ClassifyCloud – Image classification as a service

Gurupad Hegde Mahabaleshwar
CISE Department
University of Florida
Gainesville, Florida
gmahabaleshwar@ufl.edu

Sourabh Gopal Parvatikar
CISE Department
University of Florida
Gainesville, Florida
sourabh.gopalpar@ufl.edu

Abstract—**Deep Learning has emerged as a new area in machine learning and is applied to a number of image applications. The main purpose of this project is to provide a cloud service to classify images. We have used the JAVA APIs provided by Amazon Web Services(AWS) for this purpose. The major components include a load balancer which executes an autoscaling logic, a spring boot web server which accepts the user requests and a worker module which creates and terminates instances as required and predicts the class for the given image url. Additionally we have request, response and terminate queues to store request, response and terminate messages respectively. We have used the pre-trained Inception model as a deep learning image classifier and deployed it on our cloud service.**

Keywords—Deep learning, Aws, load balancer, auto scaling, scaling criteria, worker, request queue, response queue, terminate queue, SQS, inception, worker pool, EC2, S3, Spring boot, Java, Python

I. Introduction

Image classification, which can be defined as the task of categorizing images into one of several predefined classes, is a fundamental problem in computer vision[4]. Dual stage approach was being used for this purpose, where the first stage was feature extraction and the second one was training. The quality of the model largely depended on the feature extraction stage which was a highly difficult task. Deep learning eliminates the feature extraction stage as it directly extracts features from provided training images. AWS provides all the required infrastructure for developing ClassifyCloud. We have used AWS EC2 instances to deploy the image classification model and classify the input image. We have also used AWS Simple Queuing Service(SQS) queues to store requests, responses and the terminate messages. Finally, we store the image url and the result in AWS simple storage service(S3) for future reference[5]. In our project, we have used the pre-trained inception model to classify images[3]. The primary goal of this project, ClassifyCloud, is to provide a cloud service to identify the object in a given image as accurately as possible. We have created our own load balancer which scales up or scales down the workers based on the scaling criteria. Scaling criteria is a number which determines the number of workers to be created or terminated. This is calculated using a formula which resulted from various experiments. The major focus of the project is to provide an optimal cloud service to classify images using a deployed deep learning image classification model. We have used Spring boot for developing client web server which provides an interface for the users to interact with the ClassifyCloud[6]. Spring boot framework provides good environment which meets all our requirements. Users can provide an image url through GET request to get the class of the object in the image.

II. Related Work

There are many image recognition/classification services[1]. Google Cloud Vision is Google's visual recognition API using Tensorflow and REST API. It contains comprehensive set of labels and detects objects and faces in an image. It includes Optical Character Recognition(OCR) and is integrated with google image search to find similar images from the web. IBM Watson Visual Recognition is a cloud service developed by IBM. Though it has a comprehensive set of built in clasees, it is majorly built for training custom classes based on the images provided by the user. This also includes OCR like Google Cloud Vision and NSFW detection. Clarifai, unlike other image recognition APIs, has additional feature of video analysis. It offers scene recognition for videos. It also provides sentiment analysis, logo detection, text recognition and face detection for images as well as image attribute detection such as color, dominant color and brightness[2]. Our work is closely related to Amazon Rekognition, the image recognition API provided by Amazon which integrates with other AWS services. Microsot Computer Vision API is similar to Google Cloud Vision and IBM Watson, with the additional feature of celebrity detection. It also generates natural language description in addition to confidence predictions. CloudSight is slightly different to other services described. Though their website has fewer details compared to others, they provide some combination of algorithmic and human tagging.

Convolutional neural networks started in 1990s and has grown exponentially in last few years. At its core, CNN is the simulation of how a human brain works. It is a class of deep neural networks, mainly used to analyze images.

