CSE 546 — Project Report

VENKAT TEJA GADDAM (1219275759)
HEMANTH KUMAR SINDE (1219489830)
JITENDRA PRASANNA GOTTU (1220802908)

1. Problem statement

Title: Image Recognition as a Service

Motivation: To build an effective, efficient, scalable and reliable cloud application which caters Image Recognition as a service to the user with the help of AWS Cloud Resources. In this application, the input is given by the user in form of images where our application recognizes the images by performing deep learning methods.

At Present day in some of the cloud applications there are some issues pertaining to Cost effectiveness and efficiency in handling traffic demand. Our goal is to build and deploy a cloud application which will effectively handle the aforementioned issues while simultaneously taking care of improvement with fault tolerance.

Functionalities:

- Concurrent Requests will be handled by the cloud Application.
- Traffic from the Users will be regulated by the Cloud Application. In other words, the cloud application does automatic scaling out when demand plunges and automatic scaling in when demand surges.
- The cloud Application accurately acknowledges all the requests from the user without missing out any requests.
- The application return results to the users in best possible time.

Technologies: Elastic Compute Cloud (EC2), Simple Queue Service (SQS), Simple Storage Service (S3)

Type of Usage:

Elastic Compute Cloud (EC2): One machine as hosting server and remaining 19 for computing.

Simple Queue Service (SQS): Standard queue as a load balancer.

Simple Storage Service (S3): As a storing service for input and output.

2. **Design and implementation**

2.1. Architecture

Application Design:

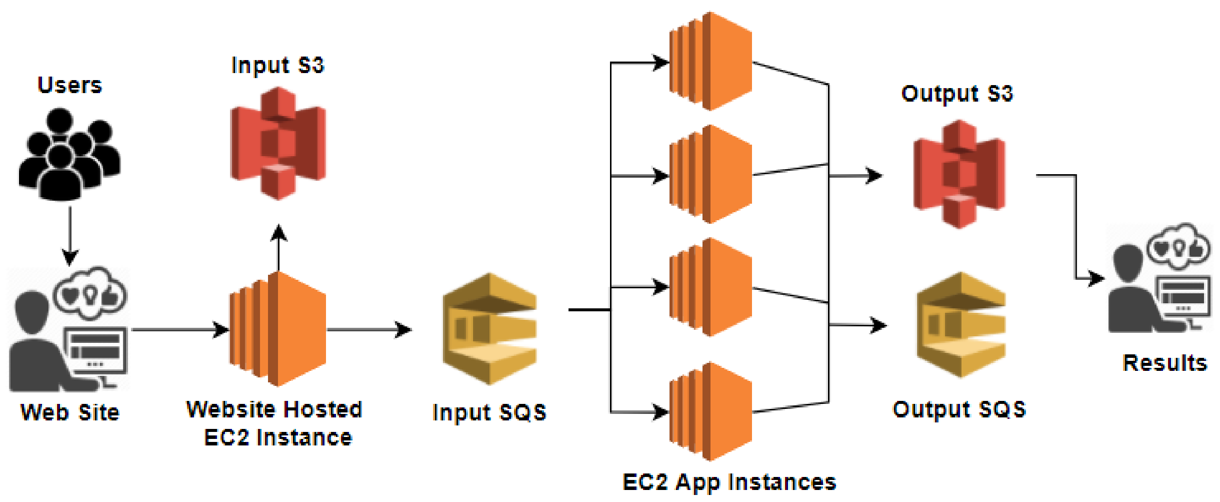
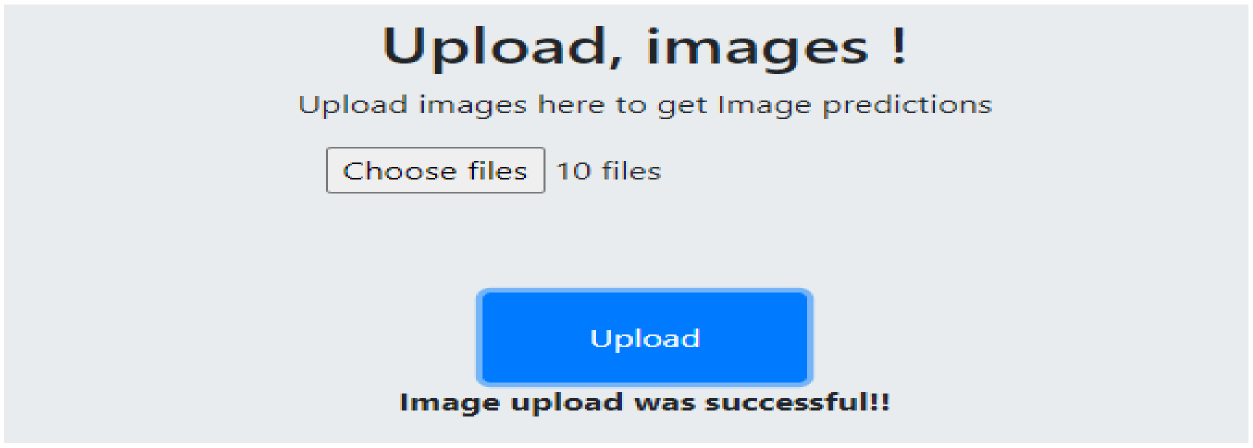


