CSE 546 — Project1 Report

Karthik Aravapalli (1225611998)
Deepak Reddy Nayani (1225418259)
Nikhil Chandra Nirukonda (1223333995)

1. Problem statement

The aim of the project is to build an elastic application that uses AWS cloud resources, specifically IaaS, to run an image recognition service that has the ability to scale-in and scale-out as per demand automatically and efficiently. The user/client effectively send requests for the images to a node.js backend, which is processed, parsed into JSON objects and leverages the storage, computation and transportation provided by AWS services to render the services of the deep-learning model and output the classification results back to the client.

2. Design and implementation

2.1 Architecture

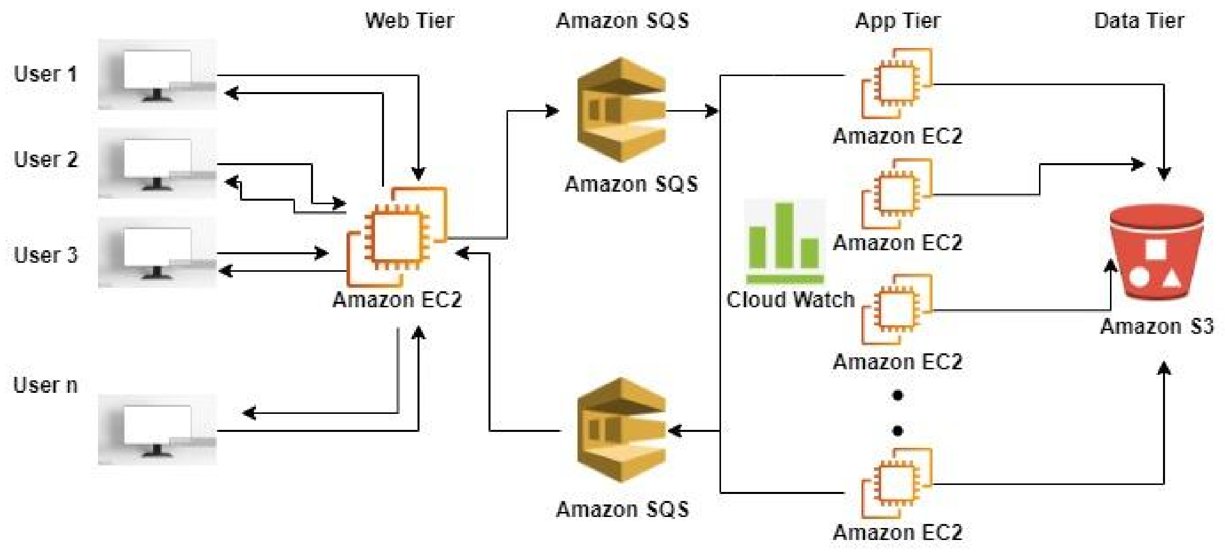

Fig. 1.1 Architecture

Architecture Description

The Architecture in fig 1.1 has one web tier Amazon EC2 instance which will take multiple images as requests at once using the workload generator from the users. These requests are sent into the App Tier EC2 instance using the amazon SQS queues whereas these requests are stored in the S3 Bucket for persistence. Here the Web Tier controls the scaling in and scaling out of requests using a load balancing algorithm that scales out App Tier EC2 instances depending on the demand of requests in the SQS Queue. A maximum of 20(1 Web Tier and 19 App Tier EC2 instances) can be scaled out due to the restrictions placed on the free account usage. A deep learning algorithm is used by the App Tier EC2 instances on the requests from SQS request Queue to predict the output labels for the requested user images. Whereas, these predicted output labels are stored as a (Key, Value) in the S3 Bucket for persistence and the final output sent to the SQS response queue. A dictionary has been created in the Web Tier EC2 instance, so that the outputs from the SQS response queue can be pushed out into the created dictionary and then this dictionary is displayed to the user when he requests for results. The dictionary is formatted after displaying the result which leads to the better performance of the architecture and simplifies our design.

At the start, we only have 1 Web Tier EC2 instance constantly running to accept the user requests and then according to the demand the Load Balancer starts the required number of App Tier EC2 instances. Here, Scaling in is handled by the App Tier EC2 instances themselves as when there are no further new requests at the SQS request queue the App Tier EC2 instance terminates itself.