III. System Architecture

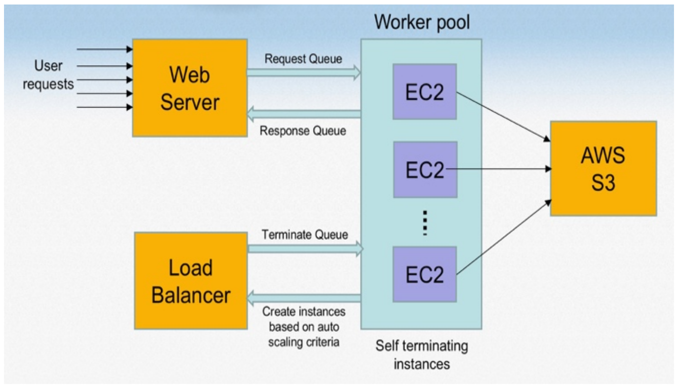


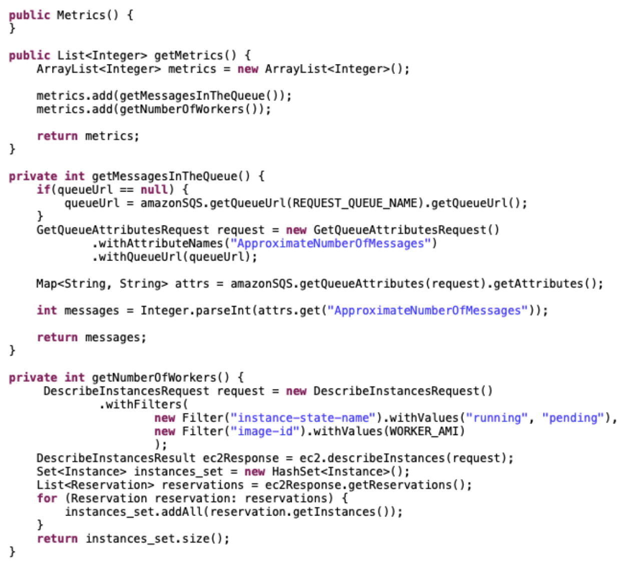
Figure 1

The overall system architecture is as shown in Figure 1. We have a web server which is developed using Spring boot. It receives user requests in the form of REST calls. The web server forms a JSON object having the input url and user UUID and stores it in the request queue (AWS SQS). It also starts the response service which waits for the response to be returned from the worker pool. UUIDs are used to create unique idenifiers for each request. The web server spawns a thread for each request which waits for the response. We kept a timeout of 120 seconds. We came with this number after performing various experiments. If response becomes available within 120 seconds, result will be returned to the user, otherwise it will be timed out. Request and Response queues will be SQS queues which contain the messages in the JSON format. The Load Balancer is the most sigificant module in our project. It is reponsible for scaling up or scaling down the number of workers (EC2 instances). Load Balancer is responsible for creating and terminating instances as required. The number of workers to be created or killed is calculated as Scaling Criteria. We have come up with an optimum formula to calculate the Scaling Criteria. If Scaling criteria is greater than 0, those many number of workers will be created. If it is equal to 0, all the worker will be terminated. If it is less than 0, those many number of terminate messages will be sent through terminate queue. The worker pool consists of worker instances which listen on the request queue and the terminate queue. Workers are created using JAVA api's provided. The created worker takes the image URL from the request queue, classifies it using the deployed CNN model and then sends out the response to the response queue. In our design we have one worker running on startup which will be ready to process requests. The workers also store the values into the AWS S3 bucket as [key, value] pairs. They also listen on the terminate queue every 5 requests and/or 20 seconds and do nothing if the messages are expired else they terminate. All the worker instances are self-terminating upon receiving a valid terminate message. Terminate queue is a SQS queue which is mainly used to terminate the instances during the scaleing process. It consists of batch messages and the batch size is determined by the scaling criteria. The workers execute the terminate requests only if their timestamps are not older than 5 seconds else the messages are considered as expired and the worker proceeds with processing requests from request queue.

IV. System Design & implementation

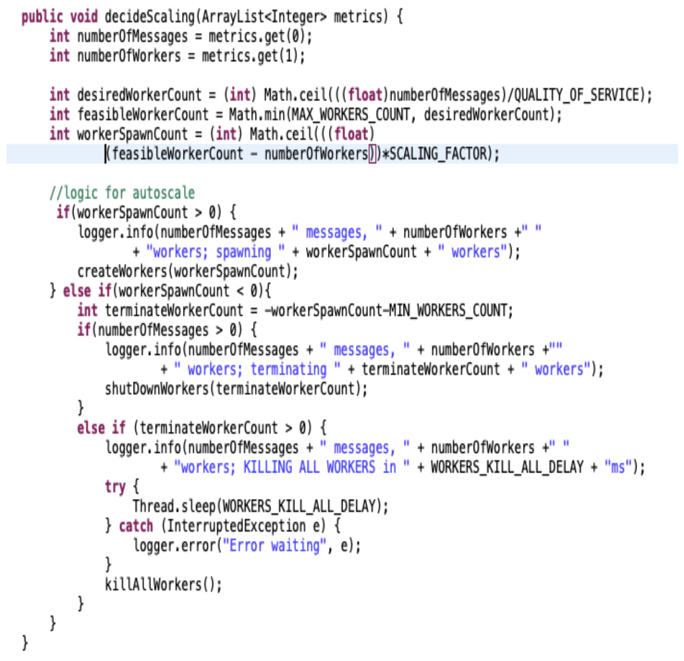
A. Load Balancer

Load balancer runs iteratively with a delay of 5 seconds. Metrics module is used to get the number of messages in the queue and the number of running instances. Figure 2 shows the implemetation of Metrics class. It has methods getMessagesInTheQueue() and getNumberOfWorkers() which return the number of messages in the queue and number of workers running respectively.



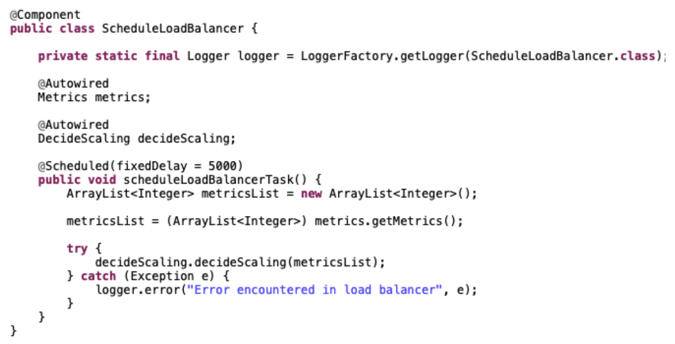
```java
public Metrics() {
}

public List<Integer> getMetrics() {
    ArrayList<Integer> metrics = new ArrayList<Integer>();

    metrics.add(getMessagesInTheQueue());
    metrics.add(getNumberOfWorkers());

    return metrics;
}

private int getMessagesInTheQueue() {
    if(queueUrl == null) {
        queueUrl = amazonSQS.getQueueUrl(REQUEST_QUEUE_NAME).getQueueUrl();
    }
    GetQueueAttributesRequest request = new GetQueueAttributesRequest()
            .withAttributeNames("ApproximateNumberOfMessages")
            .withQueueUrl(queueUrl);

    Map<String, String> attrs = amazonSQS.getQueueAttributes(request).getAttributes();

    int messages = Integer.parseInt(attrs.get("ApproximateNumberOfMessages"));

    return messages;
}

private int getNumberOfWorkers() {
     DescribeInstancesRequest request = new DescribeInstancesRequest()
             .withFilters(
                     new Filter("instance-state-name").withValues("running", "pending"),
                     new Filter("image-id").withValues(WORKER_AMI)
                     );
    DescribeInstancesResult ec2Response = ec2.describeInstances(request);
    Set<Instance> instances_set = new HashSet<Instance>();
    List<Reservation> reservations = ec2Response.getReservations();
    for (Reservation reservation: reservations) {
        instances_set.addAll(reservation.getInstances());
    }
    return instances_set.size();
}
```

