CSE 546 — Project 1 Report

Muskan Mehta (1225444701)

Sannidhya Pathania (1225329976)

Vaishnavi Amirapu (1225461211)

Table of contents

1. Problem statement

To build an elastic application that can automatically scale out and in on-demand, cost-effectively, and automatically using IaaS resources from Amazon Web Services. The application should provide an image recognition service to users, using cloud resources to perform deep learning on the images provided. The app should be able to handle multiple SQS requests concurrently. The app should automatically scale using CloudWatch based on the size of the SQS Queue holding the requests. All the inputs (images) and outputs (recognition results) should be stored in S3 for persistence.

2. Design and implementation

2.1 Architecture

Amazon Web services:

- *AWS EC2:*

  EC2 is a compute service provided by AWS to create VMs (Virtual Machines). There are two custom AMIs present in the AWS. One of them is containing the Deep learning model and the other contains the front-end application. So, we have implemented a two-tier application, web-tier (front-end) and app-tier (back-end).

- *AWS SQS:*

  SQS is a messaging service based on a queue data structure provided by AWS for decoupled applications. We have defined two SQS queues, request and response. The request SQS is queuing all the requests made by the workload generator for image processing on the requested images. The response queue will have all the responses generated by the image processing model in the form of (key, value).

  Where key: Name of the image as in S3 (id)

  value: Classification response from the image processing model (classification result)

- *AWS S3:*

  S3 is a storage service provided by AWS and is used to store images that were provided by the workload generator. It is also used to store the results generated by the image processing

model. S3 is providing the persistence for our images and results. We are using 2 buckets, one for input images and one for storing the output results from the image processing model.

- *AWS VPC:*

  VPC is a network service provided by AWS to create networks, subnets, and security groups. For this project, we have used the default network and subnets provided by AWS. We have configured additional security groups for the web-tier and app-tier applications. For the web-tier, we have created a security group to accept connections from All traffic on port HTTP (80) and port 3001 where the web-tier server is running. For the app-tier, we have created 3 security groups to whitelist our system IPs on port 22, to remotely connect to the VMs.

Architecture Design:

