CSE 546

Project I: IaaS report

Abhishek Annabathula
aannaba1@asu.edu
ASUID:1225491722

Laksh Gangwani
lgangwan@asu.edu
ASUID:1225578211

Rohan Dvivedi
rdvivedi@asu.edu
ASUID:1224063958

1. Problem Statement

In this project, we are tasked with building a distributed cloud application that can process large volumes of image classification tasks. The API of this application should be accessible over HTTP protocol. This project also addresses the problem of auto-scaling the cloud application to handle the variable load. Here, we spin up more EC2 instances for short durations when there is a higher number of concurrent requests, while we also bring these excess instances down in order to incur lesser costs.

The high-level architecture of the project and the high-level view of the interactions between different components were given to us in the project description. The project has been built entirely according to the given project description. The high-level architecture has been explained in detail in the next section.

2. Design and Implementation

2.1 Architecture

As per the project description, this project must consist of the following components: a web tier EC2 instance, 2 SQS queues (one for requests and another for responses), 1 or more (at most 20) App tier instances, and 2 S3 buckets (one to store requests and another for classification results). Please refer to the image below along with the explanation to understand the high-level architecture of the project and the interactions between different components.

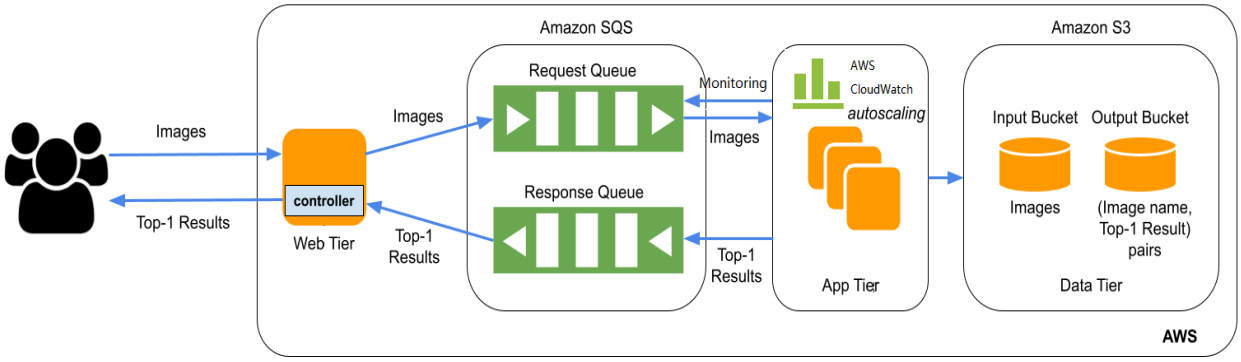


Image source: project description (CSE 546, Spring, 2023)

As shown in the figure above, the project consists of a single web-tier EC2 instance running an HTTP server. This instance is responsible for receiving requests from the HTTP clients. The clients must send an image (with its name) to this server (as multipart/form-data) to get their image classified.

Upon receiving the request the web-tier instance is required to push this image (with its name) into a request SQS queue. Now the web-tier instance waits until it receives a message in the response SQS queue for this corresponding request. Upon receiving a message from the response queue (corresponding to the given request) it will then send the classification result to the client as an HTTP response in text/plain format.

Meanwhile, App tier instances are all waiting to receive messages from the request SQS queue. After a message is received, the image will be processed by the deep learning image classification algorithm, and then the classification result (with the image name) will be pushed to the response SQS queue. Additionally, the image and image classification result will be stored in 2 different S3 buckets, keyed by their image names.

Deep learning to classify images is a slow task, while HTTP request handling has its own problems with network latency and timeouts. Hence SQS queues are used to buffer requests and responses between the web tier and the app tier.

Due to resource limitations in the free tier, we have a cap on the maximum number of EC2 instances. This cap (as per the project description) is set to 20 + 1 instances. 1 instance for the webtier and atmost 20 instances for apptier. The app tier is made to scale in or out depending on the queue size.

