CSE 546 — Project1 Report

Manan Soni – 1219783877 Nisarg Patel - 1219559757 Sarthak Vats - 1223363388

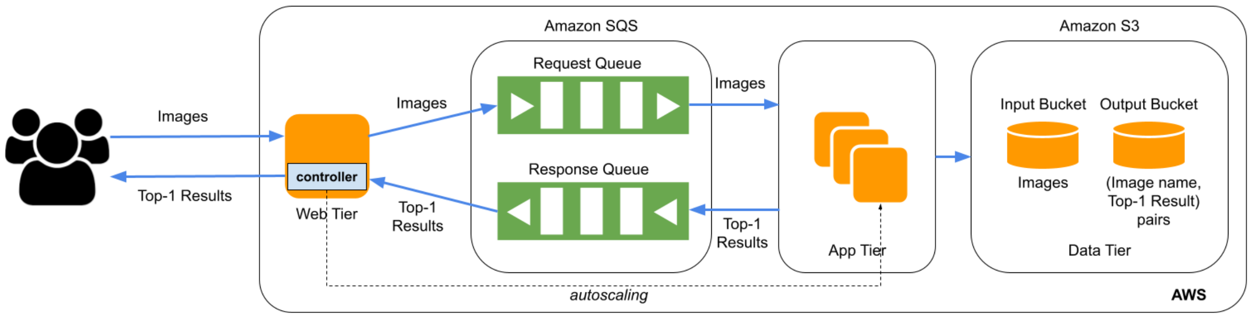
1. Problem statement

The objective of this project is to create an image recognition application using IaaS. The application should scale automatically depending on the number of requests sent to the application by the user. This project is important to understand how the cloud is designed to create elastic applications that automatically scale up and scale down based of the load. IaaS services on AWS are being used in this project to implement auto-scaling.

2. Design and implementation

2.1 Architecture

The architecture diagram for this application is as follows –



∗Image from Project Specification file

The application has the 3 major components –

a. Web Tier – This is the component with which the user interacts with the application. The UI consists of a feature to upload images. The user can select the images and click on Submit button. These images the get store in the input bucket on AWS S3. Also, a message gets put in request queue in AWS SQS for processing the image further using the app - tier.
b. Controller – The controller is being used to scale up or scale down the app tier instances based on the number of request present in the input queue.
c. App tier – This is the component which consumes message from the input queue and processes it using the image recognition model provided. The output of the model is stored in output bucket and is also passed to the response queue which relays it to the web tier which it is shown to the user.

2.2 Autoscaling

Scaling up and down was accomplished by establishing new app instances and employing multi-threading. The app instances are started automatically by the web tier, and the app tier handles multi-threading.

The web tier application keeps track of the number of requests in the input queue as well as the number of active app instances. When the number of input queue requests is smaller than the number of app instances, the web tier immediately starts the app instances in response to the demand. If the input queue contains five requests and only one app instance is running, the web tier will immediately launch four more instances to process the input queue requests.

For handling the increasing demand of requests in the app tier, the program primarily relies on multithreading. When the number of input requests in the input queue reaches 19, the app instance divides the total number of requests by the number of app instances operating and creates the requisite number of threads. If the SQS has 100 requests and 19 app instances are already operating, each app instance will divide the number of visible messages by 19 and launch the needed number of threads.

When there aren't enough messages for the app instances to consume, the app instance terminates. The scaling down of app instances is handled in this way.

3. Testing and evaluation

1) Images uploaded by a user saved in S3 bucket: - The images were visible in the AWS S3 bucket.
2) User request sent to app tier using SQS input queue: - The requests were visible in the input queue.
3) App tier running EC2 instances: - The app tier was able to access images from the S3 bucket and run the required number of EC2 instances.
4) Scaling of EC2 instances: - The EC2 instances scaled inwards and outwards perfectly. When the number of images uploaded was less than 19, only that number of instances were running. When the number of uploaded images increased to more than 19, all the instances were running and had multiple images to process.
5) Output of image classification stored in S3 bucket: - The output of the image classification was visible in the S3 output bucket.
6) Output sent to web tier using SQS output queue: - The output of the image classification was sent to the web tier using the SQS output queue.
7) Displaying the result on the front end: - The results of all the image classification queries were displayed on the front end.