Figure 2



```java
public void decideScaling(ArrayList<Integer> metrics) {
    int numberOfMessages = metrics.get(0);
    int numberOfWorkers = metrics.get(1);

    int desiredWorkerCount = (int) Math.ceil(((float)numberOfMessages)/QUALITY_OF_SERVICE);
    int feasibleWorkerCount = Math.min(MAX_WORKERS_COUNT, desiredWorkerCount);
    int workerSpawnCount = (int) Math.ceil(((float)
            (feasibleWorkerCount - numberOfWorkers))*SCALING_FACTOR);

    //logic for autoscale
     if(workerSpawnCount > 0) {
        logger.info(numberOfMessages + " messages, " + numberOfWorkers +" "
                + "workers; spawning " + workerSpawnCount + " workers");
        createWorkers(workerSpawnCount);
    } else if(workerSpawnCount < 0){
        int terminateWorkerCount = -workerSpawnCount-MIN_WORKERS_COUNT;
        if(numberOfMessages > 0) {
            logger.info(numberOfMessages + " messages, " + numberOfWorkers +""
                    + " workers; terminating " + terminateWorkerCount + " workers");
            shutDownWorkers(terminateWorkerCount);
        }
        else if (terminateWorkerCount > 0) {
            logger.info(numberOfMessages + " messages, " + numberOfWorkers +" "
                    + "workers; KILLING ALL WORKERS in " + WORKERS_KILL_ALL_DELAY + "ms");
            try {
                Thread.sleep(WORKERS_KILL_ALL_DELAY);
            } catch (InterruptedException e) {
                logger.error("Error waiting", e);
            }
            killAllWorkers();
        }
    }
}
```

Figure 3



```java
@Component
public class ScheduleLoadBalancer {

    private static final Logger logger = LoggerFactory.getLogger(ScheduleLoadBalancer.class);

    @Autowired
    Metrics metrics;

    @Autowired
    DecideScaling decideScaling;

    @Scheduled(fixedDelay = 5000)
    public void scheduleLoadBalancerTask() {
        ArrayList<Integer> metricsList = new ArrayList<Integer>();

        metricsList = (ArrayList<Integer>) metrics.getMetrics();

        try {
            decideScaling.decideScaling(metricsList);
        } catch (Exception e) {
            logger.error("Error encountered in load balancer", e);
        }
    }
}
```

Figure 4

Decide Scaling is responsible for deciding if scaling is required. Figure 3 shows the implementation of decide scaling logic. If workerSpawnCount is greater than 0, those many number of workers will be created. If it is equal to 0, all the worker will be terminated. If it is less than 0, those many number of terminate messages will be sent through terminate queue.

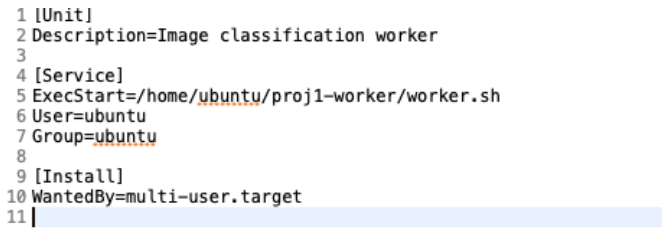
Load balancer runs iteratively every 5 seconds. Figure 3 shows the implementation of Load balancer. Method scheduleLoadBalancerTask() runs every 5 seconds to decide if scaling is required by calling decideScaling() method.

B. *Worker*

When a worker (EC2 instance) starts, worker.sh script is run at boot time. Worker.sh in turn calls worker.py which processes the input. To achieve this we created a system service which is implemented as shown in Figure 5.

Worker module gets the request, response and terminate queues and S3 bucket details. Worker runs the following two steps in loop:

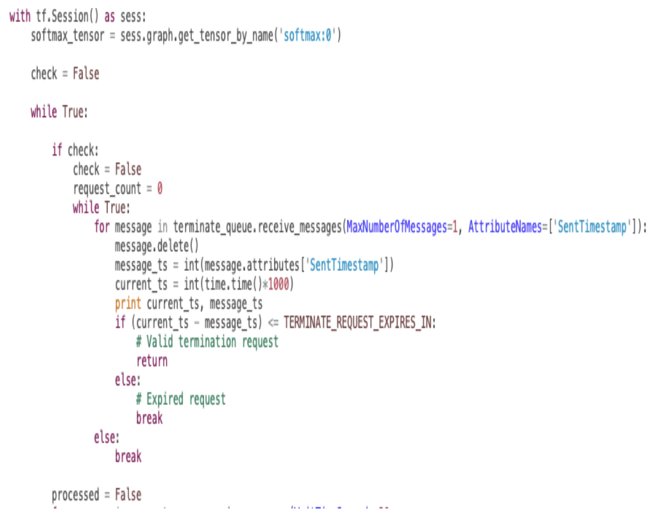
1) *If there are messages in terminate queue that are older than four seconds, it is considered to be expired messages else it is considered to be a valid termination request. The implementation is shown in Figure 6.*



```
[Unit]
Description=Image classification worker

[Service]
ExecStart=/home/ubuntu/proj1-worker/worker.sh
User=ubuntu
Group=ubuntu

[Install]
WantedBy=multi-user.target

