQS, S3 and AMI images.