CSE 546 — Project Report

Kaustubh Manoj Harapanahalli, Shravya Suresh, Sreekar Gadasu

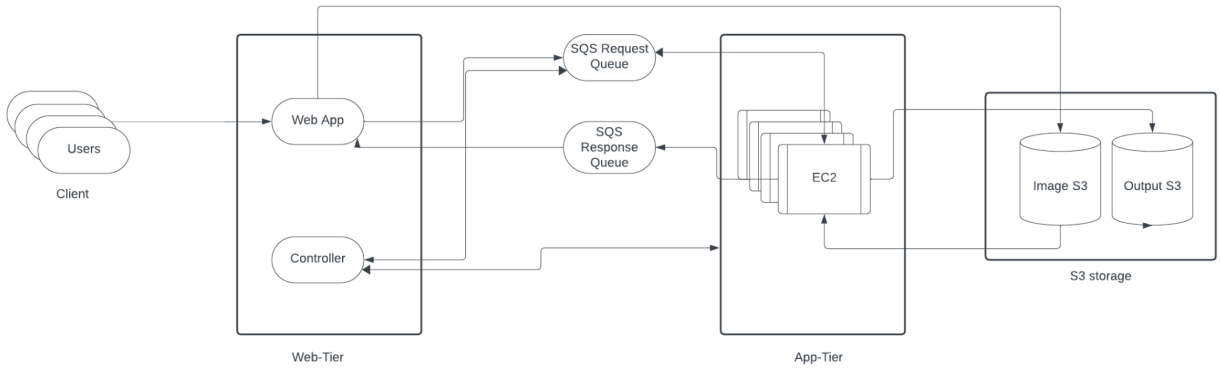
1. Problem statement

The aim of this project is to build an auto-scaling application for image recognition leveraging Amazon Web Services. The project has to be built in such a way that it can scale the computation instances based on user demand automatically. The project should efficiently manage concurrent requests and pipeline the requests using Amazon SQS. It should further perform image recognition on each dynamically created EC2 instance and store the result in an Amazon S3 bucket before returning the result to the user. This project is significant because it addresses the current world needs for on-demand services and how this can be done using the various cloud computing services available. Furthermore, this project helps in understanding various services provided by Amazon Web Services, and how they can be seamlessly integrated with each other to achieve efficient results.

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.



Architecture

Client: In this component, we have users who are sending images and requesting the classification of the image. This process is happening concurrently and continuously. We have a workload generator which is sending parallel requests to the web_app. These requests contain images and are waiting for a response from the web app containing the classification result.

Web-Tier: Web-tier has two components, one is the web application and the other is the Controller.

- Web-Application: The web application is a component of the application which bridges between the client and aws. The web-app is programmed in python language. It accepts image requests from the user. It maintains separate sessions for each user and saves this image in a S3 bucket. The web app also sends the image to the SQS Request Queue, from which the app-tier is pulling the image. Once the computation is complete, the web-app pulls the result from the SQS Response queue and returns the classification result back to the user.

- Controller: The controller component is a part of the web tier of the application. The role of the controller is to perform autoscaling based on the number of messages in the SQS Request Queue. The script automatically increases and decreases the number of ec2 instances and also terminates the instances accordingly.

SQS: SQS also has two components, Request Queue and Response Queue.

The SQS Request Queue is a FIFO Queue. The web tier sends messages to this queue. The message contains information about the session and images to be classified. Furthermore, the messages are then pulled from the queue onto the EC2 instances which are created by the controller.

The SQS Response Queue is also a FIFO Queue. Messages are sent into this queue by the EC2 instances and they carry the classification result of the images. Once the messages are added to this queue, the web-tier pulls the images to the web-app and hence, returns it to the user.

App-Tier: The app tier is the component responsible for image recognition. It pulls the messages from the SQS Request Queue. It pulls the image name from the SQS Request Queue and gets the image from the S3 bucket. It then runs inference on the image by executing the image recognition code. Once we get the result, the EC2 instance then uploads the result back to S3 bucket and then sends the result back to the response queue.