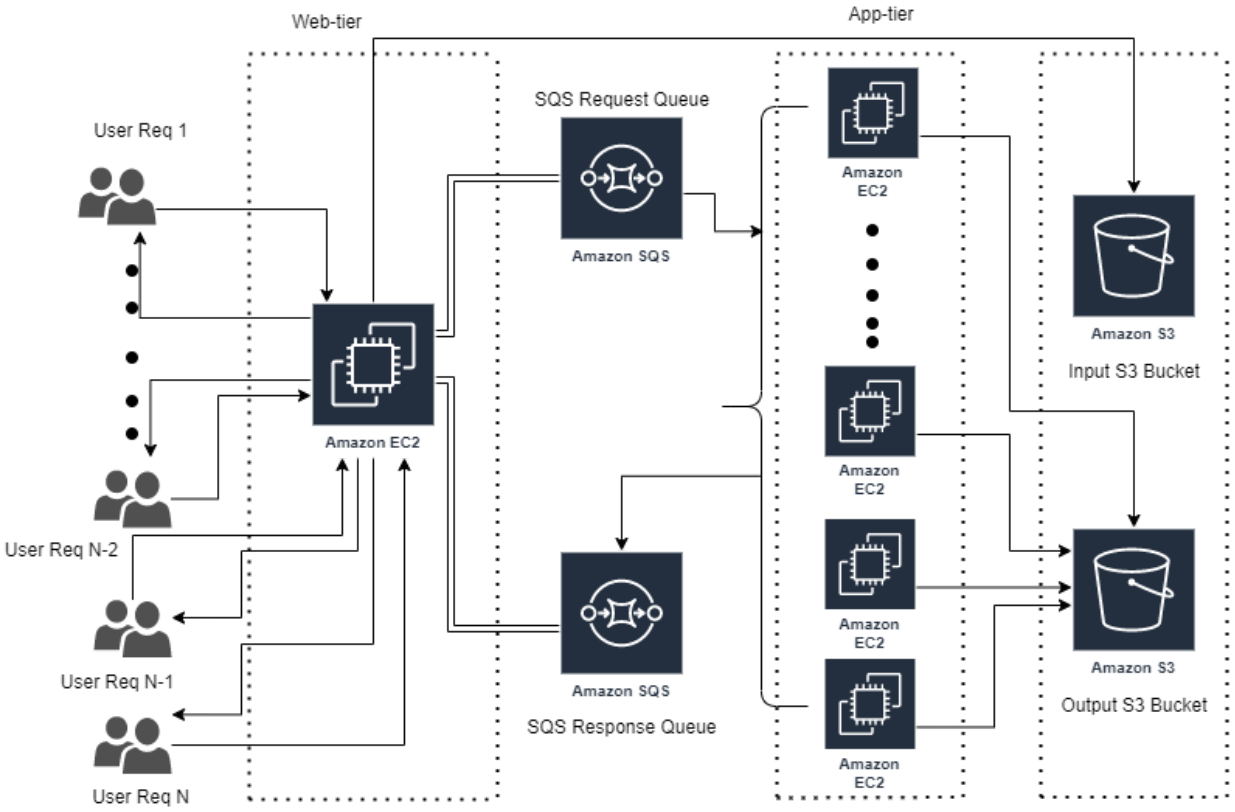

Fig.1 Architecture diagram

The Architecture diagram as seen above consists of EC2 instances, SQS queues, and S3 buckets. The web tier application is deployed on a single EC2 instance. The workload generator sends requests to this web-tier EC2 instance.

The web-tier application is queuing all the requests in request SQS. Concurrently, it is uploading the images to the S3 input bucket. There is a consumer instance listening to the response SQS queue to get the result of the image requested. The consumer instance is long polling the response SQS.

AWS uses the user data in the instance template to launch the app-tier server on EC2 instances. The image will be downloaded from the S3 input bucket and processed to be classified by the established App-tier EC2 instance. The response is a text file, which the App-tier EC2 instance uploads to the Output S3 bucket. The Auto-scaling group (ASG) defined in the AWS will automatically scale up and scale down the App-tier EC2 instances based on the pending request in the request SQS.

The App-tier EC2 instance will pipeline a message in the response SQS queue containing the image name and its classification in JSON after the output has been uploaded to the output S3 bucket. To retrieve the result, the web-tier EC2 instance long polls the response SQS queue using a consumer instance. The user will be displayed the outcome after the JSON response has been parsed.

2.2 Autoscaling

To accomplish the Auto-scaling required, we created an ASG group in AWS. This ASG group is responsible for scaling EC2 instances based on the number of messages queued in request SQS. We are using a Step Scaling Policy, where AWS takes the approximate number of messages available in the queue and subtracts it from the Auto Scaling group's running capacity, which is the number of instances in the in-service state. The output of the expression is backlog per instance i.e *e1* is used to scale up and scale down the instances based on the threshold provided. We set the threshold to 1, which means there will be one message per instance. So, the ASG will try to maintain an average of one message per instance.

The maximum number of instances in our ASG is set to 20, the minimum is set to 1 and the desired capacity is also set to 1. The threshold is set to 1 in the Step Scaling policy. The ASG will use the Step scaling policy to adjust the number of instances, based on a customized metric, *e1* (backlog per instance). ASG will scale the number of instances up or down to keep this *e1* close to the threshold (i.e. 1). We have configured two CloudWatch alarms - one for Scale-in and one for Scale-out. Each alarm contains a set of actions that the ASG must perform in order to change the number of instances.

If *e1* is greater than the threshold, the ASG will scale up the number of instances. The maximum number of instances will be limited by the maximum limit set in ASG.

On the other hand, if *e1* is less than the threshold which is 0 for downscaling, then ASG will scale down the number of instances. The minimum number of instances is also limited by the specified minimum limit of instances in ASG.

In our project, CloudWatch is monitoring the request SQS queue size and the average number of instances in service in the Auto Scaling group. Both these metrics are required to calculate the *e1* to determine when to scale up and when to scale down.

2.3 Member Tasks

We have added the Individual Project contributions under section 5.

