CSE 546 Project I: IaaS Report

Prem Kumar Amanchi
pamanchi@asu.edu
ASUID: 1224421289

Manohar Veeravalli
mveerava@asu.edu
ASUID: 1225551522

Sudhanva Moudgalya
srmoudga@asu.edu
ASUID: 1219654267

1 Problem Statement

The primary objective of this project is to develop a distributed cloud application optimized for efficient large-scale image classification accessible via the HTTP protocol. A key challenge is to implement dynamic auto-scaling to swiftly deploy and deactivate EC2 instances in response to fluctuating workloads, ensuring both high performance and cost-effectiveness. The project adheres closely to specified architectural and interaction guidelines. In the subsequent section, we provide a detailed overview of the high-level architecture.

2 Design and Implementation

2.1 Architecture

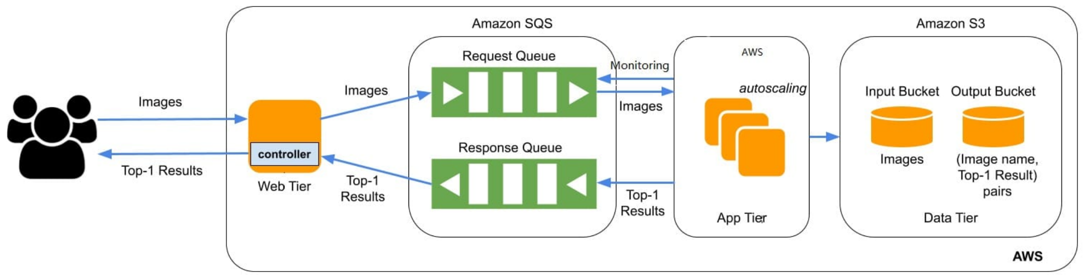

Figure 1: Infrastructure Architecture.

Within our project's architectural framework, we have established a robust infrastructure, shown in Figure 1. The configuration focuses on a single EC2 instance forming the Web Tier. This Web Tier acts as the entry point for user requests in our system. Incoming requests efficiently flow into the SQS request queue, enabling smooth workflow.

Parallel to this, image requests systematically store in an S3 bucket to guarantee data persistence. Additionally, the Web Tier plays a crucial role controlling scaling in/out using a load balancing algorithm. This algorithm bases decisions on demand seen in the SQS queue, which is key for determining the required App Tier EC2 instances ranging from 1 to 19.

The App Tier executes a deep learning model to predict labels for requested images, pulling requests from SQS dynamically. Predicted labels diligently archive in S3 as objects for future reference. Simultaneously, the output dispatches to the SQS response queue for user access. Our design has a unique dictionary in the Web Tier caching output from the response queue. When users request results, the output retrieves instantly from the dictionary, improving efficiency.

For fault tolerance, our architecture exhibits robustness in the App Tier. The Web Tier ensures image URLs consistently store in the request SQS queue, which the App Tier retrieves from. Here, the SQS visibility timeout feature comes into play, restricting other instances accessing an image while one processes it. We meticulously calibrate timeout durations based on the model's maximum processing time.

Therefore, if an App instance unexpectedly terminates, its message becomes visible to others after timeout expiration. If an instance succeeds, the message promptly deletes from the queue, enabling smooth workflow. This timeout mechanism enhances fault tolerance and mitigates contention. Furthermore, our architecture leverages SQS long polling, patiently awaiting message arrival before empty responses. Long polling optimizes performance by conserving resources, adeptly handling request surges.

2.2 AutoScaling

In our scaling-out strategy, we've implemented a nuanced approach designed to effectively manage the number of running instances based on the incoming workload. Specifically, the system follows the following logic: for the first nine incoming messages, a single app instance will efficiently process the workload.

However, when the queue contains ten or more requests, we opt to scale out incrementally. For every additional five messages beyond the initial nine, we initiate the creation of an additional app-tier instance. This streamlined process ensures that our system is exceptionally responsive to user requests, providing rapid and efficient service. This strategy aligns with the overarching goal of optimizing resource allocation and balancing operational costs.

