CSE 546 — Project Report

Adwait Patil- 1222493688
Akul Rishi - 1222359905
Dhaval Shah - 1223303770

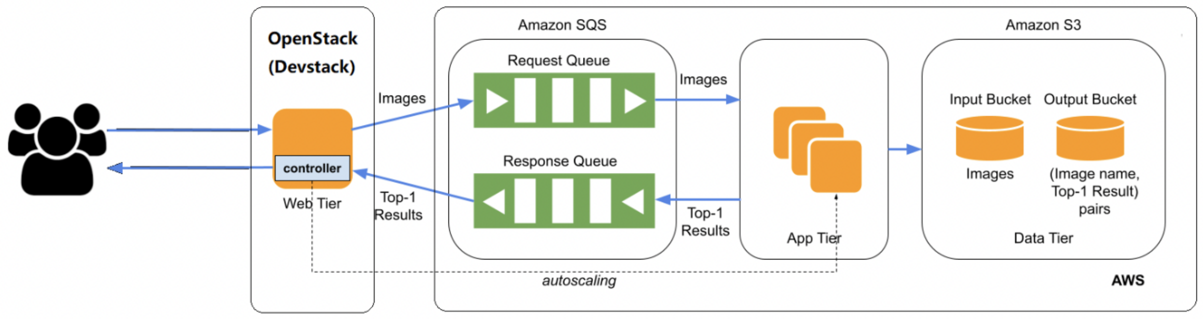
1. Problem statement

On Amazon AWS, we created Project 1's elastic image recognition program, which scales up and down as needed (public cloud). In order to simulate a hybrid cloud, we plan to employ openstack, which will operate on our hardware stack (private cloud), together with Amazon AWS (public cloud), to autoscale our image recognition depending on demand.

2. Design and implementation

2.1 Architecture

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



Elastic Compute Cloud (EC2), Simple Queue Service (SQS), and Simple Storage Service were three Amazon services that we heavily utilized (S3).
As a developer stack for the private cloud, we also utilized Openstack.
There are four parts:
1. Web Tier 1 (Openstack compute instance)
2. Queues for requests and responses (Amazon SQS)
3. Level of Application (EC2 and AMI)
4. A container for input and output (Amazon S3)

Web Tier:

The elements of the Web Tier were as follows:

A Linux instance running on Openstack serves as the web tier, and we upload our web-tier code (including the main file, scaling code, and output listener) there using SSH.

b) a Flask program (appTier.py), which served as the host server and took user-submitted photos. An exclusive ID is produced for every picture. The image and its ID are sent as a message to the request queue. This code searches a folder for a text file with the Unique ID included in its name once the message has been placed into the queue.

c) Response Queue Listener, which monitors the response's messages and creates a text file with the image's classification and unique ID in its name.

d) 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 AMI with the image recognition model is part of the application tier. When any instance with the AMI is generated, it also instantiates an image classification and listener file in addition to the model. In order to activate the image recognition file that categorizes the photographs, we gave the required Ubuntu instructions while establishing the instance using the AMI. The picture and the classification result are placed in the input and output S3 buckets, respectively, and the classified label and its distinct ID are pushed to the Response queue. The quantity of requests received from the Web Tier determines how many instances of the App Tier are produced.

Reaction and inquiry Queues: These queues serve as buffers to handle the message delay brought on by the image recognition logic's processing time. Through the use of queues, we are able to scale by separating the web and application tiers.

Input and Output Bucket: This makes use of the incredibly robust S3 (Simple Storage Service). The user-submitted photographs are kept in the input bucket, and the categorized label for the associated image is kept in the output bucket.

2.2 Autoscaling

In this project, we wanted to make use of autoscaling, which is one of the key aspects of cloud computing. We must build and terminate instances in accordance with the design and architecture described above depending on the input image/requests. As a result, we raise the number of instances when the number of requests rises and decrease the number of instances when the number of requests fall.

1) The first step is to determine how many requests are currently in the input queue. The parameters "Approximate Number of Messages," "Approximate Number of Messages Not Visible," and "Approximate Number of Messages Delayed" are used to accomplish this. The size of the queue will be determined by adding the two factors.

2) The following step is to find out how many instances are active or pending. Using the Instances Filters command, we can filter the instances by specifying the statuses of operating and pending and the Application Tier name. This will provide us with a count of instances that are now running or about to start operating (number of instances in the pending or running stage).

3) The number of instances that are active right now is stored in the current variable. The number of instances should not exceed 19, since we are only allowed to utilize 20 instances, therefore this will also assist us decide when to start and stop running the instances.

4) Scaling Out: From step 2, we already know how many instances are active or waiting. We utilized the ceiling value of (queue size/2) to calculate the approximate number of instances required to handle the queue. We scale out, or generate new instances, if the required number of instances is more than the number of instances that are already operating or waiting. Additionally, we must monitor that there are no more than 19 instances at any given moment, which may be done by utilizing the current value from step 3.
Then, a 1. increase is made to the current variable.