Our Architecture also handles the fault tolerance from the App Tier EC2 instances. The sequence of steps in the architecture can be described as follows: The Web Tier EC2 instance sends the Image URL to SQS request queue from where it is picked up by the App Tier EC2 instance. Due to a feature of SQS (Visibility timeout), other App Tier EC2 instances cannot read an image when an App Tier EC2 instance is already processing it. Here, the set visibility timeout is enough for the deep learning algorithm to produce the results. So, this feature makes sure that only 1 App Tier EC2 instance is processing the image at a time and if the instance gets terminated then after the visibility timeout a new App Tier EC2 instance starts processing the image and if it produces a result, only then will the image from the SQS queue will be removed. So, we can say that the visibility timeout feature here not only provides fault tolerance, but also contention avoidance. If the number of user requests surpass a certain limit we can create a new Web Tier EC2 instance, but due to the restrictions placed on a free account. Here, we are only running a Web Tier EC2 instance continuously to provide responses to the user requests. SQS request queue also provides another feature called “Receive Message Wait Time” which helps us to limit the number of API requests sent to the SQS. This feature helps SQS to wait for a message to show up before returning an empty response as it makes sense to wait a bit before terminating an App Tier EC2 instance thinking that a new request is available.

2.2 Autoscaling

Through the usage of AWS Service CloudWatch, we have been able to add auto-scale and load balancing properties to our cloud application. CloudWatch is an AWS service that provides monitoring and management services to cloud applications.
For our particular implementation, we use CloudWatch and trigger alarms to monitor the length of the SQS queue and scale in and scale out based on the user traffic. We set the maximum threshold value to 20 instances to abide with the project guidelines (and remain within AWS free

tier usage policy), and use step-scaling to set the number of instances at any given time based on the depth of SQS input queue.

To ensure that our app instance runs the code to start the app tier of our application, while creating the ‘Launch Template’ for our EC2 instance, we set the user data to certain linux commands (i.e, git clone ‘repo’ && python3 app_tier.py), to ensure that on startup, each instance immediately processes the items in the SQS queue and produce the output, thereby making it a lot more efficient that one using only a single machine for all requests.

2.3 Member Tasks

2.3.1 Deepak Reddy Nayani

Design:

I took part in the brainstorming meetings that were held to determine the architecture to be used for our application. We determined which AWS services to use and how to connect them after reviewing various AWS resources.

My most significant contribution to this project was the design and implementation of the app-tier. The app-tier listens to the SQS queue and retrieves messages as soon as they are populated from the web-tier using EC2 instances. After that, the image is written to the S3 bucket before being sent to the Face recognition model for classification. The output result we obtain from the model is written back to the S3-output bucket for persistence. The result is then written to S3, and a message is then issued 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. The JSON objects received on the SQS queue is decoded and the original image is obtained, which is then subsequently run through the image classification model and output of this model is written to the output S3 bucket , which is then again encoded and pushed into the output SQS queue to the web tier which is then outputted to the client.

Testing:

The design of the application's app-tier involves numerous testing phases.

Unit-testing: To ensure appropriate operation, different tests were run on tasks like categorizing the image and receiving messages from the SQS queue.

Integration testing: It was done to make sure that pictures or requests provided by the client were being transmitted from the web tier to SQS and arriving at the app tier after the web tier and cloud watch alarms were configured. As a result, the information being written to the S3 bucket was successfully transferred to the response SQS queue, which is where the web tier is located.

End-to-end testing: During this stage, the complete application was examined to make sure that auto-scaling at the app-tier was functioning properly and that no more than 19 instances were ever created. The S3 bucket's output was compared to what was anticipated to ensure it was accurate.

2.3.2 Nikhil Chandra Nirukonda

Design:

I was heavily involved with the auto-scaling architecture and AWS settings setup that was necessary for the project. In order to use the AWS services, my teammates and I worked together to create the necessary AWS IAM credentials. We have chosen to use EC2 for the application and web layers, SQS for the request message queue, and S3 buckets for input and output storage. I used AWS's Step-scaling policy to perform the autoscaling after looking into a number of alternatives.

Implementation:

The project's auto-scaling phase was crucial for handling several concurrent queries. When there are many requests in the message queue, we had to scale out the EC2 instances with a threshold of 19. When there aren't enough messages in the SQS queue, the program has to automatically terminate App Tier EC2 instances that aren't being used.

I did this by using AWS's Step-scaling policy, which organizes EC2 instances into collections. With a new dynamic scaling strategy, an auto-scaling group is created. There are now two additional scaling-in and scaling-out cloud-watch alerts. The ApproximateNumberOfMessagesVisible SQS statistic served as the basis for the alerts' creation. Using the aforementioned criteria, the scale out alerts set the EC2 instances' capacity between 1 and 19. It is activated when there are more than one messages displayed in the queue. Similar to this, scale-in alerts are set off when there are fewer than 1 message in the queue.

