CSE 546 --— Project Report

Hari Sai Charan Challa(1225461861)

Sathwik Katakam(1225445585)

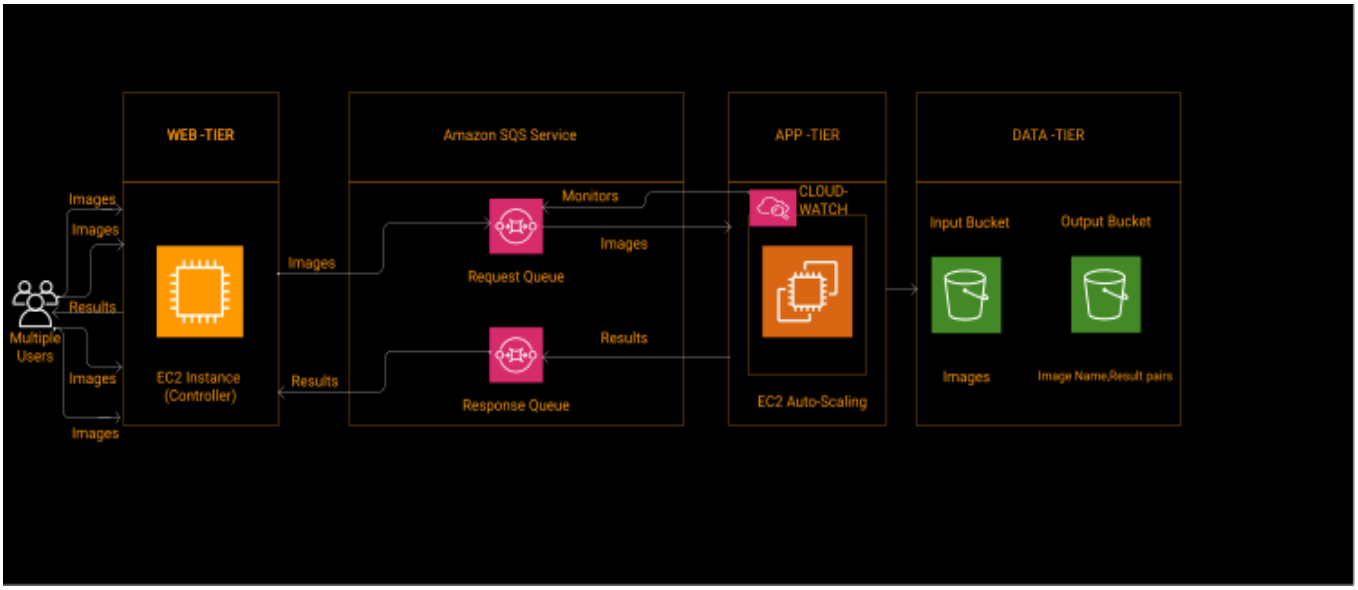
Hari Murugan Ravindran(1222324688)

1. Problem statement

In this project, we are building an elastic image recognition application which detects the images provided by the users with the help of the deep learning model. We are building our application in such a way as to scale out and scale in on-demand effectively to accommodate the sudden surge in the number of users. We are leveraging the performance of our application with the help of Iaas resources provided by Amazon Web Services(AWS). We are allowing users to access our application in the form of RESTful services. Our application takes images as input and provides the recognition result as the output to the users with the help of various kinds of AWS Services for Storage, Computation and message services.

2. Design and implementation

2.1 Architecture



Architecture 1.1

Description:

Our application is divided into three sections:

1. The Web Tier
2. The App Tier
3. The Data Tier.

Multiple users concurrently access our application by uploading input images through the Web Tier. The images are then passed through the Web Tier instance and added to the request queue. In the request queue, the images are sent to the App Tier for image recognition. We use a deep learning model provided to us as an AWS image to detect the image and output the result. The App Tier utilizes the auto-scaling group controlled by CloudWatch, which monitors the request queue. We scale out our instances based on the depth of the request in SQS. We have set a maximum threshold of 20 instances.

Therefore, while scaling out the App Tier instances to meet the increase in demand, we won't exceed the threshold. Similarly, when the demand decreases, our App Tier instances are automatically scaled in, for which we have set a lower threshold of 1 instance. Our application stores the computed results in the Data Tier, which consists of two S3 buckets for persistence. We have an input bucket where the input images are stored and an output bucket where the image name and output result are stored in pairs. The output is also sent to the users through the response queue.

Usage of AWS Services:

- **AWS EC2:**
    - We use Amazon EC2 (Elastic Cloud Computing) for our computational needs. We utilize virtual servers for both the Web Tier and App Tier.
    - In the Web Tier, we use a single instance that acts as the controller. It receives input images from the users who concurrently access our application and pushes them to the queue.
    - Additionally, it is responsible for retrieving results from the response queue and delivering them to the users.
- **AWS SQS:**
    - We use AWS SQS to manage communication within our Web Tier and App Tier. The Web Tier instance pushes the given input images into the request queue, and the instances in the App Tier retrieve the images for image detection.
    - The App Tier instances detect the image and retrieve the next images as input from the queue.
    - Similarly, the output is pushed to the response queue, where the output is sent to the users who uploaded the images as input.
- **AWS CloudWatch:**
    - We use AWS CloudWatch to monitor the request queue and scale out or scale in our App Tier instances based on user demand. We have made our App Tier instance a part of the auto-scaling group, based on the depth of the Request SQS. This allows us to adjust our capacity as needed, without exceeding the maximum threshold.