S3-Storage: For this project, we are essentially creating and utilizing two buckets. The first bucket is the input bucket, where all the images sent by the user are saved.

The second bucket is the output bucket, where the classification result of each input image is saved by the EC2 instances.

2.2 Autoscaling

The project description's autoscaling requirement details the need for the cloud application to dynamically modify its resource capacity in response to changing demand levels. The particular project performs autoscaling specifications as follows:

Automatic scaling: The application is able to adjust its resource allocation automatically in response to demand which is based on the number of requests sent by the user to the Request Queue. As a result, it increases compute instances when demand is high and decreases them when demand is low.Basically , we will be creating one instance for every 10 images. Once the instance purpose is completed, and we have more instances than the number of messages, we terminate the required number of instances.

Maximum Instance Limit: The application is only permitted to use a maximum of 20 instances at any given time.

Awaiting Requests in the Queue: When the application's maximum instance limit is reached, all the images are divided among the existing instances.

2.3 Member Tasks

1. Kaustubh Manoj Harapanahalli: Worked on creating the EC2 instance for image recognition and converting it into an AMI on the console. Worked on integrating the different components related to web-app, defining EC2 requirements for auto-scaling, creation of IAM profiles for smoother setup of AWS instances.
2. Shravya Suresh : Worked majorly on the web-tier of the application. Developed the web-app which accepts images from the client and maintains a session for each image. Further, worked on the creation of SQS queues and the queue functions. The web- app is further sending images to the queue. Once the result is available at the Response Queue, worked on extracting the result from the response queue and sending it to the appropriate user who requested for that particular image.
3. Sreekar Gadasu : Worked on the controller and auto scaling functionality of the application. Worked on monitoring the number of messages in the queue and creating EC2 instances based on the number of messages in the queue. Added the necessary code to boot new EC2 instances. Additionally, monitor the number of instances that are running or stopped and depending on the remaining number of messages, these instances are then terminated.

3. Testing and evaluation

We tested the implementation in two ways:

Manual testing of the code: In the manual implementation, we started the flask app, the controller app and ran the app-tier scripts in different terminals.

Automated testing of the code: In the flask app, we implemented a function to create ec2 instances depending on the number of messages available in the SQS request queue and started the EC2 instances using this metric. Based on the number of instances created, we automatically autoscaled the number of EC2 instances. These are some of the scenarios we tested in , and the results received.

Test Case	Expected Result	Test Result
No of messages in Request Queue when 100 requests are sent from user before pulling it to the EC2 instance	100	Pass
No of messages in Request Queue when 100 requests are sent from user after pulling it to the EC2 instance	0	Pass
No of instances when no of requests is 0	0	Pass
No of instances when no of requests is 10	1	Pass
No of instances when no of requests is 100	10	Pass
No of messages in Response Queue when 100 requests are sent from user before pulling it to the Web App	100	Pass
No of messages in Response Queue when 100 requests are sent from user after pulling it to the Web App	0	Pass
Correct classification sent to the user	Sent	Pass
No of images in the Input and Output bucket	100,100	Pass

4. Code

Explain in detail the functionality of every program included in the submission zip file.

Web-application

- webapp/app.py:

  This is the main script that is going to boot the web-server. This file uses flask to run the server. In this file, we have defined the /upload api, which is the API the client uses to send images to the server. The API accepts POST requests which contain images that are to be sent to the server. It further saves the image file in the S3 bucket and sends the image details to the SQS Request

Queue. For each user requesting the classification of an image, we will be creating a new session along with a unique session id. We are then mapping this to the image before sending it across the request queue.

