CSE 546 - Project Report

Awani Kendurkar 1225438149

Shivani Nandani 1225446014

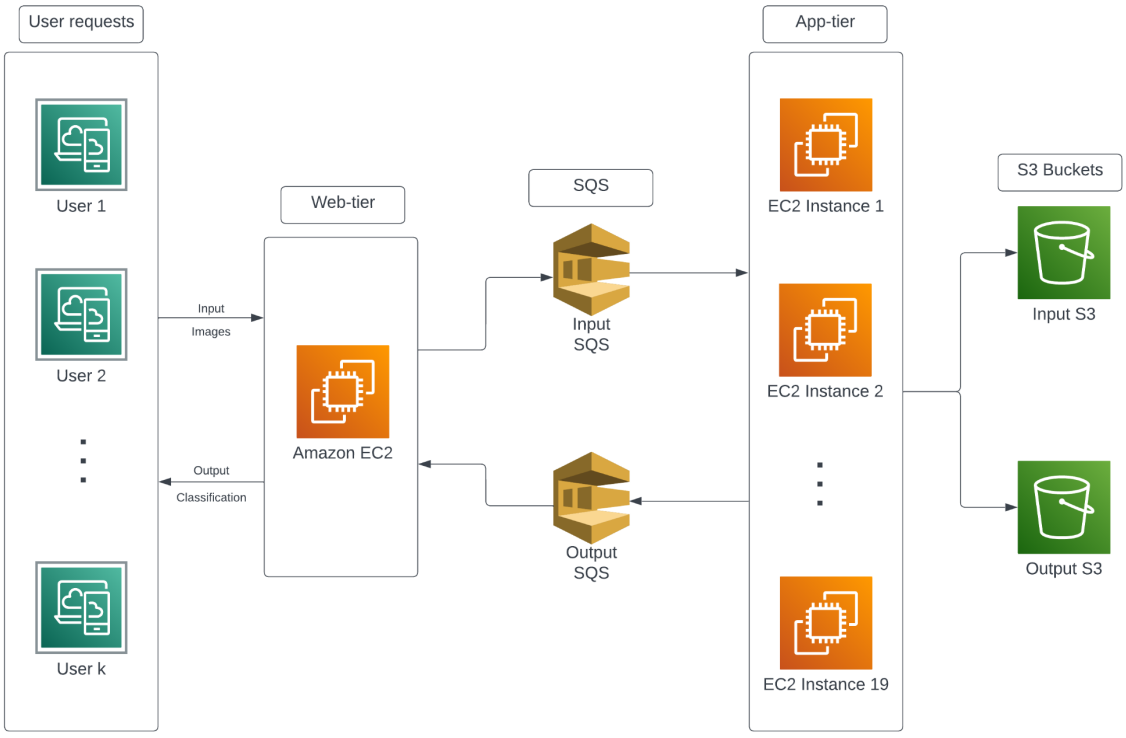
Sidharth Dinesh 1225238352

1. Problem statement

The main goal of this project is to build an elastic image recognition service that utilizes IaaS resources from Amazon Web Services (AWS), provided to the users as a cloud application. The application takes images provided by the users and performs image recognition using a deep learning model that is provided in an AWS image (AMI) to return the predicted output. To ensure our application automatically scales in and out on-demand and cost-effectively, we have used various AWS services for compute (EC2), storage (S3), and message-queuing (SQS).

2. Design and implementation

2.1 Architecture



The architecture of the application encompasses 3 widely used AWS services:

1. EC2: We created virtual machine instances at the web-tier and app-tier layers.
2. S3: For persistently storing our results, images uploaded by the user are stored in the input-bucket and the predicted result (key-value pair) is stored in the output-bucket.
3. SQS: To facilitate message-queuing services that connect the app-tier and web-tier, we use an SQS request queue (input-queue) which stores the requests from the web-tier, and an SQS response queue (output-queue) which stores the messages from the app-tier.

Initially, we have 1 EC2 instance – our web-tier – which will be continuously running and hosting a simple Flask application that exposes an API endpoint. This layer, also known as the controller layer, is responsible for housing the autoscaling logic. A user can send images to this server which are then sent to the SQS input-queue, from where they are forwarded to the app-tier instance(s). The input images are also stored in the S3 input-bucket for persistence. The app-tier instance(s) run the deep learning algorithm on the images and classify them. The classification output is stored in the S3 output-bucket, sent to the output-queue, and lastly, to the user. In the case where dynamic request traffic is to be processed, scaling in or out of our app-tier instance is programmatically handled by our controller in the web-tier (minimum: 1, maximum: 19), as per the number of requests in the queue.