- **AWS S3:**
    - We use two S3 buckets for data persistence, which serves as our Data Tier. We store the input images provided by the users in one bucket called the input bucket, and the image name and result as pairs in another bucket known as the Output bucket.

2.2 Autoscaling

Autoscaling happens in our application with the help of Cloudwatch. This ensures that there are enough EC2 instances running to handle the load of the application. EC2 instances are grouped into an Autoscaling group. A launch template is created for the auto-scaling group with. A metric alarm is created based on the number of messages in the SQS queue which creates new EC2 instances accordingly. We use step scaling policies based on the above scaling metric. We define a lower threshold for the EC2 instances as 1 and the upper threshold as 20. 1 instance is allocated to the web-tier and 19 is the max-limit for app-tier. This ensures that even if the number of messages in SQS is above 19, we still create only 19 instances and don't run above the threshold. The cloud watch alarm keeps monitoring the SQS queue and scales app instances.

2.3 Member Tasks

2.3.1 Hari Sai Charan Challa

Design

I was involved in the brainstorming sessions we had to decide on the architecture to use for our application. After going through various AWS resources, we decided on which resources to pick and how to link them.

The major contribution of mine in this project was in the design and implementation of app-tier. Leveraging the EC2 instances, the app-tier listens to SQS queue and fetches messages as soon as they get populated from the web-tier. Once this is done, the image is passed to the Face recognition model for classification. Input image along with the output result we fetched from the model is being written back to S3-output bucket for persistence. After writing the result to S3, a message is sent to the response queue.

Implementation

- The code for App-tier was implemented in Python3. Boto3 was used as the AWS SDK to create and configure AWS services.
- Created a util module for the S3 related functionalities so that both web tier and app-tier can leverage them to write and read from S3 buckets.
- A listener is included to fetch messages from SQS and the message payload is being decoded to get the image file path and contents.
- We then delete the image from the message queue so that the input queue is cleared.

Testing:

There were various phases of testing involved in designing the app-tier for the application.

1. Unit-testing: There are several individual modules involved in the app-tier. Tasks such as receiving messages from SQS queue, writing to S3, classifying the image were tested separately to ensure proper working.
2. Integration-testing: Once the web-tier and cloud-watch alarms were set up, I performed integration testing to verify images or requests sent by the client are being passed from the web-tier to SQS and reaching the app-tier. In-return, the results being written to S3 bucket were correctly passed on to the response SQS queue that reaches the web-tier.
3. End-end testing: This phase involved testing the entire application as a whole to ensure that auto-scaling was happening at the app-tier properly and not more than 19 instances were created at any time. Output from the S3 bucket was verified with the expected output.

2.3.2 Sathwik Katakam

Design

I was majorly involved in the design of auto-scaling and AWS configuration setup required for the project. I have created AWS IAM User Credentials required for the group and collaborated with my teammates to figure out the AWS services to use. We have decided on using EC2 for app-tier and web-tier, SQS for the request message queue and S3 buckets to store the inputs and outputs. After exploring several options for the auto-scaling, I have leveraged the Step-scaling policy of AWS to do the autoscaling.

Implementation

I have initially set up the EC2 instances required for both the Web and App Tier.Collaborated with teammates and created the necessary request and response queue as well as the S3 buckets required for the persistent storage.

Auto-scaling was an important step in the project to handle concurrent requests. We had to scale out the EC2 instances with a threshold of 19 when there are multiple requests in the message queue. The application had to automatically terminate EC2 instances not being used when there aren't enough messages in the SQS queue.

To achieve this, I have used the Step-scaling policy of AWS that groups EC2 instances into a collection.

1. An auto-scaling group is created with a new dynamic scaling policy.
2. Two new cloud-watch alarms were created for scaling-in and scaling-out respectively.
3. The alarms were created based on the SQS metric - ApproximateNumberOfMessagesVisible. The scale out alarms sets the capacity of EC2 instances between 1 and 19 based on the above metric. It gets triggered when the messages visible in the queue are >= 1. Similarly the scale-in alarms get triggered when the messages in the queue go below 1.
4. A launch template is created with the above alarms

5. To launch app-tier automatically on starting an EC2 app instance, bash script was created and provided as user-data commands in the launch-template which would clone the code and start app-tier once an instance is created.
6. We link this cloud-watch alarm to our EC2 instance which completes the auto-scaling setup.