```

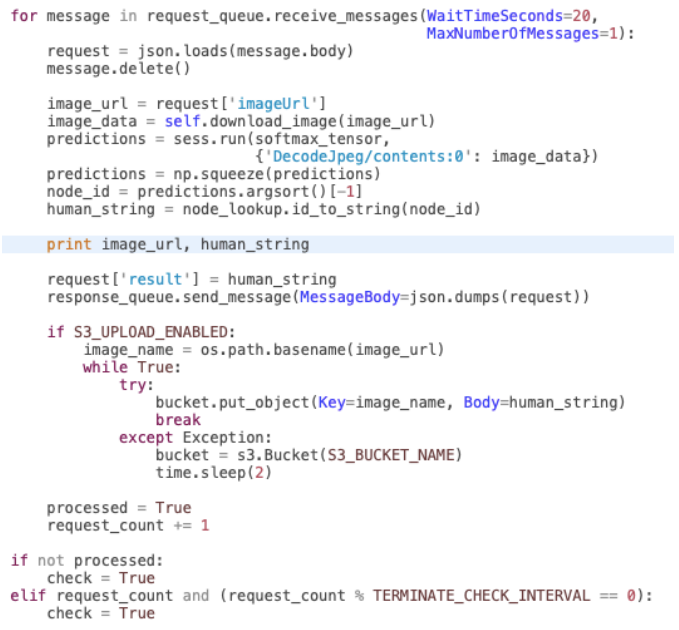
Figure 5

2) *The messages in the request queue are then taken up sequentially. The image is downloaded and then the classify_image predicts the results and enters the result in the S3 bucket and the respon queue. It deletes the processed item from the request queue before moving on to the next. This is implemented as shown in Figure 7.*



```python
with tf.Session() as sess:
    softmax_tensor = sess.graph.get_tensor_by_name('softmax:0')

    check = False

    while True:

        if check:
            check = False
            request_count = 0
            while True:
                for message in terminate_queue.receive_messages(MaxNumberOfMessages=1, AttributeNames=['SentTimestamp']):
                    message.delete()
                    message_ts = int(message.attributes['SentTimestamp'])
                    current_ts = int(time.time()*1000)
                    print current_ts, message_ts
                    if (current_ts - message_ts) <= TERMINATE_REQUEST_EXPIRES_IN:
                        # Valid termination request
                        return
                    else:
                        # Expired request
                        break
                else:
                    break

        processed = False
```

Figure 6



```python
for message in request_queue.receive_messages(WaitTimeSeconds=20,
                                              MaxNumberOfMessages=1):
    request = json.loads(message.body)
    message.delete()

    image_url = request['imageUrl']
    image_data = self.download_image(image_url)
    predictions = sess.run(softmax_tensor,
                           {'DecodeJpeg/contents:0': image_data})
    predictions = np.squeeze(predictions)
    node_id = predictions.argsort()[-1]
    human_string = node_lookup.id_to_string(node_id)

    print image_url, human_string

    request['result'] = human_string
    response_queue.send_message(MessageBody=json.dumps(request))

    if S3_UPLOAD_ENABLED:
        image_name = os.path.basename(image_url)
        while True:
            try:
                bucket.put_object(Key=image_name, Body=human_string)
                break
            except Exception:
                bucket = s3.Bucket(S3_BUCKET_NAME)
                time.sleep(2)

    processed = True
    request_count += 1

if not processed:
    check = True
elif request_count and (request_count % TERMINATE_CHECK_INTERVAL == 0):
    check = True
```

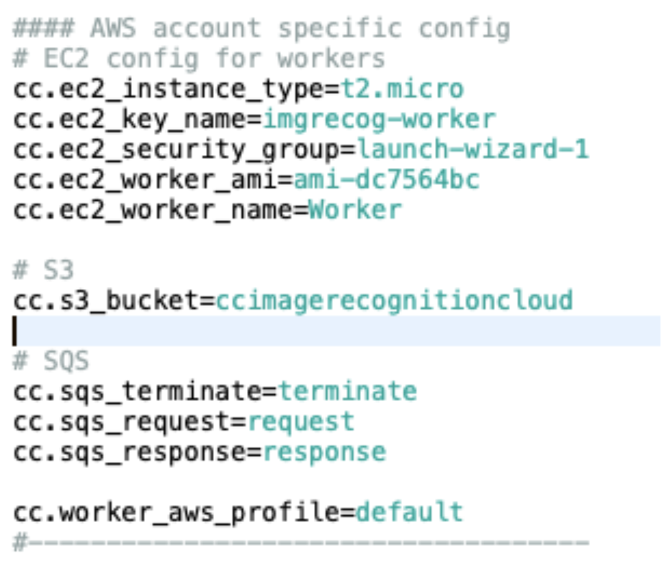
Figure 7

C. *Web Server*

The implementation of request and response is facilitated by using 2 queues (request and response AWS SQS). It has following sub modules.

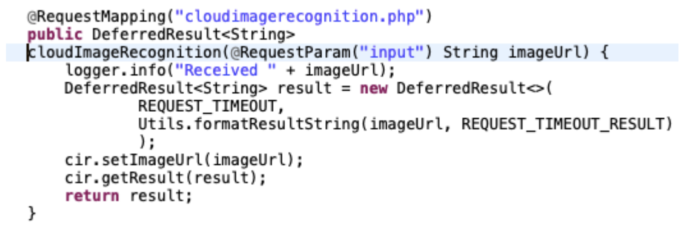
1) *Controller*

Controller takes the url of image as the input. For each request received, it creates a new DefferedResult object with a predefined timeout defined in case the result is not obtained in time. This is implemented as shown in Figure 9.



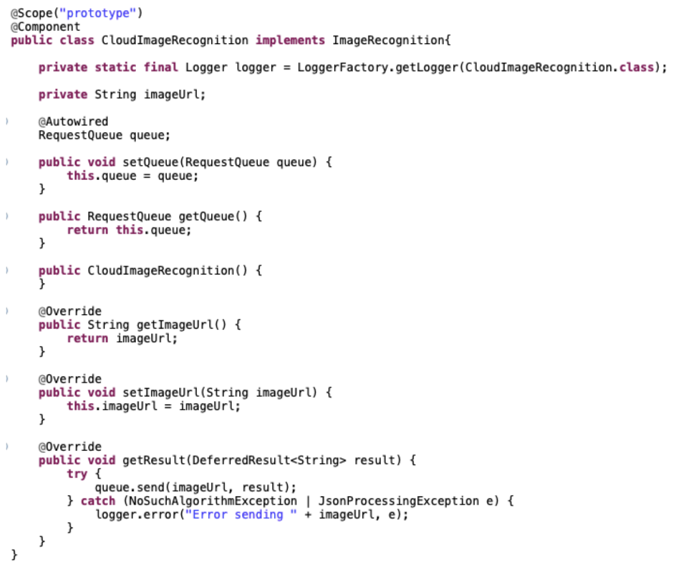
```
#### AWS account specific config
# EC2 config for workers
cc.ec2_instance_type=t2.micro
cc.ec2_key_name=imgrecog-worker
cc.ec2_security_group=launch-wizard-1
cc.ec2_worker_ami=ami-dc7564bc
cc.ec2_worker_name=Worker

# S3
cc.s3_bucket=ccimagerecognitioncloud

# SQS
cc.sqs_terminate=terminate
cc.sqs_request=request
cc.sqs_response=response

cc.worker_aws_profile=default
#-----------------------------------
```