Scaling in operations are efficiently conducted within the App Tier, with app instances thoughtfully managing their lifecycle. Once an app instance successfully executes the deep learning algorithm and generates the desired output, the results are securely stored within the S3 bucket, and the corresponding data is swiftly dispatched to the SQS response queue.

The app instance subsequently enters a monitoring phase, actively scrutinizing the SQS request queue for a brief twenty-second interval. This is made possible through the utilization of the "Wait Time" feature within SQS services, allowing app instances to patiently await new requests. In the absence of any new requests during this timeframe, the app instance will initiate a graceful self-termination process, effectively managing resources and optimizing operational efficiency.

2.3 Member Tasks

1. **Tasks for Prem:**

    (a) Development of Web Tier and Autoscale Logic: Responsible for the development and management of the Web Tier, including `app.js` and `autoscale.js`. These scripts were meticulously designed and fine-tuned to facilitate efficient message handling and dynamic resource allocation based on queue demand.

    (b) Comprehensive Debugging Efforts: Engaged in extensive debugging of the entire system pipeline, addressing technical challenges to ensure seamless application operation.

(c) Architectural Contributions: Actively contributed to the architecture of the App Tier, playing a crucial role in shaping its design and functionality, including informed decisions on system behavior and resource allocation.

(d) Exploration of Automation Techniques: Investigated alternative methods to enhance automation within the App Tier, beyond conventional user data scripts. Evaluated various approaches and technologies to streamline app instance deployment and operation.

2. **Tasks for Manohar:**

    (a) App Tier Configuration: I took charge of setting up the App Tier, configuring it for image classification, and ensuring its readiness to process incoming image requests.

    (b) SQS Integration: I developed the code for establishing a robust connection with Amazon SQS. This code enables the efficient transfer of output messages to the SQS once image processing is complete. This seamless integration ensures the timely communication of classification results.

    (c) S3 Integration: My role included the development of code to interact with Amazon S3, facilitating the smooth transfer of output files to the designated S3 bucket post-image processing. This integration ensures that processed data is safely stored and accessible.

    (d) Utility Tools Development: I actively contributed to the development of utility tools essential for transferring data within the system. These tools are responsible for effectively routing data between various components, including the deep learning classifier, Request SQS, Response SQS, and the S3 storage. This development plays a pivotal role in streamlining the flow of data within the application.

3. **Tasks for Sudhanva:**

    (a) EC2 Web Tier Instance Orchestrated the creation of an EC2 instance dedicated to serving as the web tier. The instance was meticulously configured to ensure seamless handling of HTTP requests and interaction with the project's core components.

    (b) SQS Queues: Played a pivotal role in setting up the project's communication infrastructure by establishing two essential SQS queues: the request queue and the response queue. These queues form the backbone for data transfer between the App Tier and the web server. Their creation and effective management ensured reliable data exchange.

    (c) S3 Buckets: Implemented two distinct S3 buckets, each serving unique roles in the project's data management system. These buckets are fundamental for efficient data storage and retrieval. Their optimization enhances data management processes, a critical component of the project's functionality.

    (d) IAM Roles and Policies: Designed and created specific IAM (Identity and Access Management) roles to enforce rigorous access and interaction control within our AWS environment. These roles were tailored to restrict access to particular resources for specific AWS instances, such as the EC2 instance. Additionally, policies were defined to specify permissible actions for each role, maintaining a high level of control and security in our AWS setup.

3 Testing and Evaluation

To evaluate the performance and functionality of our system, we employed the workload generator provided by the project to generate a substantial volume of requests directed towards the Web Tier. These requests were efficiently processed by the Web Tier, which systematically encoded the images into message bodies and deposited them into the request SQS queue. Subsequently, the Web Tier vigilantly monitored the messages within the request SQS queue, thereby instigating the launch of App Tier instances for the purpose of processing these messages.

In our application, we have configured the Web Tier on a single EC2 instance, streamlining the process of image uploads to the S3 bucket. Our approach involved the creation of an Input bucket within S3, guaranteeing the secure storage of input images from users processed by the App Tier. As a measure to ensure the robustness of our pipeline, we introduced comprehensive logging at various levels, facilitating meticulous tracking and verification of the entire system's operation. Additionally, we established a response SQS queue, which played a pivotal role by accommodating messages added to the queue upon the successful completion of result uploads to the output S3 bucket. This rigorous testing and validation process was crucial in affirming the reliability and effectiveness of our system architecture.