I also Performed Unit, Integration and End-to-end testing along with others.

2.3.3 Karthik Aravapalli

The tasks performed by me during the duration of the project can be classified into the following three:

Design:

We have had numerous sessions to conceptualize how the workflow of our application would be and have had lengthy discussions about how to develop this application. I read a lot of tutorials, publications, and blogs about the various Amazon Services that we would utilize in this application, notably EC2, SQS, CloudWatch, and S3, after settling on the structure of our image detection program. My main contribution to the project is that I'm in charge of designing and implementing the Web Tier.

In particular, I was in charge of comprehending the data flow from the Workload generator, which simulates a concurrent user's scenario of uploading inputs (images), and how the Web Tier will handle it. The Web Tier serves as the controller before pushing the requests into the request queue. The Web Tier is also in charge of showing users the results. At the Web Tier, two types of processing are carried out: pre-processing of the input image before it is added to the request queue and post-processing of the output following image recognition.

Implementation:

The essential tasks, including setting up an EC2 server to serve as the controller and configuring SQS to manage the request and response queue, have been accomplished.

In addition, EC2 has been configured with user groups, key-value pairs, and security groups.

Boto3, an AWS SDK for Python that is used to create and configure AWS Services, was utilized to construct the Web tier.

The user's request, which consists of input images from the workload generator, is then transformed into a string and posted to the queue, where the Web tier is deployed using the Fast API Server.

The response queue is continuously monitored by the queue listener, which analyzes any updates and transmits them to the output.

Testing:

In order to design the Web Tier for the application, testing was done at various stages.

Unit-testing: Each component necessary for the efficient operation of the Web Tier, including the workload generator, the Fast API server, pushing input into the request queue, retrieving output from the response queue, and providing output to the users, was tested separately to make sure everything functions as planned.

Integration-testing: The Workload Generator and SQS are connected to the Web Tier, and the input and output between the Web Tier and the Queues is reviewed to make sure there are no mistakes during the API calls.

End-to-end testing: This stage involves testing the complete application from the Web Tier's point of view. In this test, the input is delivered from the workload generator and is confirmed to have arrived at the web instance, which serves as the queue's controller. This stage also entails making sure that the users receive the output, which is the response from the queue.

3. Testing and evaluation

Initially we tested the web tier implementation multiple times to upload the multiple images at once and even all the 100 pictures at once which are provided by the professor via canvas. Using the workload generator we upload the images to the EC2 and S3 input bucket using the Amazon Web Services SQS queues. Once the SQS messages are generated we check the images are uploaded to the input S3 bucket. For this entire process we implemented the autoscaling to increase the number of instances up to 20 based on the load which we calculated based on the input SQS queue so, once the workload generator runs the web tier will pick those images to try to push EC2 instances using SQS queues so, for this process we implemented this auto scaling part in our cloud watch so, once the SQS loaded with messages then our cloud watch input alarm will create more instances till 20 to process the images to further to store in S3 bucket using app tier. Also, the app tier launch part was kept in the launch template so, the app tier is automatically triggered and images will be sent to image recognition EC2 instances and input S3 bucket. Also, once the image recognition is identified the top results will be sent to the output S3 bucket then, the results are pushed back to the SQS queue then the results will be shown to the user. So, all this activities are tested one by one like the SQS queues are verified in the AWS SQS and the Autoscaling is evaluated in AWS EC2 dashboard where the CloudWatch Auto Scale in doing Scale In and Scale Out and the testing image recognition results in S3 output bucket as well as the terminal will get those results from S3 where we run the Workload generator.

4. Code

Functionality:

1) web_tier.py - This program collects the requests on the port and places them in the input SQS queue as a JSON object for the app tier to process and takes the output from the output SQS queue . We use Quart and boto3 to implement the code, we add async statements to cause the program to wait until it receives a hit on the port, to avoid cases of null data being sent to the app tier.

2) app_tier.py - This program collects the JSON objects in the input SQS queue. It performs multiple functions, it parses the the data of the JSON file to get back the image, it pushes the data into the input S3 bucket, initiates the image classification model with the image and finally collects the output images into the output S3 bucket, parses it into JSON file and pushes them into the output SQS queue.

3) settings.py - This file simply contains the constant variable names to hold the AWS environment variables such as SQS Receive and Response queue name, Input and Output bucket names, Region, Machine instance etc.