Figure 8



```
@RequestMapping("cloudimagerecognition.php")
public DeferredResult<String>
cloudImageRecognition(@RequestParam("input") String imageUrl) {
    logger.info("Received " + imageUrl);
    DeferredResult<String> result = new DeferredResult<>(
            REQUEST_TIMEOUT,
            Utils.formatResultString(imageUrl, REQUEST_TIMEOUT_RESULT)
            );
    cir.setImageUrl(imageUrl);
    cir.getResult(result);
    return result;
}
```

Figure 9



```
@Scope("prototype")
@Component
public class CloudImageRecognition implements ImageRecognition{

    private static final Logger logger = LoggerFactory.getLogger(CloudImageRecognition.class);

    private String imageUrl;

    @Autowired
    RequestQueue queue;

    public void setQueue(RequestQueue queue) {
        this.queue = queue;
    }

    public RequestQueue getQueue() {
        return this.queue;
    }

    public CloudImageRecognition() {
    }

    @Override
    public String getImageUrl() {
        return imageUrl;
    }

    @Override
    public void setImageUrl(String imageUrl) {
        this.imageUrl = imageUrl;
    }

    @Override
    public void getResult(DeferredResult<String> result) {
        try {
            queue.send(imageUrl, result);
        } catch (NoSuchAlgorithmException | JsonProcessingException e) {
            logger.error("Error sending " + imageUrl, e);
        }
    }
}
```

Figure 10

2) *Cloud Image Recognition*

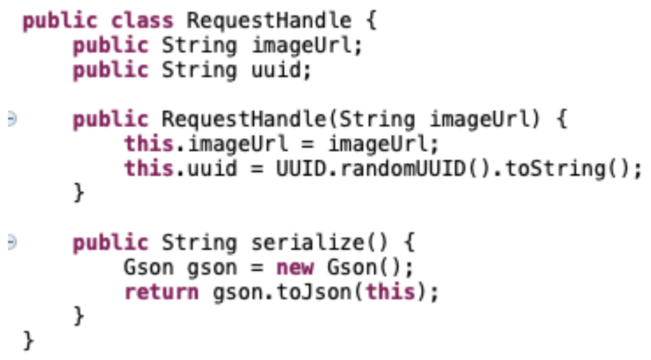
Cloud Image Recognition module is responsible for putting objects in the request queue. This is implemented as CloudImageRecognition class as shown in Figure 10.

3) *Deferred Result*

Deferred result allows asynchronous response receiveing. It creates a thread for each request which terminates only when the result is returned.

4) *Request Handler*

Request handler is used to store imageURL and a random number, uuid. It creates a random uuid and sets imageURL and uuid. This is implemented as shown in Figure 11.

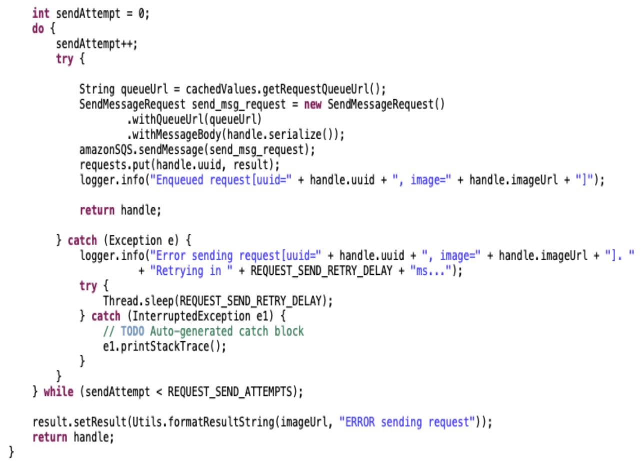


```
public class RequestHandle {
    public String imageUrl;
    public String uuid;

    public RequestHandle(String imageUrl) {
        this.imageUrl = imageUrl;
        this.uuid = UUID.randomUUID().toString();
    }

    public String serialize() {
        Gson gson = new Gson();
        return gson.toJson(this);
    }
}
```

Figure 11

5) *Request Queue*

Request queue contains requests, which will be a Hash Map of uuid(a string consisting of a random number) and Deffered Result object of each of the URL as response. This Hash Map is used to obtain the deffered result object for a given uuid. This is implemented as shown in Figure 12.