figure 1

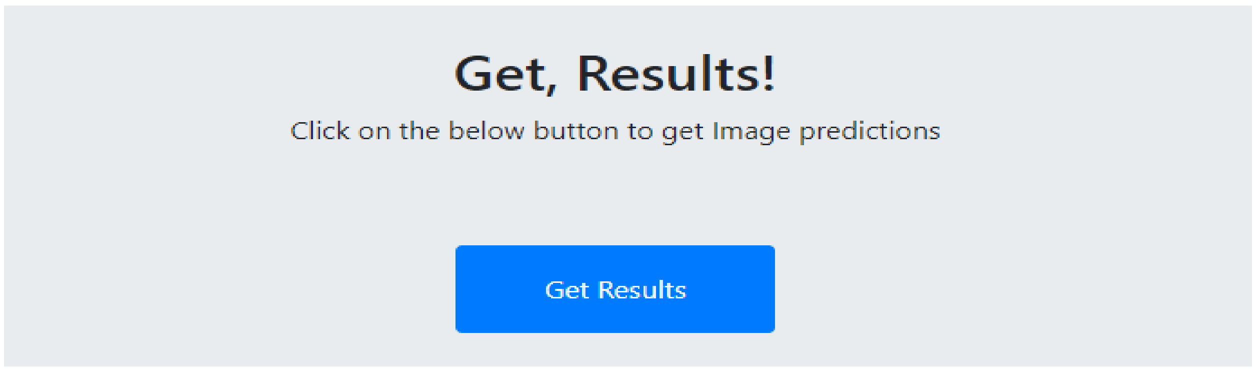
Website Design



Upload, images !

Upload images here to get Image predictions

Image upload was successful!!



Get, Results!

Click on the below button to get Image predictions

Detailed Architecture:

Our Cloud Application is comprised of two main components one is Web Tier and the other is Application Tier. Input from the user is taken and is stored by web tier whereas app tier takes the images stored by web tier as input with the help of a queue and performs deep learning then gives the output in a storage bucket.

In brief, our architecture employs an EC2 Web tier instance (1 instance) that takes the requests from users in the system. These requested image names will be pipelined into an input SQS which will be fed into application tier EC2 instances (19 max,1 min) in later stage. In parallel to storing image names in input SQS, we store images in input S3. Application instances will be scaled in and out through the web tier instance. Load balancing algorithm will be used to decide whether the additional application tier instances will be required. We can have a maximum of 19 application tier instances for scaling out. Application tier instances will run deep learning application. They take image names from Input SQS and get image using those names from Input S3. App Tiers produce predicted output label for the images. These requests and its subsequent results will be stored on a S3 storage for persistency. Initially we have 1 web tier and 1 App instance running to handle quick response. Web tier takes care of starting the other App instances as per demand. Scaling in is handled by App instances themselves as they auto terminate when they do not find messages in SQS. Once the output has been obtained, they will be fed into another output SQS queue which would push these results to the EC2 web instance. We actually pushing results from output S3 only.