2.2 Auto Scaling

The Auto scaling of instances takes place for scaling the App tier. In order to Autoscale, two CloudWatch Alarms were created to monitor the queue length and trigger the scaling policies. The Auto Scaling Group represents the App Tier, with a minimum and maximum of 0 and 20 instances, respectively. When an alarm is triggered, a scaling policy is set up which scales in or out depending on the scenarios.

After the web tier processes the messages in the Response Queue, the number of messages in the queue is monitored by the metric "ApproximateNumberOfMessagesVisible". If the Maximum number of messages visible was greater or equal to a certain threshold at any time for a period of 10 seconds, the Alarm responsible for scaling out is triggered, and the scaling out policy, which references the Auto Scaling group, will come into action, resulting in an increase of the desired number of instances in the App Tier.

Similarly, if the metric was below a certain threshold, another alarm responsible for scaling down would trigger, and the desired number of instances would terminate from the App Tier because of the scaling in the policy. If there are no messages in the queue then all the running instances will be terminated.

For the purpose of the project, I decreased the cooldown period, which refers to the time interval that must elapse before another scaling activity can be triggered after the completion of the previous scaling activity of the Auto Scaling Group. However, this is not to be done in real-world scenarios, because a cooldown period of several minutes can be beneficial in preventing rapid fluctuations in the capacity that can cause issues such as increased costs or reduced performance, thus making the scaling in and out of EC2 instances much more efficient.

Overall, scaling in and out based on queue length is a powerful technique for automatically adjusting the capacity of our system to handle varying levels of workload. By monitoring the queue length metric and using CloudWatch alarms and AWS Auto Scaling, we ensure that our system can handle sudden spikes in traffic and maintain high levels of performance and availability.

2.3 Member Tasks

We divided the tasks required for the successful completion of this project into 3 parts, one for each of the three members. The task distribution is as follows:

Abhishek Annabathula	Implementation testing and deployment of **App tier**
Laksh Gangwani	Setup of all of AWS resources and **Auto scaling for App tier**
Rohan Dvivedi	Implementation testing and deployment of **webtier**

2.3.1 Tasks for App Tier (Abhishek A.)

- Set properties for the AWS utility functions like RequestQueue, ResponseQueue, RequestS3, and ResponseS3
- Implement utility to send and receive messages from SQS request queue and response queue respectively.
- Implement a utility to upload and download messages from the S3 request bucket and response bucket respectively.
- Write ImageClassification code to:
    - fetch images from the Request queue.
    - save it to request an S3 bucket.
    - Run ‘image_classification.py’ with the image.
    - Create a response object {image_name: recognition_result}
    - Save it to the response bucket.
- Add an error handler in the code to fetch the system signals and perform exit gracefully.
- Test the code locally
- Integrate with web tier
- Debug
- Testing

2.3.2 Tasks for Web Tier (Rohan D.)

- Setting up an empty Django project.
- Make a controller to receive HTTP requests, to receive images.
- Implement utility to send and receive messages from SQS request queue and response queue respectively.
- Implement a concurrent data structure (a python dictionary with locks and a condition variable) to store received messages from the SQS response queue.
- Add an additional thread to the application. That will:

    - wait to receive messages from the SQS response queue
    - On receiving a message, save the response in the global data structure (our global dictionary)
    - And then wake up all the threads that are waiting to receive the response message.
    - Delete the message from the queue.
- Implement logic in the controller to do the following:
    - Receive image from HTTP request.
    - Send the image in the request SQS queue in the format:
        - image_name + ":" + base64(image_data)
    - Go to wait on the condition variable, until the response is received
    - Once the response is received, send it to the client in an HTTP response
- Add error handling code, wherever necessary.
- Debug.
- Deploy the application on an EC2 instance.
- Configure and set up the application to use a production-grade Apache2 server.
- Test vigorously.