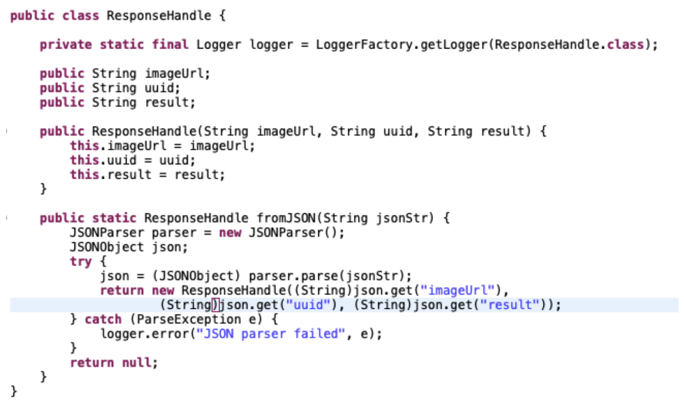


```
    int sendAttempt = 0;
    do {
        sendAttempt++;
        try {

            String queueUrl = cachedValues.getRequestQueueUrl();
            SendMessageRequest send_msg_request = new SendMessageRequest()
                    .withQueueUrl(queueUrl)
                    .withMessageBody(handle.serialize());
            amazonSQS.sendMessage(send_msg_request);
            requests.put(handle.uuid, result);
            logger.info("Enqueued request[uuid=" + handle.uuid + ", image=" + handle.imageUrl + "]");

            return handle;

        } catch (Exception e) {
            logger.info("Error sending request[uuid=" + handle.uuid + ", image=" + handle.imageUrl + "]. "
                    + "Retrying in " + REQUEST_SEND_RETRY_DELAY + "ms...");
            try {
                Thread.sleep(REQUEST_SEND_RETRY_DELAY);
            } catch (InterruptedException e1) {
                // TODO Auto-generated catch block
                e1.printStackTrace();
            }
        }
    } while (sendAttempt < REQUEST_SEND_ATTEMPTS);

    result.setResult(Utils.formatResultString(imageUrl, "ERROR sending request"));
    return handle;
}
```

Figure 12

6) *Response Handler*

It has the same overall purpose as Request handler. The difference is it also has result as one of its variables. This is implemented as shown in Figure 13.

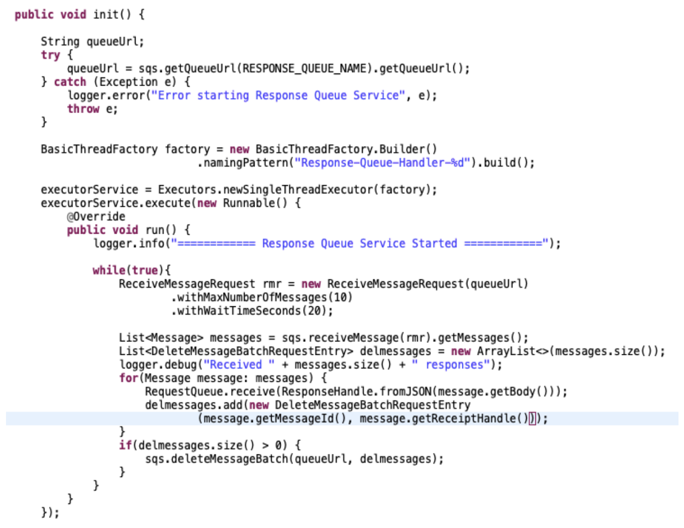


```java
public class ResponseHandle {

    private static final Logger logger = LoggerFactory.getLogger(ResponseHandle.class);

    public String imageUrl;
    public String uuid;
    public String result;

    public ResponseHandle(String imageUrl, String uuid, String result) {
        this.imageUrl = imageUrl;
        this.uuid = uuid;
        this.result = result;
    }

    public static ResponseHandle fromJSON(String jsonStr) {
        JSONParser parser = new JSONParser();
        JSONObject json;
        try {
            json = (JSONObject) parser.parse(jsonStr);
            return new ResponseHandle((String)json.get("imageUrl"),
                    (String)json.get("uuid"), (String)json.get("result"));
        } catch (ParseException e) {
            logger.error("JSON parser failed", e);
        }
        return null;
    }
}
```

Figure 13

7) *Response Service*

This module is always listening to the response queue and querying it for messages. The messages are received as a batch of maximum size 10, they are deleted from the queue and each of them are passed to the receive of Request Queue, which ultimately makes the response being presented to the client. This is implemented as shown in Figure 14.



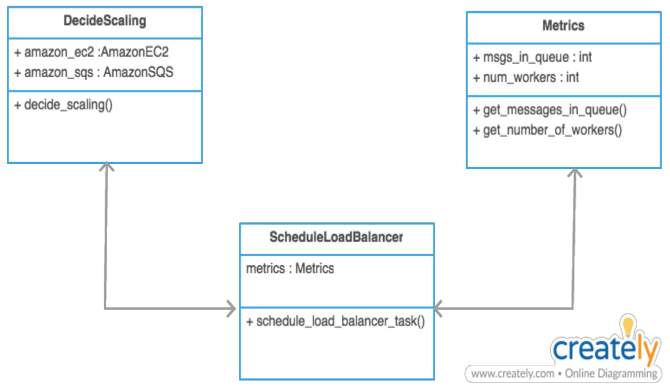
```java
public void init() {

    String queueUrl;
    try {
        queueUrl = sqs.getQueueUrl(RESPONSE_QUEUE_NAME).getQueueUrl();
    } catch (Exception e) {
        logger.error("Error starting Response Queue Service", e);
        throw e;
    }

    BasicThreadFactory factory = new BasicThreadFactory.Builder()
                        .namingPattern("Response-Queue-Handler-%d").build();

    executorService = Executors.newSingleThreadExecutor(factory);
    executorService.execute(new Runnable() {
        @Override
        public void run() {
            logger.info("=========== Response Queue Service Started ===========");

            while(true){
                ReceiveMessageRequest rmr = new ReceiveMessageRequest(queueUrl)
                        .withMaxNumberOfMessages(10)
                        .withWaitTimeSeconds(20);

                List<Message> messages = sqs.receiveMessage(rmr).getMessages();
                List<DeleteMessageBatchRequestEntry> delmessages = new ArrayList<>(messages.size());
                logger.debug("Received " + messages.size() + " responses");
                for(Message message: messages) {
                    RequestQueue.receive(ResponseHandle.fromJSON(message.getBody()));
                    delmessages.add(new DeleteMessageBatchRequestEntry
                            (message.getMessageId(), message.getReceiptHandle()));
                }
                if(delmessages.size() > 0) {
                    sqs.deleteMessageBatch(queueUrl, delmessages);
                }
            }
        }
    });
```

Figure 14

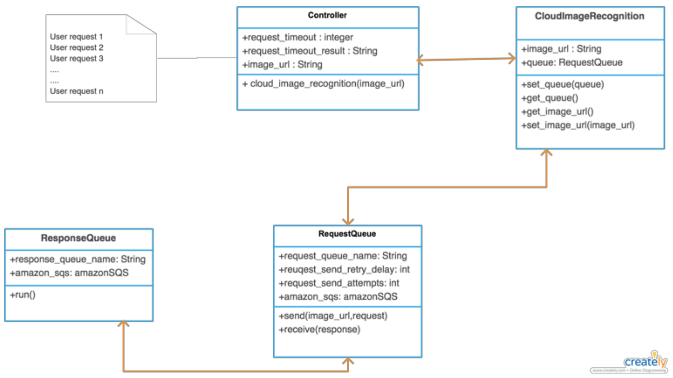
Figure 8 shows the configurations for connecting to AWS and using their services.