3. Testing and evaluation

- The steps mentioned below were followed in order to test our application:
    - The application was tested with the 100 images that were provided as a part of the assignment.
    - We then verified if the S3 input bucket is receiving the correct image and in the desired format.
    - Simultaneously, we checked the status of the input queue to verify the number of messages sent to the SQS queue.
    - Tested the implementation of the Auto Scaling Group by keeping a check on the number of running instances on the EC2 dashboard.
- The following are the results of testing and evaluation:
    - The results generated by the classification are stored in a .txt file in the S3 for persistent storage.

4. Code

4.1 Files

Web-tier-server:

- *server.js*
    - This code will set up the HTTP server, which will listen to POST requests made by the workload generator to classify the image.
    - When an image is received, it first uploads it to the S3 input bucket and then creates a message object (unique ID and the S3 key) for queueing it to SQS for processing.

    - It long polls to listen to the response SQS queue using a consumer instance to fetch the result of the classification. Once the classification result is available, the SQS consumer populates the Result map.
    - It waits asynchronously to retrieve the classification output from the resulting map and sends it back as the HTTP response.
- *package.json*
    - It describes the Node.js project dependencies, metadata, and scripts.

App-tier:

- *aws_config.json*
    - This is a JSON configuration file containing various parameters for an AWS service, such as AccessKeyID and SecretAccessKey for authentication, region for the AWS region, InputS3 and OutputS3 for S3 bucket names, InputS3URL and OutputS3URL for the corresponding S3 URLs, RequestSQS and ResponseSQS for the SQS queues, and AccountID for the AWS account ID.
- *script.py*
    - This is a Python script that reads the AWS configuration from the *aws_config.json* JSON file.
    - It then long polls the request SQS queue to retrieve a message.
    - The same script deletes the message from the request SQS queue.
    - It then downloads the image from the input S3 bucket to the EC2 instance.
    - It uploads the resulting *txt* file to the output S3 bucket.
    - It will then sends the message containing the results of image classification, to the response SQS queue.
- *image_classification.py*
    - Contains an image classification code that was already provided.
- *imagenet-labels.json*
    - JSON file through which the image classification code will provide labels to each image.

4.2 Installation and Running

Web-tier application:

- The application is written in NodeJS. The shell script is in place to install node, npm, and all other package dependencies required for the application. The user needs to run the following command in the folder where the server.js file resides.

  Following is the command to run the web-tier application:

  >> node server.js

App-tier application:

- An AMI image containing the Deep learning model was given as a part of the assignment. We further ramped up the AMI with the required packages and dependencies for our app-tier application to run. Whenever AWS launches an EC2 instance, the user data is already in place in the instance template to start the app-tier application.

  Following is the command to run the app-tier application i.e user data given in the instance template:

  >> python3 script.py

5. Individual contributions

5.1 Muskan Mehta (1225444701)

DESIGN:

- After exploring a couple of choices for the web tier, upon careful evaluation and from my experience with application development, I decided to move forward with NodeJS to accept the images from the workload generator. This is based on two reasons -
    - As per the requirement, we needed a server-side application that could take the requests from the workload generator to process the image.
    - NodeJS is widely used for web application development which has its own HTTP server.
- I started with configuring and developing the application locally on our systems to test various features/tasks of the project.

IMPLEMENTATION:

The following were the implementation performed by me in the given order:

- Implemented a function to accept POST requests on a specified endpoint (/postApi/image) from the workload generator and download it on the local system.
- Implemented a function to upload the locally saved image to the input S3 bucket.

Creation of SQS queue:

- I configured one standard SQS queue for requests which would accept the message from the web-tier which contains the image path in the S3 bucket.
- I added the attributes to long poll the SQS queue in the application code.

Creation of S3 bucket:

- I configured an input S3 bucket that would accept the images from the EC2 instance.

Creation of Auto Scaling Group from app-tier:

- For the ASG, I provided the Launch Template that was created by one of the team members.
- Thereafter provided the Minimum, Maximum, and Desired capacities.
- Also configured the Name tag which would be attached to all the VMs in the group.

Creation of an EC2 instance for web tier application:

- After testing the working of the above code and getting the expected results locally, I deployed the app-tier application on an EC2 instance.

RESULTS:

- The web tier application was able to accept the requests from the workload generator and upload them to S3 and send the message to SQS as expected.
- The configurations performed on the AWS service were also as expected.
- I got to learn the various services provided by AWS and how they can be used and configured to fulfill various project requirements.