2.3.3 Tasks for Infrastructure & Auto Scaling (Laksh G.)

My main contribution included the architecture creation and handling of the scaling activities of Auto Scaling Group.

Infrastructure

- Created Users and grouped them in a group, giving them access to resources that I defined with the help of IAM.
- Created roles and policies using CloudFormation
- Used CloudFormation to provision and manage my infrastructure instead of manually configuring infrastructure
- Defined Infrastructure as a code. It makes it easier for me to manage and replicate infrastructure across multiple AWS accounts.
- Created two stacks, which are the network and Infra using CloudFormation.
- In the Network stack I defined the configuration of my network components
- The security group allows traffic from port 22 and port 80(HTTP).
- In the Infra stack, I created the resources which are required for this project.
- Managed both stacks and updated them to ensure consistency

Auto Scaling

- Created two CloudWatch Alarms, one ASG, and two scaling policies to implement the auto Scaling logic.
- Used the "ApproximateNumberOfMessagesVisible" metric to monitor the RequestQueue.

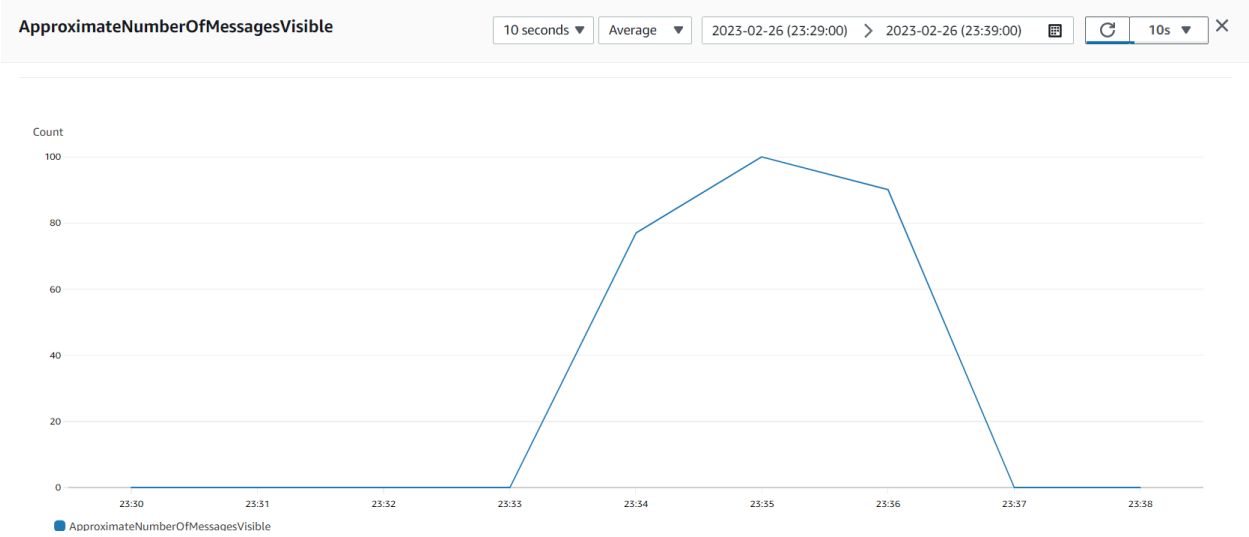
- Attached IAM role to the ASG
- Decreased the Cooldown period of the ASG so that scaling happens after a delay of a few seconds to accomplish the scaling up and down quickly.
- Monitored the “Activities” tab for the ASG and compared it against the RequestQueue metric “ApproximateNumberOfMessagesVisible”
- Used CloudWatch metrics to show results