Another important functionality of this script is to keep the session active, keep track of the images sent, and listen to the Response Queue. Once the results are present in the response queue, it pulls the result, and sends the appropriate response to the user. Finally , it deletes the response message from the response queue.

```
create_s3_bucket(
    s3_client=S3_CLIENT,
    bucket_name="quiz-on-friday-input-images",
)
S3_CLIENT.upload_fileobj(
    image, "quiz-on-friday-input-images", image_name
)

# Send a message to the request queue with the session ID
response = create_sqs_request_queue(
    sqs=SQS_CLIENT,
    user_session=user_session,
    image_name=image_name,
)
```

- webapp/Utils/sqs_queue.py:

In this script, we have added the functions to handle the SQS queue. We have two queues, for request and response, and the functionalities are called using this file. We have three main functions. The first function create_sqs_request_queue() is used to create a new fifo queue to carry all the requests from web-application to the aws-application. We then have a function called read_sqs_messages() to read the responses from the response queue and thus extract the classification result. The third function is to delete the message from the queue based on the receipt handle once the message is received.

```
# Function to read messages from the SQS response queue
def read_sqs_messages(sqs):
    queue_url = sqs.get_queue_url(QueueName="ResponseQueue.fifo")
    messages = sqs.receive_message(
        QueueUrl=queue_url["QueueUrl"],
        AttributeNames=["SentTimestamp"],
        MessageAttributeNames=["All"],
        VisibilityTimeout=10,
        MaxNumberOfMessages=10,
        WaitTimeSeconds=20,  # Adjust the wait time as needed
    ).get("Messages", [])

    return messages, queue_url


# Function to delete a specific message from an SQS queue
def delete_sqs_message(sqs, queue_url, receipt_handle):
    sqs.delete_message(QueueUrl=queue_url, ReceiptHandle=receipt_handle)
```

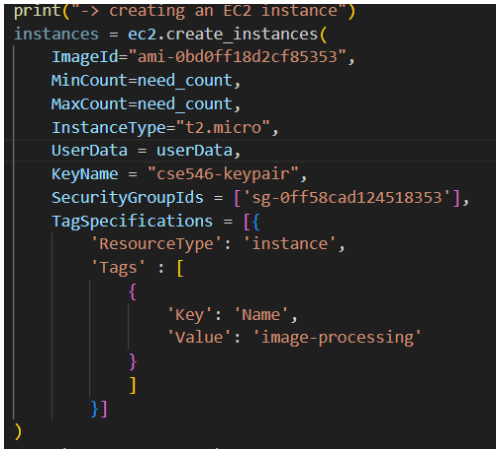
- webapp/Utils/s3_functionalities.py:

This script is used to handle S3 bucket related functionalities. The core function of this script is to create a S3 bucket.

Controller

-script.py:

This is the script which monitors the number of messages in the SQS Request Queue. The script checks the number of messages in the SQS Request Queue. It creates a new EC2 instance for every 10 messages in the queue in order to maintain the load of the instances. Along with this, this also monitors to check if we have more number of instances that are not being used and they will be terminated.The script also makes sure that the number of instances doest exceed 20.



```
print("-> creating an EC2 instance")
instances = ec2.create_instances(
    ImageId="ami-0bd0ff18d2cf85353",
    MinCount=need_count,
    MaxCount=need_count,
    InstanceType="t2.micro",
    UserData = userData,
    KeyName = "cse546-keypair",
    SecurityGroupIds = ['sg-0ff58cad124518353'],
    TagSpecifications = [{
        'ResourceType': 'instance',
        'Tags' : [
            {
                'Key': 'Name',
                'Value': 'image-processing'
            }
            ]
        }]
)
```

App Tier:

- main.py : The provided Python script appears to be a part of an application designed to process images using Amazon Web Services (AWS) services and Python libraries. The script utilizes the Boto3 library to interact with AWS services. It starts by establishing connections to AWS services such as Amazon S3, Amazon SQS (Simple Queue Service), and Amazon EC2 (Elastic Compute Cloud) in the "us-east-1" region.