Number of Images Sent (Requests)	**Amount of Time Taken for Processing the Images**
1	1min 8s
10	1min 20s
20	1min 32s
50	1 min 48s
100	2 mins 8 s

Table 1: Number of images processed with respect to time.

4 Code

4.1 Web Tier

The web-tier instance consists of the following files.

- app.js
- autoscale.js

4.1.1 app.js

The app.js performs following operations in the code. This is used for hosting the webserver.

- `uploadFile(base64Image, fileNamePlusIp)`: This function uploads a file to an SQS queue along with image metadata.
- `receiveMessages()`: This asynchronous function receives messages from an SQS queue. It specifies the maximum number of messages to poll from the queue, waits for messages, and processes them.
- `app.post("/", (req, res) => { ... })`: This is an HTTP POST request handler that handles file uploads. It extracts the uploaded file, processes it, and sends it to the SQS queue.

- `app.get("/receive", async function (req, res) { ... })`: This is an HTTP GET request handler that receives the processed result from the SQS queue and sends it as a response to the client.

4.1.2 autoscale.js

- `createInstance()`: This asynchronous function creates a new EC2 instance, configuring it with the specified parameters, such as the Amazon Machine Image (AMI), instance type, key pair, security groups, and user data script. It returns the instance ID for the newly created instance.

- `getQueueMessageCount()`: This asynchronous function gets an estimate of the number of messages in an SQS queue using the `getQueueAttributes` method. It returns the message count in the queue.

- `scaleInScaleOut()`: This asynchronous function implements the scaling strategy. It continuously monitors both the message count in the SQS queue and the number of active EC2 instances. If the message count exceeds the number of running instances, it creates additional instances (up to a predefined maximum) to efficiently handle the incoming workload. It relies on the `createInstance()` and `getQueueMessageCount()` functions to make informed scaling decisions.

- `sleep(milliseconds)`: This utility function introduces a pause (measured in milliseconds) between successive iterations of the scaling logic. It allows for periodic evaluations and adjustments as needed.

4.2 App Tier

- request.js
- response.js
- terminate.js
- app_tier.sh
- app_tier_installation.sh
- terminate.sh

4.2.1 request.js

- `uploadFile(fileName, imageBuffer)`: This function uploads an image to an Amazon S3 bucket. It takes the file name and image data as parameters and uploads the image to the specified S3 bucket.

- `SQS.receiveMessage(sqsParams, function (err, data))`: This function is used to receive messages from an Amazon Simple Queue Service (SQS) queue. It listens for messages in the specified SQS queue using the provided parameters and processes them.

- `uploadFile(imagePlusIp, imageBuffer)`: This function saves the received image data locally and then uploads it to an S3 bucket with the specified image name.

- `SQS.deleteMessage(deleteParams, function (err, data))`: This function deletes a processed message from the SQS queue. It takes the delete parameters and deletes the message once it has been successfully processed.

4.2.2 response.js

- `sendMessage()`: This function sends a message to an SQS queue. It takes the output as input and creates a message with message attributes and a message body before sending it to the specified SQS queue.
- `fs.readFile()`: This function reads the content of a file, specifically the "output.txt" file, which contains the results from a machine learning model.
- `S3.upload()`: This function uploads a file (in this case, the result content) to an Amazon S3 bucket. It takes S3 parameters, including the bucket name, key, and body, and uploads the content.
- `sendMessage()`: This function sends the result content to an SQS queue after uploading it to S3.
- `fs.unlinkSync()`: This line of code deletes the "output.txt" file.
- `SQS.getQueueAttributes()`: This function retrieves the attributes of an SQS queue, specifically the approximate number of messages in the queue. It uses this information to determine whether further actions are required based on the message count.

4.2.3 terminate.js

- `metadata.getMetadataForInstance()`: This method retrieves the metadata for the current EC2 instance, which includes information such as the instance ID. Its purpose is to obtain the instance ID of the running EC2 instance.
- `EC2.terminateInstances()`: This method is employed for terminating EC2 instances. It expects a `params` object that contains the instance IDs of the instances to be terminated. In this particular usage, it terminates the instance associated with the obtained instance ID. The callback function handles any errors that may occur and manages responses from the termination operation.