D. *Class Diagrams*

Below is the class diagram for load balancer module. Class ScheduleLoadbalancer interacts with class Metrics to get the number of messages in the queue and the number of workers running. It then calls method decide_scaling of class DecideScaling to calculate the scaling criteria and scale up and scale down the workers accordingly.

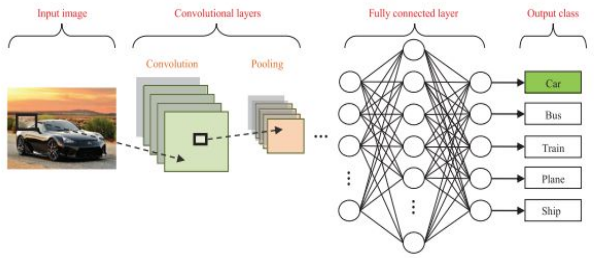


Below is the class diagram for web server module. User requests are received by the class Controller. Controller sends the image url to class CloudImageRecognition. Class CloudImageRecognition generates uuid for that request and calls the send method of classs RequestQueue. RequestQueue stores the {request_url, uuid} in request SQS in JSON format.



Class ResponseQueue listens to AWS response SQS. When resonse queue gets the result from Worker, ResponseQueue calls the receive method of RequestQueue. We used Creately to create class diagrams[7].

E. *CNN Model and Implementatiom*



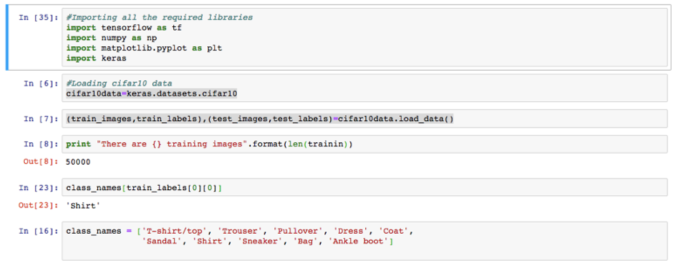
CNNs belong to category of feedforward networks where information flow takes place in one direction only, from their inputs to their outputs. CNNs are biologically inspired. CNNs have multiple layers, among which the most important ones are convolutional layers and pooling layers. The convolutional layers serve as feature extractors, and thus they learn the feature representations of their input images.The purpose of the pooling layers is to reduce the spatial resolution of the feature maps and thus achieve spatial invariance to input distortions and translations. Several convolutional and pooling layers are usually stacked on top of each other to extract more abstract feature representations in moving through the network.

a) Input

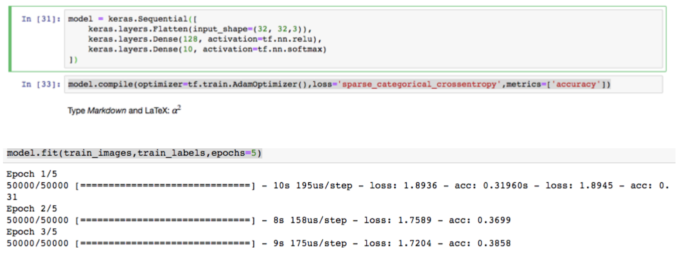
We initially used cifar10 and fashion mnist dataset to test our cloud service as the data are relatively small which made it easy to learn CNN[8]. Cifar10 dataset has 50000 labelled training images and 10000 testing images. Each image 32X32 pixel size. Cifar 10 dataset has 10 classes. Fashion mnist dataset has 60000 training images and 10000 testing images which are associated with 10 classes. Each image is of 28x28 pixel size[9].

b) Training

The first layer in training is flatten layer which tansforms 2D array into 1D array. For ex: 3x32 image will be converted to 1D array of 1024 elements. This unstcaked image will be fed to fully connected keras layer with 128 neurons. Next layer is a 10 node softmax layers which returns a probabilities of the labels for a given image. Before the model is ready for training, we add a loss function for measuring how acuurate is the model during training. The value of the loss function should be as minimal as possible for the model to get trained in right direction. We added a optimizer using which model would be updated based on the loss and the data it sees. We added accuracy as the metrics to see how accurate the model is during training. Below are some of the screenshots taken during this process.



```
In [35]: #Importing all the required libraries
         import tensorflow as tf
         import numpy as np
         import matplotlib.pyplot as plt
         import keras

In [6]:  #Loading cifar10 data
         cifar10data=keras.datasets.cifar10

In [7]:  (train_images,train_labels),(test_images,test_labels)=cifar10data.load_data()

In [8]:  print "There are {} training images".format(len(trainin))
Out[8]:  50000

In [23]: class_names[train_labels[0][0]]
Out[23]: 'Shirt'

In [16]: class_names = ['T-shirt/top', 'Trouser', 'Pullover', 'Dress', 'Coat',
                        'Sandal', 'Shirt', 'Sneaker', 'Bag', 'Ankle boot']
```



```
In [31]: model = keras.Sequential([
             keras.layers.Flatten(input_shape=(32, 32,3)),
             keras.layers.Dense(128, activation=tf.nn.relu),
             keras.layers.Dense(10, activation=tf.nn.softmax)
         ])

In [33]: model.compile(optimizer=tf.train.AdamOptimizer(),loss='sparse_categorical_crossentropy',metrics=['accuracy'])

         Type Markdown and LaTeX: α²

model.fit(train_images,train_labels,epochs=5)

Epoch 1/5
50000/50000 [==============================] - 10s 195us/step - loss: 1.8936 - acc: 0.31960s - loss: 1.8945 - acc: 0.
31
Epoch 2/5
50000/50000 [==============================] - 8s 158us/step - loss: 1.7589 - acc: 0.3699
Epoch 3/5
50000/50000 [==============================] - 9s 175us/step - loss: 1.7204 - acc: 0.3858
```