5) Scaling In: We utilized the ceiling value of (queue size/10) to calculate the approximate number of instances required to handle the queue. We also use a counter to make sure that the termination of instances only happens when the needed number of instances stays constant for a long time. This is carried out in order to avoid accidental deletion. Instances may be generated immediately if the queue fluctuates, but by the time they are created, the queue size will have decreased and the deletion process will have started.
We scale in, i.e. terminate the instances, when the demand has decreased, if the number of instances needed is less than the number of instances in the running or waiting state. The total difference between the number of instances required to process the queue and the number of instances in the running/pending state is used to determine how many instances need to be terminated.
Then, a 1 is subtracted from the current variable. We have applied scaling operations both inward and outward using the boto3 Python library.

2.3 Member Tasks

Adwait:-
I began by setting up VirtualBox 7.0.0 and making a VM using the settings provided by the TA in the video that was published on Canvas. As stated on the project page, I gave the virtual machine 50GB of storage space and 4GB of RAM. In addition, I allocated 4 virtual CPUs for quicker, parallel processing.
Using the Ubuntu 20.0.4 ISO image, I then started the virtual machine. I initially launched it using Ubuntu 22 but switched back to 20.0.4 after watching the TA's video.

Last but not least, I began with the OpenStack installation where I encountered a few problems, including the attribute of type name, which was resolved by switching pyOpenSSL from version 22.1.0 to version 22.0.0. As a result, after some debugging and checking, I was able to install OpenStack and access the dashboard.

Dhaval:-
We had to eliminate the resources due to free tier limitations, thus I accepted the portion where I had to duplicate the AWS architecture of project 1. I tested the entire project after the architecture was developed before exporting the web tier AMI to the VM.
Following the testing, I exported the web tier AMI (vmdk file) using the procedures listed below:
-
1) Establish a new S3 bucket to house the exported AMI.
2) Give this S3 object object write permissions and enable ACL for it.
3) use the AWS CLI to execute the following export command.
4) After the nova compute instance built from the web tier AMI's network settings were complete, I created

Akul:-
After the AMI was exported, I downloaded it into a virtual machine and used it to build an image. I then configured the network in the following way: -
I went to the networks tab in openstack and completed the actions listed below:

a. Added a private network with IP 192.168.10.0/24 by selecting create networks and clicking it.
b. I added a new router with the external network set to public under routers.
c. I selected "Manage Rules" under "Security Groups" and added three rules for Custom TCP, Custom ICMP, and Custom UDP.
d. I made a fresh floating IP.

started working on the project's last phase, using SSH to launch the web tier code. I launched the flask server, started the auto scaling program, and started the daemon that watches the output queue for updates by running the appTier.py, scaling.py, and outSqsListener.py files. After finishing this, I thoroughly tested the entire infrastructure by running through several payload scenarios. We discovered that because this is a hybrid cloud, it took a little longer than the public cloud.

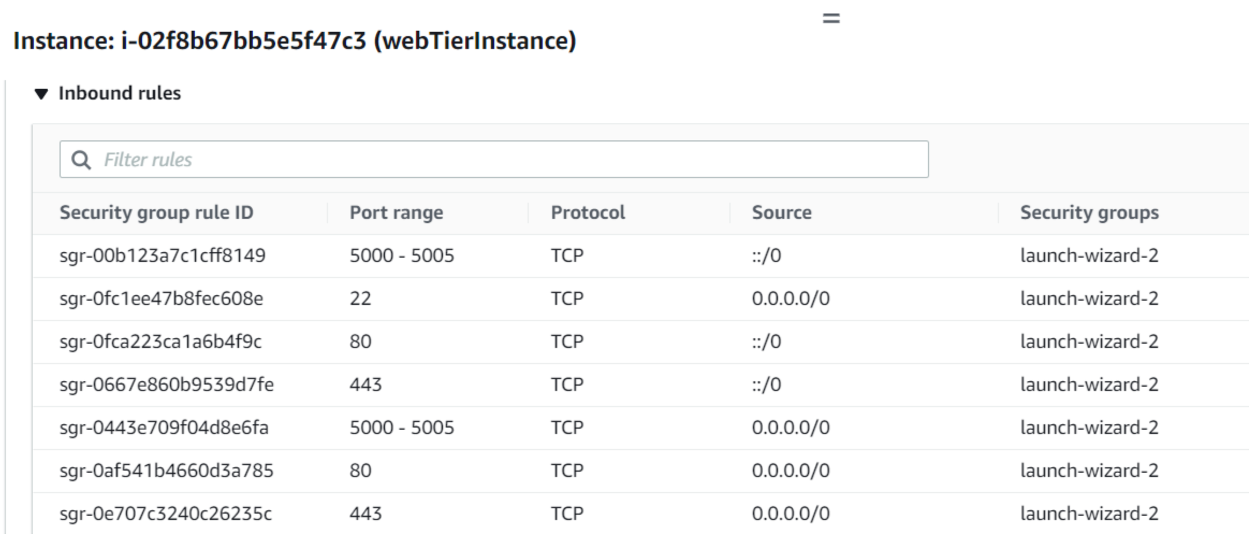
3. Testing and evaluation

- By checking to see if the code files we moved to our ec2 instances are executing independently, we first
- The photos were then examined to see if they were uninterruptedly flowing from beginning to end. In order to do this, we transmitted one image at a time, checked to see if it was being used by the input queue before being sent to the output queue, and verified that the classification result was being saved in the appropriate format in the S3 output bucket.
- We sent 100 requests from the workloadGenerator to simulate the full load while also keeping an eye on the flow. We also verified whether there were more than 20 active instances.