4) sqs_util.py - This file is mainly used to invoke helper functions for the web and app tier, its main functionality is to send, receive and delete messages from the SQS queues.

How to run the code:

- Web Instance:

  Launch an ubuntu image on AWS EC2,

log in as “ubuntu”,

clone the repo (https://github.com/deepaknayani22/CSE546_Project1.git),

Cd into folder and run web_tier.py

- App instance

  Similar to that of web tier, only difference is to run app_tier.py.

Individual Contribution Report

Name: Nikhil Chandra Nirukonda
ASU Mail Address: nnirukon@asu.edu
ASU ID: 1223333995

The following tasks that I carried out while working on the project can be used to summarize my contributions to it. Design, Implementation, and Testing make up the primary three areas of the jobs I performed.

Design:

I was highly involved with the auto-scaling architecture and AWS settings setup that was necessary for the project. In order to use the AWS services, my teammates and I worked together to create the necessary AWS IAM credentials after setting up our respective AWS Root User Accounts. We had chosen to use EC2 for the application and web layers, SQS for the request message queue, and S3 buckets for the storage of inputs and outputs. I decided to use the AWS's Step-scaling policy to perform autoscaling after looking into a number of alternatives.

Implementation:

The project's auto-scaling phase was crucial for handling several concurrent queries. When there are many requests in the message queue, we had to scale out the EC2 instances with a threshold of 19. When there aren't enough messages in the SQS queue, the program has to automatically terminate the App Tier EC2 instances that aren't being used. I tackled this by using AWS's Step-scaling policy, which organized the EC2 instances into collections.

I also created two additional scaling-in and scaling-out cloud-watch alerts. I used the statistic The ApproximateNumberOfMessagesVisible to SQS to serve as the basis for the alerts' creation.

Testing:

I have performed various Tests on the system along with Deepak. The tests are as follows:

Unit-testing: To ensure that all operations are being performed efficiently, I ran different tests on tasks like Image Categorization, receiving messages from the SQS queue and writing to the S3.

Integration testing: We performed this testing mainly to make sure that pictures or requests provided by the client were being transmitted from the web tier to SQS and arriving at the app tier after the web tier and cloud watch alarms were configured. As a result, the information being written to the S3 bucket was successfully transferred to the response SQS queue, which is where the web tier was located.

End-to-end testing: Here, we performed testing on the complete application to make sure that auto-scaling at the app-tier was functioning properly and that no more than 19 instances were ever created. The S3 bucket's output was also compared to what was anticipated to ensure it was accurate.

Individual Contribution Report

Name: Karthik Aravapalli
ASU Mail Address: karavapa@asu.edu
ASU ID: 1225611998

My contributions to the project can be described based on the following tasks performed by me during the project. The tasks I performed are divided into mainly three categories: Design, Implementation and Testing.

Design:

We had many sessions to conceptualize how the workflow of our application was supposed to look like and had discussions about how to develop this application. I went through a lot of tutorials, publications, and blogs about the various Amazon Services that we could utilize in this application, and notably EC2, SQS, CloudWatch, and S3, were the better options available of the services, after we had settled on the structure of our image detection program. My main contribution to the project was that I was in charge of designing and implementing the Web Tier of the application.

In particular, I was in charge of comprehending the data flow from the Workload generator, which simulated a concurrent user's scenario of uploading inputs (images), and how the Web Tier will handle it. The Web Tier here serves as the controller before pushing the requests into the request queue, and it is also in charge of showing the results to users. At the Web Tier, I carried out two types of processing namely: pre-processing of the input image before it was added to the request queue and post-processing of the output following image recognition.

Implementation:

I performed essential tasks like setting up an EC2 server to serve as the controller and configuring SQS to manage the request and response queue. In addition, I configured the EC2 with user groups, key-value pairs, and security groups.

To construct the Web Tier, I utilized Boto3, an AWS SDK for Python normally used to create and configure AWS Services.

I transformed the user's request consisting of input images from the workload generator into a string and posted it to the queue, where the Web tier is deployed using the Fast API Server. I also made sure that the response queue was continuously monitored by the queue listener, which analyzed any updates and then transmitted them to the output.

Testing:

I also performed Unit, Integration and End-to-end testing to check if the system was performing as anticipated where I checked whether each individual component required for the operation of Web Tier, components when connected with Web Tier, and the complete application from view point of Web Tier to check if the components and the application was working as imagined at all the stages of the application.