AWS EC2: Amazon Elastic Compute Cloud is a web service. It is used in creating virtual machines where user can access them and run multivariate applications & programs in a secure, robust and reliable environment. In our cloud application, we employed EC2 in Web Tier as well as Application Tier.

AWS S3: Amazon Simple Storage Service is an object storage service which is scalable and reliable. By using AWS S3 Bucket we store any type of objects. In this Project We have Employed S3 at Application Tier level and Web Tier Level as Well. S3 receives Input from EC2 Instance of the Web Tier and then the input is transferred to EC2 instances of application tier with help of a queue and then the output is stored at s3 Bucket which is at application Tier.

AWS SQS: Amazon Simple Queue Service (SQS) is a fully managed message queuing service which helps in movement of data to and from between applications or micro-services by hosting a highly scalable queue for managing the data. To move data from S3 to EC2 instances in application tier and for scaling the data we use SQS.

Framework: We have used JAVA Spring Boot framework for our REST application at web tier and App Tier. We have shell scripts for running our logic for auto-start and termination of App Instances.

For front end, we used React js. This is used as interface for loading images.

2.2 AutoScaling

Scaling in and Scaling out of the Instances are determined by Application Tier and Web Tier respectively. When the web tier instance gets a new request, it puts it into the input SQS as shown in figure 1.1. Most difficult part for the AutoScaling was to create the environment which is thread-safe. As in our web application there can be concurrent request served by different threads and they individually cannot do auto scaling as they don't share memory, so keeping shared variable who keeps track of running instance is not possible The different threads available in our application can't perform AutoScaling individually because the threads do not share the memory and at the same time there could be multiple simultaneous requests handled by different threads in our application. In order to overcome this, we use a technique called multithreading where one thread is used for processing, other threads are used for scaling and for result fetching.

First of all, we tried program our application in such a way that the application won't invoke more than 20 instances at any given time though it has some issues. The AutoScaling mechanism constantly checks for messages in AWS SQS, whenever it encounters a message the inception of an EC2 instance takes place and then the input goes through the deep learning process to get the output. The output Is stored into the S3 Bucket. AutoScaling mechanism looks again for a message and does the whole process again and make sure that at any point of time not more than 20 EC2 instances are running. If the AutoScaling mechanism, doesn't find any message in SQS, it terminates itself by coming out of the loop which results in scaling of the instances.

The aim of our logic is to match the number of running instances and the number of visible SQS messages, if we can do so in the given maximum 20 app instances. If number of SQS messages is higher than the 20, then we are not going to match with the number of the instances, but we will create the instances so that overall 20 app instances will be running. This is all about the scale out logic that we implemented in the Web Tier. We can say that it is almost like the greedy solution to provide the service to users as fast as possible. Here in our custom AMI which we are using to create the app instances have the capability of running the java application using shell script every time it boots up. Then app instance will start doing the work that it needs to do. Scaling in works at the application tier. After an application tier instance predicts the output a given request, it feeds this output to the SQS output and put the key-value object pair in the S3 bucket. After completing the one request that was in SQS, it again looks for the message in the SQS. If it gets the message, then it will do whole deep learning process again. If there are no messages in the queue it instantly comes out of the loop and terminates itself, which scales in the instances. We can also use one of the SQS services which is the "Wait time". If we give wait time as 20 seconds then receive message request from app instance will wait for 20 seconds looking for the messages in the SQS in single request, but as per the project instructions we kept it 0 wait time. So, in a nutshell, app instances will run in loop till they are able to find some request in the SQS, and if not, then it automatically terminate itself or overall scale in our whole app instances setup.