Testing

1. Unit-testing: Cloud watch alarms are carefully monitored to see if they are getting triggered when messages in SQS get populated. Verifying the activity of the alarms ensures that EC2 instances are getting spawned according to the rules set up.
2. Integration and End-end testing: Once the web tier and app tier are set up and integrated, we sent requests ranging from 1-100 and tested the auto-scaling mechanism. We ensured that linking between cloud-watch and EC2 instances were correct and alarm triggers are being set properly. Even when the requests count was more than 20 in the SQS message queue, the number of app-instances being spawned was limited to 19 by the cloud-watch staying true to the requirements.

2.3.1 Hari Murugan Ravindran

Design

We have extensively discussed how to build this application and have had many sessions conceptualizing how the flow of our application will be. After deciding on the architecture of our image detection application, I went through many tutorials, documents, and blogs regarding the various AWS Services that we are going to use in this application, namely EC2, SQS, CloudWatch, and S3. The major contribution of this project is that I am responsible for the design and implementation of the Web Tier.

Specifically, I was responsible for understanding how the data flows from the Workload generator, which mimics the concurrent user's scenario of uploading the inputs (images), and how it will be processed by the Web Tier, which acts as the controller before pushing it into the request queue. Additionally, the Web Tier is responsible for displaying results to users. The Web Tier is the area where two kinds of processing are done; the input image is pre-processed before being pushed into the request queue, and the output is post-processed after the image recognition is done.

Implementation

- The base tasks, such as creating an EC2 instance which acts as the controller, and setting up the request and response queue with SQS, have been completed.
- Also, key pair and security groups have been set up in EC2, and user groups were created.
- The Web tier has been implemented with the help of Boto3, which is an AWS SDK in Python used for creating and configuring AWS Services.
- Then, the Web tier is implemented using the Fast API Server, where the user's request, which is input images from the workload generator, is converted into a string and uploaded to the queue.
- The Queue Listener continuously listens to the response queue, and if it is updated, it processes it and sends it to the output.

Testing:

There were various phases of testing involved in designing the Web Tier for the application.

1. Unit-testing: To ensure that everything runs as expected, all of the individual components required for the smooth functioning of the Web Tier, such as the workload generator, the Fast API server, pushing the input to the request queue, getting the output from the response queue, and sending output to the users, were tested separately.
2. Integration-testing: The Web Tier is connected with the Workload Generator and the SQS, and the flow of input and output between the Web Tier and the Queues is checked to ensure that no errors occur during the API calls.
3. End-end testing: During this phase, the entire application is tested from the perspective of the Web Tier. This testing involves verifying that the input is sent from the workload generator and reaches the web instance, which acts as the controller, as it is sent to the queue. Additionally, this phase involves ensuring that the output, which is the response from the queue, reaches the users as well.

3. Testing and evaluation

We tested our application against the 100 image test data by sending them as requests concurrently. It took approximately 4-5 minutes to complete processing the requests. We ensured the output of the workload generator and the multi-threaded workload generator are correct. After running the application, looking at the data from S3-output bucket we verified that the output class predicted my ML model matches with the expected output. There were 100 entries in both the input and output S3 bucket which validates that both the request and response handling processing worked correctly. We also checked the status of SQS queue to ensure that messages are being sent and deleted once they are pooled in by the app-tier. We verified that the number of EC2 instances being created by the auto-scaling policy are in-line with the number of requests and less than threshold.

4. Code

We have included 5 programs in the submission file

web_tier.py

This program has 3 responsibilities

1. It is responsible for listening to requests from external clients who upload images for classification
2. It is also responsible for listening to messages on the response queue.
3. Finally, it collects classification output and sends them to the client.

s3_util.py

This program acts as an interface between S3 and other programs. It contains methods required to push objects into S3 buckets. We have also added error handling capabilities since network calls are being used underneath.

sqs_util.py

This program acts as an interface between SQS and other programs. It contains methods required to send, receive and delete messages SQS queues. We have also added error handling capabilities since network calls are being used underneath.

config.py

This is used to define all the required constants for the project like queue names, security credentials, bucket names etc.

app_tier.py

1. This is the most important part of our project. This is where the actual image classification happens.
2. This program listens to the request queue and does classification on images sent by web-tier instance.
3. This program also stores the input image file and output of classification into S3 bucket.

Execution of the code:-

Setting up web-instance

- Launch an instance from ubuntu image available on AWS console
- Login to the instance as user **ubuntu**
- Clone our git repository into instance created using git clone command (git clone https://github.com/hchalla2/Cloud_Project_1.git)
- Change into git project directory (cd Cloud_Project_1)
- Start the fastapi server using the uvicorn command (uvicorn web_tier:app --host 0.0.0.0 --port 8080)

Setting up app-instance manually

- Launch an instance from image created by professor
- Login to the instance as user **ubuntu**
- Clone our git repository into instance created using git clone command (git clone https://github.com/hchalla2/Cloud_Project_1.git)
- Change into git project directory (cd Cloud_Project_1)
- Create a tmp folder in project directory (mkdir tmp)
- Assign all required permissions into tmp folder (cd tmp/)
- Change into home directory (cd /home/ubuntu)
- Start app-instance by running app-tier.py file (python3 Cloud_Project_1/app_tier.py)