  The main part of the script is a while loop that runs indefinitely. Within the loop, it retrieves up to 10 messages at a time from an SQS queue using the read_sqs_messages function. Each message is processed, where it extracts information from the message body, such as the image name, user session, and potentially other data. It then performs image recognition on the specified image using the image_recognition function. After processing, the message is deleted from the queue using the delete_sqs_message function.

- utils/image_processing.py : The provided code defines a Python function image_recognition that is responsible for processing and recognizing images. The function appears to be part of a larger application designed to work with images stored in an S3 bucket and perform image recognition using a pre-trained ResNet-18 neural network model from the torchvision library. Here's a breakdown of what the function does:

  The function is designed to be used as part of a larger image processing pipeline, and it can be called to process and recognize images stored in the "quiz-on-friday-input-images" S3 bucket,

saving the results in the "quiz-on-friday-output-images" bucket. The recognized labels and processed images are saved with modified filenames for later use or retrieval.

```python
def image_recognition(image_name: str, s3):
    if not os.path.exists("images"):
        os.mkdir("images")

    s3.download_file(
        "quiz-on-friday-input-images",
        image_name,
        os.path.join("images", image_name),
    )

    img = Image.open(
        os.path.join("images", image_name),
    )

    model = models.resnet18(weights=models.ResNet18_Weights.DEFAULT)

    model.eval()
    img_tensor = transforms.ToTensor()(img).unsqueeze_(0)
    outputs = model(img_tensor)
    _, predicted = torch.max(outputs.data, 1)

    with open("./imagenet-labels.json") as f:
        labels = json.load(f)

    result = labels[np.array(predicted)[0]]
    img_name = image_name.split(".")[0]
    save_name = f"{img_name}##{result}.png"

    create_s3_bucket(
        s3_client=s3,
        bucket_name="quiz-on-friday-output-images",
    )

    img_data = open(os.path.join("images", image_name), "rb")
    s3.upload_fileobj(img_data, "quiz-on-friday-output-images", save_name)

    print(f"{save_name}")

    return image_name, save_name, result
```

- utils/aws_functionalities.py - The provided code defines a set of functions for working with Amazon Web Services (AWS) resources, specifically Amazon Simple Queue Service (SQS) and Amazon S3.

Implementation details:

Create the following setup on the AWS account:

IAM Services

- Create an IAM user group which has the following permissions: AWS EC2 Full Access, AWS S3 Full Access, IAM Full Access. Refer this link for more details on IAM User Group Creation.
- Create an IAM user with the above created user group. Refer this link for more details on how to create IAM user.
- Create an IAM role that will be assigned to EC2 instances created by the web application which has the permissions to read, write and delete in particular S3 buckets. Refer this link how to create an IAM Role for a particular AWS service. For our use-case select EC2 while creating this IAM profile.
- The created IAM user and user group are essential to restrict access to AWS. This user credentials can be shared within the project group without worrying about incurring extra costs.

The created IAM role will restrict access to S3 to whatever resource is essential instead of having complete access to S3.

S3 Services

- Create two buckets - Input Data Bucket and Output Data bucket with their own unique names. The input data bucket contains data that web app fetches and the output data bucket contains the output of the image recognition process.

Once the setup is complete, create an EC2 instance and setup the web-app. The EC2 instance will have the IAM profile linked to it along with two security group CIDR ranges - one for SSH into the instance and other to access the web page through the port 5000.

Similarly, create an AMI with the help of the AMI provided by the professor, and modify the implementations to suit your needs. We implemented the App tier functionalities and saved this EC2 instance as an AMI which we used for automated creation of EC2 instances.

In one terminal we started the web app using the command - python3 web_app/app.py, in another one we started the controller - python3 web_app/script.py and in the last one we started the app-tier using - python3 app_tier/main.py. Once all the sessions were active, we started generating input data using the provided multithread_workload_generator and workload_generator scripts to start and run the application.