5.2 Sannidhya Pathania (1225329976)

DESIGN:

- To set up and deploy the project on AWS, I first created a Free tier AWS account. Once the account was created, I proceeded to set up an IAM group and added all team members as IAM users to the group.
- Next, I evaluated and listed all the services required for the project, and created appropriate IAM policies with corresponding permissions. After creating the policies, I attached them to the previously created IAM group. I also shared necessary credential files privately with the team.
- As part of the setup process, I generated an Access Key and Secret Access ID to facilitate programmatic communication with AWS services.

IMPLEMENTATION:

- For web tier development, I contributed by writing the following functions –
    - Function to send an SQS message to the request SQS queue.
    - Function to long poll the response SQS queue to retrieve the results from the app-tier.
- For app-tier applications, I developed the following functions –
    - Function to long poll the request SQS queue.
    - Function to download the image from the input S3 bucket.
- Creation of response SQS queue –
    - I set up one response SQS queue that will be used by the consumer in the web-tier application to get the result for the requested image.
- Creation of output S3 bucket –
    - I configured an output S3 bucket to accept the output text file from the app-tier application (EC2 instance).
- Creating custom AMIs –
    - For this, I installed the dependencies and downloaded the web tier and app tier codes on created EC2 VMs.
    - Once the end-to-end workflow from image uploading to receiving the classification results worked as expected, I created custom AMIs from the respective VMs.
    - The app-tier AMI was used in the Launch template for the ASG.
- Creating Step Scaling Policy –
    - For this task, I created CloudWatch alarms for scale-in and scale-out based on threshold 0 and 1 for custom metric e1 respectively.
    - I then configured the policy for the ASG by attaching the Alarms created above and adding conditions to scale up or down.

RESULTS:

- The web-tier application was able to send messages to the request queue and long-poll the response queue to display the result. The app-tier application was able to download images from the S3 bucket and long-poll request queue.
- The custom AMIs, step scaling policy, S3 bucket, and SQS queue all were working as expected.
- I learned how IaaS application work in practice and how different components of applications interact with each other. I got to learn some key services that AWS provides and how to configure these services.

5.3 Vaishnavi Amirapu (*1225461211*)

DESIGN:

- I was working on the app tier component of the project and upon thorough evaluation, I decided to move forward with Python for two reasons.
- Firstly, since our backend Deep Learning model is already configured in python it would be easier to even develop the app-tier in the same language to communicate with the model.
- Secondly, AWS boto3, the package provided by AWS makes it very easier to code and communicate with AWS services in python.
- One of the main tasks for the app tier component was to rightly execute the Deep Learning model with the image and to share the results

IMPLEMENTATION:

A series of tasks were performed to develop and test the app tier component of the project. They are as follows:

- Creating an AWS config JSON file that contained all the AWS service configurations and credentials required for the application to communicate with AWS.
- Initially, I configured and created an EC2 instance for the app-tier application from the preconfigured AMI provided to us. This task was done to view and understand the flow of the classification script present in the AMI.
- I then downloaded the classifier scripts locally on my system.
- The app tier component was then developed and tested with the classifier locally on my system before deploying it on AWS.
- For the app tier component, I worked on the following features:
    - To run the Deep Learning model on the image downloaded locally from the output S3 bucket
    - To send the classification results of the image to the response SQS queue
- Once the application was working locally, then I started working on its deployment to the EC2 instance created earlier.
- I created Security Groups for the app tier application to whitelist the system IP addresses of the team members in order to remotely access the EC2 instances.
- Also created Security Groups for the web tier application to receive connections on ports 80 and 3001 from All Traffic (Internet).
- Once the app tier AMI was ready I created a Launch Template for the Auto Scaling Group.
    - This template contains all the required configurations for creating a single VM in ASG
    - Configurations include Instance Type, AMI, User Data to run upon launching the VM, Key Pair, Security Groups, etc.

RESULTS:

- The app tier application of the project was able to long poll the SQS queue, retrieve the image from S3, perform the classification, and send the results to the SQS was working as expected locally and on AWS.
- One of the key takeaways of this project was to learn about different services provided in AWS and how they can be configured and customized to fit the requirements of the project.
- I learned how to configure the VMs to connect with them remotely using KeyPair PEM files.