4.2.4 Shell Scripts

We have three shell scripts. The app_tier_installation.sh is used for installing all the libraries in the instance. The app_tier.sh is used for running the response.js and then run the python model given by the professor. This also runs the response.js file for returning the data to Response queue. The terminate.js is used for terminating the ec2 instance if the request queue is empty.

4.3 SetUp

1. **Clone the Project**
   - Open the terminal in your project directory location.
   - Clone the project repository:

```
git clone https://github.com/PremAmanchi/CSE546-IaaS.git
```

2. **Open the Project in Visual Studio Code**
   - Open the project directory in Visual Studio Code:

     ```
     code .
     ```

3. **Project Directory Structure**
   - Inside the project directory, you'll find three main directories:
     - app-tier
     - web-tier
     - imagenet-100
     - load-generator

4.3.1 Set Up AWS Resources:

1. **S3 Buckets:**
   (a) Create an S3 bucket for input images.
   (b) Create another S3 bucket for output results.
   (c) Note the names of these buckets for configuration.

2. **SQS Queues:**
   (a) Create an SQS queue for request messages.
   (b) Create an SQS queue for response messages.
   (c) Note the URLs of these queues for configuration.

3. **AWS Credentials:**
   (a) Configure AWS credentials with appropriate permissions for your AWS account.
   (b) You can set up your AWS credentials using the AWS Command Line Interface (CLI) or through environment variables.
   (c) Ensure that you have access to the created S3 buckets and SQS queues.

4.3.2 Setting Up the App-Tier AMI

1. **Create an App-Tier AMI**
   (a) **Security Group Setup**
   - Create a security group for the app-tier instances and label it as "app-tier-sg."
   - Configure the security group to allow SSH access only.

   (b) **Launch an EC2 Instance**
   - Create an EC2 instance with the following specifications:

i. AMI: Use the professor-provided AMI.
ii. Security Group: "app-tier-sg."
iii. Key Pair: Use your IAM key pair.

(c) **App-Tier Directory Setup**

- Remove the existing "app-tier" directory in the instance.

(d) **Copy App-Tier Directory to EC2**

- Update all the resource names such as SQS links, S3 bucket names and AWS credentials.
- Securely copy the "app-tier" directory to the EC2 instance:

```
scp -i web-tier-1.pem -r /Users/premkumaramanchi/CODE/PROJECTS/ACADEMIC/CSE546-
IaaS/app-tier ubuntu@<EC2_IP>:/home/ubuntu/
```

(e) **SSH into EC2 and Prepare App-Tier**

- SSH into the EC2 instance.
- Navigate to the "app-tier" directory and provide execution permissions to the scripts:

```
cd app-tier
chmod +x app_tier.sh app_tier_installation.sh terminate.sh
```

- Run the installation script:

```
./app_tier_installation.sh
```

- Create an AMI Image
  - Stop the current EC2 instance and create an Amazon Machine Image (AMI) from the instance.

(f) **Update AMI Image ID in web-tier's app.js**

- Update the AMI image ID in the "app.js" file located in the "web-tier" directory.

4.3.3 Setting Up the Web-Tier

(a) **Configure Web-Tier Security Group**

- Create a security group labeled "web-tier-sg."
- Configure the security group to allow inbound traffic on TCP port 3000.

(b) **Launch a Web-Tier EC2 Instance**

- Create a new EC2 instance named "web-tier" using the "web-tier-sg" security group.
- Update all the resource names such as SQS links, S3 bucket names and AWS credentials.
- Securely copy the "web-tier" directory to the EC2 instance.

(c) **SSH into Web-Tier and Configure**

- SSH into the "web-tier" EC2 instance.
- Navigate to the "web-tier" directory and provide execution permissions to the installation script:

```
cd web-tier
chmod +x web_tier_installations.sh
```

- Run the installation script:

```
./web_tier_installations.sh
```

- Start the ”autoscale.js” and ”app.js” applications:

```
node autoscale.js &
node app.js
```