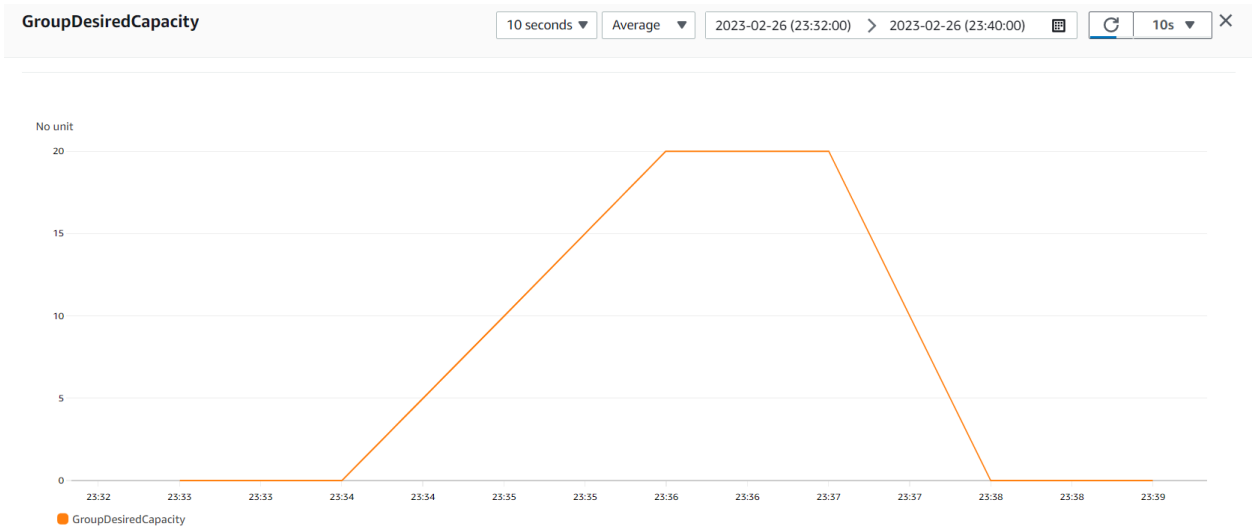
3. Testing and Evaluation

We have tested the codebase using the workload generators provided with the project description. We have tested with both multithread_workload_generator and workload_generator with all 100 images. The server that we are using in the web tier (apache2) supports atmost 700 concurrent requests. We tested using the command below:

python3 multithread_workload_generator.py --num_request 100 --url 'http://ec2-54-82-248-126.compute-1.amazonaws.com/image/push' --image_folder imagenet-100/

Results: All of the 100 requests received their responses. We monitored the number of EC2 instances and the queue size and we observed that 20 EC2 instances were spun up by the Autoscaling logic.





The two graphs illustrate the relationship between the "ApproximateNumberofMessagesVisible" in the RequestQueue and the number of instances in our Auto Scaling Group. The graphs show that when more messages were processed, the number of instances also increased linearly. Conversely, when there were no messages, all instances were terminated. The graphs demonstrate that the number of instances reached a maximum value of 20 before being scaled down to 0.

4. Code

4.1 Web Tier

4.1.1 Description

Even though the Web tier repository may contain plenty of files. Primarily, our logic is composed in the following 2 files only.

- CSE546-webtier/CSE546WebtierImage/imagepushcontroller.py
- CSE546-webtier/CSE546WebtierImage/imagequeue.py

All other files are bootstrap logic required for the Django framework. We might have made a few changes to these other files here and there, but they are not of major importance to understanding the inner workings of the project.

Imagepushcontroller.py consists of the function “pushcontroller” that gets called when a POST HTTP request is made to the web server at URL path “/image/push” with an image file as its contents in multipart/form-data format.

If this happens, we (in imagepushcontroller.py) call functions in imagequeue.py to do the following:

- mark a flag in the global data structure, denoting that we are looking forward to receiving a classification result for the given file name.
- Then we go ahead and send the image to the request SQS queue. As discussed above the image format is image_name + “:” + base64(image_data). Here we have to base64 encode the image, because the image is binary data, while SQS queues can only accept ASCII strings.
- After that we go on to wait until a timeout of 15 minutes expires or the classification result for the image is received.