4. Code

Steps to run the program:-

- The Web tier ec2 instance of type t2.micro was the first thing we created. All other settings were left at their default values.
- We established some inbound rules because the Web tier would have the code to handle the API hit.



Instance: i-02f8b67bb5e5f47c3 (webTierInstance)

▼ Inbound 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
- We then moved the outputQueueListener.py, scalingCode.py, and appTier.py files to the web tier instance located inside the ec2-user folder using fileZilla.
- Then, create the app tier ec2 instance using the AMI provided by the professor.
- Create a connection using Ubuntu, run the pip3 command to install the facenet pytorch library, and then use fileZilla to transfer the recognition.py file to the app tier instance.
- To ensure that this snapshot is used for scaling out the app tier instances, create an AMI with the changes to the app tier and copy that AMI id into the scalingCode.py file.
- We then created an AMI for the web-tier

  1)Create a new S3 bucket which will store the exported AMI
  2)Enable ACL for this S3 object and give it object write permissions
  3)Run the following export command from AWS CLI:
  aws ec2 export-image --image-id ami-0c4dcc007b6844017 --disk-image-format VMDK --s3-export-location S3Bucket= webtierimage,S3Prefix=exports/

- 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.
- We may submit the requests from our local terminal using the workload generator file provided by the professor now that all of the web tier's components are operational.
    - Then running 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.
- Using the commands listed in this openstack installation tutorial, install openstack on Ubuntu: After logging onto your Ubuntu virtual machine, visit https://docs.openstack.org/devstack/latest.
- The HOST IP address and HOST IPv6 address, which can be discovered by running the ipconfig command in the Ubuntu terminal, are the only modifications we made to the devstack local.conf file (compared to the instructions in the tutorial).
- Use the given URL and credentials to access the horizon dashboard after installing openstack.
- The web tier instance is now ready after downloading the web-tier AMI for Ubuntu and launching a nova compute instance from it.
- Execute the remaining commands as in Project 1

appTier.py (Web Tier):

- We gain access to the input queue by using the BOTO3 library and the AWS credentials, including the access key id, secret access key, and input queue url.
- This file, which primarily serves as a link between the user and the input queue, contains the code for the flask application.
- We intercept the API request and retrieve the image that the user sent for classification. The file is saved in a folder called requests files if the filename is not empty.
- The image is then encoded and given a special ID before being added to the input queue. Because the SQS is unable to store images in the JPEG format, encoding is required.
- We remove the input image from the requests files after the file has been uploaded to the queue and then we wait for the classification results.

recognition.py (App Tier):

- This file continuously checks for new arriving messages by calling the sqs.receive message API, which it uses to consume messages from the input queue. In the process message method, each message from the response of the queue is processed for additional classification.

- The message ID (UID), image name, and actual picture are all extracted via the process message method. After base64 decoding, the image is saved locally in the appTier instance as a JPEG. The path to these locally saved photos is supplied to the classifier function after this image has been processed.
- The professor's classification file is used as the source code for the classifier algorithm. This extracts the label from the supplied image and returns it.
- The image file is transformed to an Object after classification is complete and saved into the input S3 bucket. The S3 output bucket stores the expected label and the fileName of the current image. The process message method returns a result with the file name, predicted label, and unique ID.
- By using the send message method, the output SQS receives this result. Images that have finished processing are removed from the input and output queues. Until there are no more messages in the queue, all of these stages are repeated.

outputQueueListener.py (Web Tier):

- We gain access to the output queue by utilizing the BOTO3 library and the AWS credentials, such as the access key id, secret access key, and output queue url.
- This file periodically checks the output queue for fresh messages. The code opens the text file with the UUID as the name and inserts the anticipated label inside if it discovers a new message (a new UUID).
- The message is removed from the output queue once it has been successfully processed.

scalingCode.py (Web Tier):

- We gain access to the input queue by utilizing the BOTO3 library and the AWS credentials, such as the access key id and the secret access key.
- The size of the input queue is returned by the get instance count method. ApproximateNumberOfMessages and ApproximateNumberOfMessagesNotVisible are the parameters used.
- The value of the running instances is stored in the current variable, which also aids in giving the newly formed instances a name.
- Using the 'apptier-' prefix, the create apptier instances function generates apptier instances with the specified AMI id, user data (which contains the commands for recognition.py), and security group. Section 2.2 provides a quick explanation of the scale out logic.
- Depending on the tracker array, the terminate apptier instances method employs the appropriate termination technique. A certain number of instances will be removed if the track array is completely filled with the same number (x) throughout its whole size. Section 2.2 provides a quick explanation of the scale in logic.