2.2 Autoscaling

The controller.py file, which is hosted in the web-tier-instance, takes care of scaling in and scaling out. To efficiently scale our application, we number of messages in the input queue and the number instances running as metrics. In our specific application, we have determined that each instance can comfortably handle two requests without causing noticeable delays in serving results to users. When the calculated required instances (requests/2) are fewer than the active instances, we scale up to meet demand. Conversely, if the necessary instances exceed the current count, we terminate the surplus. We also consider the scenario where no EC2 instances are required. In this case, we implement a brief one-minute delay before shutting down all instances to ensure no lingering computations.

Since we want a maximum of 20 instances at a time, we get between 1 and 19 app-tier instances to ensure we always have 1 instance for the web-tier running.

2.3 Member Tasks

Awani

1. Created a POST API for the users to send HTTP requests to and designed a simple Flask server.
2. Performed encoding of the images to be sent to the SQS queue.
3. Wrote the logic to facilitate sending messages to the SQS request queue and also listening for the output through the SQS response queue.
4. Conducted testing and ensured code standards.
5. Worked on the report and documentation.

Shivani

1. Wrote functional code to decode in the input images from the request queue and run the image classification model on them.
2. Created the logical flow to upload input images to an S3 bucket and image classification output to another S3 bucket.
3. Also wrote the code to send output messages to the SQS response queue.
4. Conducted testing and ensured code standards.
5. Worked on the report and documentation.

Sidharth

1. Implemented the logic to handle scaling in and scaling out of the EC2 instances.
2. Wrote a shell script for automating the newly created instances.
3. Created the web-tier and app-tier EC2 instances, the input and output SQS queues, and the input and output S3 buckets on AWS console.
4. Conducted testing and ensured code standards.
5. Worked on the report and documentation.

3. Testing and evaluation

We use the multithreaded generator for testing purposes. The Python script queues the 100 images provided in the Imagenet folder in the input queue, which is then accessed by the app tier for processing and classification. To ensure that the program scales and stores as expected, we look at the input and output S3 buckets and monitor the number of instances created and terminated during the testing process. Finally, we also take the output produced by the multithreaded generator and compare it with the truth values produced in the Imagenet folder to ensure that all the images are correctly processed.

4. Code

Web-Tier

- `web.py`: Hosts a server that exposes an API endpoint where users can upload images to be classified.
- `controller.py`: Houses the logic to handle scaling in or out of app-tier instances based on the number of requests present in the SQS input queue.

App-Tier

- `app.py`: Receives the encoded images, decodes them and runs the deep learning model on them to classify them. Stores the input as well the output in the assigned S3 buckets, and sends the output back to the SQS output queue.

Steps to reproduce the code:

1. To spin up the web-tier, run the command `python3 web.py` by accessing the web-tier EC2 instance through SSH.
2. In the same instance but in a different terminal, also run the command `python3 controller.py`.
3. To spin up the first app-tier instance, connect to the EC2 instance through SSH as the `ubuntu` user and run `python3 app-tier/app.py`.
4. Run the following command to get the output on your local machine.

```
python3 multithread_workload_generator.py --num_request 100
--url
"http://ec2-54-152-67-251.compute-1.amazonaws.com:8081/"
--image_folder "./imagenet-100/"
```

Individual Contributions Report for Awani Kendurkar (1225438149)

This report provides details about the contributions made by me towards the development of an elastic cloud-based application that runs a deep learning image classification model on images provided by a user and returns the predicted result. The application is built using cloud services provided by AWS, such as EC2, SQS and S3.

The following are my contributions:

1. Created a POST API for the users to send HTTP requests to and designed a simple Flask server.

   I wrote the code for the Flask application in web.py. This application exposes an API endpoint for users to upload images to be classified. There is a simple route that just returns a string “Application Home Page” and another route which supports two HTTP methods: POST and GET. This application is served on port 8081.

2. Performed encoding of the images to be sent to the SQS queue.

   I also wrote the code to facilitate communication with the SQS queues. I imported boto3 – an AWS SDK for Python which is used to create, configure and manage AWS services – to set up the SQS client. I performed base 64 encoding on the images before sending the file name and the encoded image as a message to the SQS request queue.

3. Wrote the logic to facilitate sending messages to the SQS request queue and also listening for the output through the SQS response queue.

   After sending the appropriate messages to the SQS request queue, I also wrote code to listen for the output messages from the SQS response queue. After processing the output and sending it back to the user as the HTTP response, I also deleted messages from the SQS response queue.

4. Conducted testing and ensured code standards.

   I tested the code thoroughly using the workload generator Python scripts provided by the course instructor. In the event of failure in receiving the correct output, I refactored the code to ensure all the images were being correctly classified and the predicted result was correctly reaching the user.

5. Worked on the report and documentation.

   I created the architecture diagram for this application using Lucid. I also contributed to this report by completing the Design and Implementation section. The README.md file present in the submission zip file was also created by me.