The major logic for the functioning of the webtier lies in imagequeue.py. It does the following tasks:

- It will initialize the global data structure (a simple python dictionary) and a mutex lock and a condition variable associated with it. This lock must be taken while accessing this global data structure. While this condition variable must be held when you are waiting to be notified of any changes in any values of this data structure.
- It will also start a separate background thread, that will keep running forever, this thread will wait to receive messages from the response SQS queue. These messages are of the format image_name + “:” + classification_result. If a valid message is received, then this classification result is then stored corresponding to the image_name in the global data structure, if it is marked required. Once set, It will then wake up all the threads that are waiting for a change in the values of the global data structure.
- It provides a utility function to send an image with its name to the request SQS queue, in the above-mentioned data format.
- It provides a utility function to mark a result as required, corresponding to an image_name.
- Finally, it provides a utility function to wait until a result is received. It will loop to check if the result is already present in the global data structure, if not it will go ahead and wait on a condition variable until any change occurs (or until the timeout expires).

4.1.2 Setup

- git clone git@github.com:RohanVDvivedi/CSE546-webtier.git
- cd CSE546-webtier
- pip3 install pipenv

- pipenv install
- Now to run development server you can do
    - pipenv run ./manage.py runserver 0.0.0.0:8080
- sudo apt install apache2 apache2-utils libexpat1 ssl-cert libapache2-mod-wsgi-py3
- sudo a2enmod mod_wsgi
- Copy the below site file to /etc/apache2/available-sites/webtier.conf

```
<VirtualHost *:80>

DocumentRoot /home/ubuntu/CSE546-webtier

<Directory /home/ubuntu/CSE546-webtier/CSE546webtier>
        <Files wsgi.py>
                Require all granted
        </Files>
</Directory>

WSGIDaemonProcess webtier_group threads=700 socket-timeout=1000
python-path=/home/ubuntu/CSE546-webtier
python-home=/home/ubuntu/.local/share/virtualenvs/CSE546-webtier-jOGGz7x0
WSGIProcessGroup webtier_group
WSGIScriptAlias / /home/ubuntu/CSE546-webtier/CSE546webtier/wsgi.py

</VirtualHost>
```

- sudo a2ensite webtier.conf
- systemctl start apache2
- systemctl enable apache2

4.2 App Tier

4.2.1 Implementation Details of App Tier:

In the AppTier Module, there are four files AWSUtils.py, AppTierProperties.py, ImageClassifier.py, and AppTier.py. The implementation flow runs as per the below sequence.

1. AppTierProperties.py
   This file consists of the class that initialized the RequestQueue, ResponseQueue, Request_S3, and Response_S3 along with the AWS keys.

2. AWSUtils.py

This file contains the implementation of the utility functions such as reading messages from the request queue, pushing messages to the response queue, reading messages from the S3 bucket, and writing the result to the S3 bucket along with the initialization of the local variables with the values imported form AppTierProperties.py.

3. ImageClassifier.py
   The class handles the reading of the messages from the request queue by running a while loop. As soon as there is a message in the queue, it is read and sent to the deep-learning model for image recognition and classification. Once the recognition is done, the results are fetched and written to the Response S3 bucket and pushed into Response Queue. The image received from the Request Queue will be in {image_name: base64 encoded image string}. The message is read and converted back to byte code and written into the Request S3 bucket as a jpg file. A response dictionary is created with {image_name: recognition_results} format.

4. AppTier.py
   ImageClassifer.py is imported into this file which executes the Classifier function that in turn calls the image_classification.py. This file runs on the EC2 instance as soon as an instance is up and running.

5. An IAAS.service file is created so as to run the AppTier.py script whenever a new instance is up and running.

4.2.2 Setup

- git clone https://github.com/aabhi98/IAAS_CC_Spring_23.git
- cd IAAS_CC_Spring_23/AppTier/
- Create a systemd service for the application and name it IAAS.service
- systemctl enable IAAS.service
- systemctl start IAAS.service