3. Testing and evaluation

Testing: The User Interface receives input from the multiple users and we move on to the evaluation phase. The technique we have employed in testing phase is called Manual testing.

We make sure all the requests are processed and loaded into input SQS and input S3.

Evaluation: In this phase we made the application to check each and every request along with the instance count. We made sure that auto scaling function works well when it comes to instances creation with respect to the number of requests from the users.

4. Code

4.1 Code Modules

a) Web-Tier module

a.1) This module is executed by the Web Tier instance.

a.2) Rest Controller (WebTierController.java): This controller maps the image request to the function `uploadFiles` present in ImageService.

a.3) Send message to the input queue and input S3. This service uses AWS APIs to place the message in the input queue and S3.

a.4) `getResults:` This module retrieves results from output S3 bucket.

b) ListenerAndDispatchingService module

b.1) This module is executed by the individual App instances.

b.2) It will receive the message from the input queue and if the message is not available then the App instance will terminate itself. If it receives a message from the input SQS queue, first it will load that image from input S3 bucket and will provide the message (i.e. image) to the deep learning module for classification. A process builder will execute the deep learning computation instructions.

b.3) The classification result will be written to the S3 storage in bucket using AWS S3 APIs.

b.4) The classification result will also be sent to the output SQS queue. After that message is deleted from the input SQS queue.

c) LoadBalService module

c.1) This is a module running in one of the threads of web-tier application.

c.2) This module queries input SQS queue for the approximate number of messages.

c.3) It also queries EC2 for the number of running instances at a given point in time.

c.4) Based on the above two query results, appropriate number of instances will be spawned by this LoadBalService. This ensures proper scale out of the App instances. Scaling in is taken care by the individual App instances.

4.2 Instructions for Project Execution

4.2.1 Setup

a) Setup of Application Tier-EC2

a.1) First we created one app instance using professor provided AMI. (ami-0ee8cf7b8a34448a6)

a.2) Create a jar file of AppTierScaleIn submitted in blackboard in your local machine using eclipse maven build.

a.3) Now transfer this AppTierScaleIn jar to app ec2 instance which was created earlier and give file permission (chmod +x jar name) .(Note: Java environment is already available in instance)

scp command: scp -i security_key.pem /path_of_AppTierScaleIn _jar ubuntu@ec2-public_dns:~

a.4) Create a bash file(AutoStartScaleIn.sh) in app tier instance and move it to boot location. This bash file should have commands to run the AppTierScaleIn jar we put in the same instance.

a.5) Create AMI of this app tier ec2 machine. This will be used in web tier.

a.6) After the AMI is created with snapshot, we can terminate the app instance.

a.7) Now, create an AMI image of the listener application EC2 instance. This will create one snapshot of the same size as ec2 app instance with the required environment. Now this will be the base AMI for all other Listener App EC2 instances later (we are using this created AMI for 18 App instances in our code). Note: If you want to change credentials (such as access key, you should change the values of those credentials in constants.java in the code module)

b). Setup of Web-tier-EC2 (create only one web-tier instance)

b.1) Created one web instance using professor provided AMI. (ami-0ee8cf7b8a34448a6)

b.2) Create a jar file of web-tier application submitted in blackboard in your local machine using maven build.

b.3) Now transfer this web-tier jar to web-tier ec2 instance which was created earlier and give file permission (chmod +x jar name).

scp -i security_key.pem /path_of_webtier_jar ubuntu@ec2-public_dns:~

b.4). Setup a react js application in Web tier itself for users to upload images.(port:3000)

4.2.2 Execution Steps

a.1) First we need to run react front end application in web tier

```
nohup npm start >frontend.log &
```

a.2) Then we need to start web-tier application so that it can respond to http requests from user. Web tier can be started automatically or manually. To see the debug log, we do it manually as:

```
Java –jar WebTier jar_file_name
```

Now the web tier server is up and ready to use.

Users can upload images and results will be shown.