4. Code

1) WebappApplication.java:- It is the web tier file. It initializes all the objects.
2) AppConfig.java:- The configuration class that produces beans necessary for the web tier to run. All these beans are handled by IoC controller and configured when the application starts, making the application loosely coupled as it injects dependency automatically. As Spring beans can be reusable these objects are created only once during the start of the application where our Spring IoC is initialized and configured for all the beans.
3) BasicAWSConfigurations.java:- This file will configure all the AWS services we are going to be using in this project.
4) ProjectConstant.java:- All the constants used in the project are in this file. This helps us keep the code clean and easy to maintain.

5) WebAppAPIController.java:- Includes all the APIs required to run the application.
6) AWSS3Repo.java:- Contains methods to upload files to AWS S3 input bucket.
7) AWSS3RepoImpl.java:- Contains the implementation of uploading images to AWS S3 input bucket.
8) AWSUtil.java:- Contains methods to create and run instances.
9) AWSUtilImpl.java:- Implements methods to create and run instances.
10) BusinessLogic.java:- Contains methods to run the entire project including displaying the output on the frontend.
11) BusinessLogicImpl.java:- Implements the methods in BusinessLogic.java. Its main focus is the structure of the project.
12) bootstrap.css:- Contains the CSS information for responsive the frontend.
13) style.css:- Contains the CSS information for the presentation of data on the frontend.
14) imagerecognization.jsp:- Webpage where the user uploads the images.
15) result.jsp:- Webpage where the results are displayed.
16) pom.xml:- It is an XML file that contains information about the project and configuration details used by Maven to build the project.

Installing and Running the program:-
1) Download the project from our private git repository using the command:-
git clone https://github.com/nisargpatel58/AWS_IAAS_Image_Recognisation
2) Import it as a maven project in your workspace.
3) Run maven clean.
4) Add dependencies from pom.xml file.
5) Update ACCESS_ID, ACCESS_KEY according to your setup.
6) Run Maven-install again.
7) Make use of the given AMI to create an EC2 instance.
8) App-tier:- Import the app tier jar file on this EC2 instance using the command:-
```
scp -i security_key.pem /path_of_app_tier_jar ubuntu@ec2-public-dns:~
```
9) Now create a new AMI.
10) Use this new AMI ID and paste it as a constant in the ProjectConstant.java file.
11) Run maven install again to update web-tier jar.
12) Transfer this jar file to the EC2 instance using command:-
```
scp -i security_key.pem /path_of_web_tier_jar ubuntu@ec2-public-dns:~
```
13) Login to web tier instance using the command:-
```
ssh -i security_key.pem ec2-user@ec2-public-dns:~
```
14) setup java environment in the instance
```
yum list java*
sudo yum install java-1.8.0
```
15) change the permissions of the jar
```
chmod +x <jarname>
```
16) run the web jar file
```
java -jar <path to jar file>
```

5. Individual contributions (optional)

Nisarg Patel –

a. Design

The whole architecture was actively discussed by all the team members. There were a number of small features that we had to implement so as to get the application running. I have majorly contributed to two of the features that were thoroughly used throughout the project. They are autoscaling, creating and stopping EC2 instances.

b. Implementation

There were a few challenges that had to be handled while designing and implementing the auto scaling module. Firstly, the app instances were to be scaled till the maximum count of 19. As the free tier can allow a maximum of 20 EC2 instances at a time, I had to make sure to design the autoscaling with that parameter. Secondly, they were to be scaled up and scaled down on demand. When a user sends 10 images, only 10 EC2 app instances have to run at a time. While, when a user uploads 100 images, it can only run 19 Application instances and run concurrently to provide the output. I have implemented the code for auto scaling the app instance from the web instance using multithreading. The reason for using thread to perform autoscaling is because it is an application controller to handle load and we have implemented a controller in the web tier instance itself in order to modularize our code. Another reason is we have used Spring boot for our project which has a Tomcat server, and Tomcat can serve upto 10000 HTTP requests parallelly. And sometimes while serving requests it can interfere with our main thread of the web tier thus decreasing the performance of the application. So to resolve this I have kept a separate thread that runs the controller without interfering with the main thread and Tomcat can smoothly handle all the requests from concurrent users. I have also implemented the utilities related to the EC2 instances such as configuration,creation, and stopping of the EC2 instances. I have also given the idea to my team member to format the input request in order to serve concurrent users requests. My team members have nailed it and now we were able to serve concurrent user requests correctly and fastly.

c. Testing

In the testing phase, I have tested the modules that I worked on individually. I have also been involved in regression testing of the modules I implemented and integration testing of the whole application. I tested the autoscaling with various numbers of images ranging from 1 to 200 images, where I was checking if the autoscaling is working properly. Here, the time wasn’t considered as a parameter for testing.