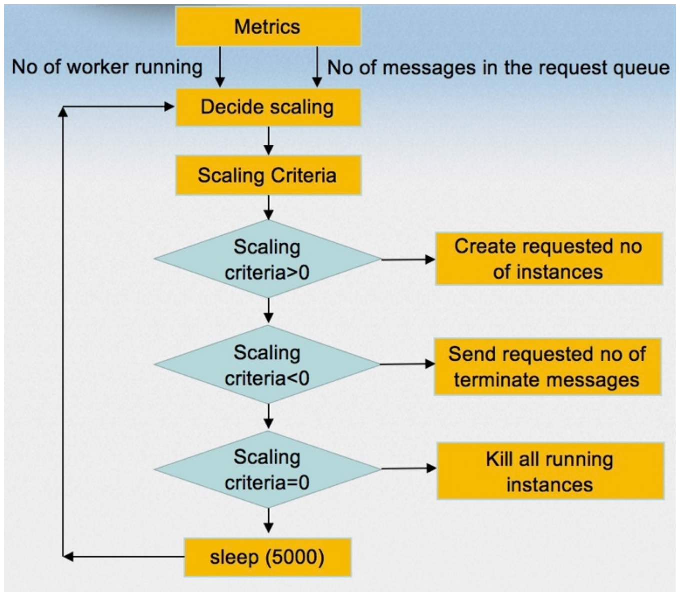
c) Evaluation

When we evaluated the accuracy on test images, we found out that it resulted in overfitting. Model is slightly more accurate on training data than testing data. We overcame this issue by using dropout techniques.

d) ImageNet and Inception

Though we trained a CNN model on CIFAR-10, we decided to use a pre-trained Inception model[3], as it is a better fit for our cloud service. Inception model was responsible for setting the new state of art for image classification and detection in ImageNet[10] large scale visual recognition challenge 2014(ILSVRC-2014).

F. Autoscaling


Metrics
No of worker running
No of messages in the request queue
Decide scaling
Scaling Criteria
Scaling criteria>0
Create requested no of instances
Scaling criteria<0
Send requested no of terminate messages
Scaling criteria=0
Kill all running instances
sleep (5000)

The load balancer which handles the autoscaling is integrated in the Spring boot application and it executes on a different thread with a delay of 5 seconds. It decides the scaling by calculating a scaling criteria. To calculate the scaling criteria, it first gets the number of messages in the request queue and the number of workers in the worker pool metrics. To calculate the scaling criteria, we first need to calculate the Feasible Worker Count(*Fwc*),

$$F = min(W ,D)$$

Where, D_{wc} is the Desired worker count, which is defined by, $D_{WC} = N4/QOS$ where, QOS is the quality of serice, N_q is the number of messages in the request queue. W_{max} is the maximum worker count. Scaling criteria,

$S = F_{wc} - N_w$ where, N_w is the number of workers already running. Algorithm goes like below:

```
If $S > 0$
        Create S number of workers
Else if $S < 0$
        Terminate (-S) number of workers by sending a batch
        terminate request with batch size of (-s)
Else if $S$ == 0
        Kill all idle workers
```

V. Performance Evaluation

A. Experimental Setting

Following are the services and configurations we used.

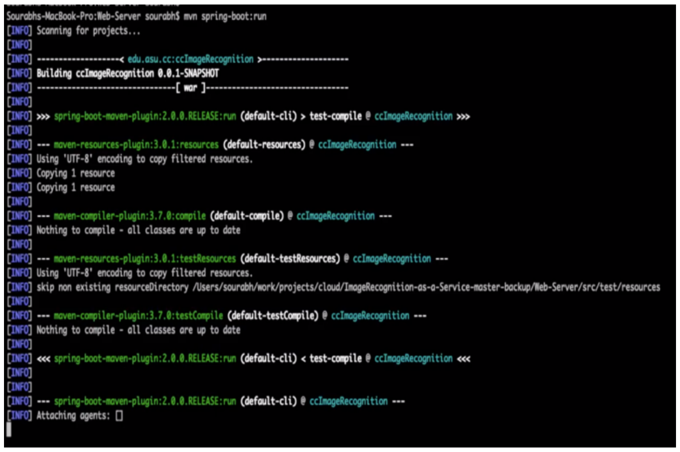
1) *AWS Elastic Compute Cloud(EC2)*
   Instance type- t2.micro
   Maximum number of instances- 19
   Operating System- Ubuntu
   Softwares installed- Tensorflow, Keras, Python.

2) *AMI(Amazon Machine Image)*
   AMI is created from the base EC2 instance named as worker, which contains the image recognition model and the code to process the input image. All the subsequent workers are created from this AMI and are named as classifycloud. Figure 15 shows how worker pool looks like. Figure 17 shows the created AMI.

3) *AWS Simple Queue Service(SQS)*
   Type- Standard SQS Queue
   Queue names- request queue, response queue and terminate queue
   Response queue timeout- 120 seconds
   Figure 16 shows the created queues. Figure 19 shows the messages in the request queue.

4) *Simple Storage Service(S3)*
   Region- us-west-1
   Name- ccimagerecognition
   Figure 20 shows the screenshot of the S3 bucket which stores the image url

5) *Web server*
   Server- Tomcat
   Port- 50003
   Framework- Spring Boot
   Figure 21 shows the Tomcat server initialization. Figure 18 shows how the workers are automatically killed based on the autoscaling logic.



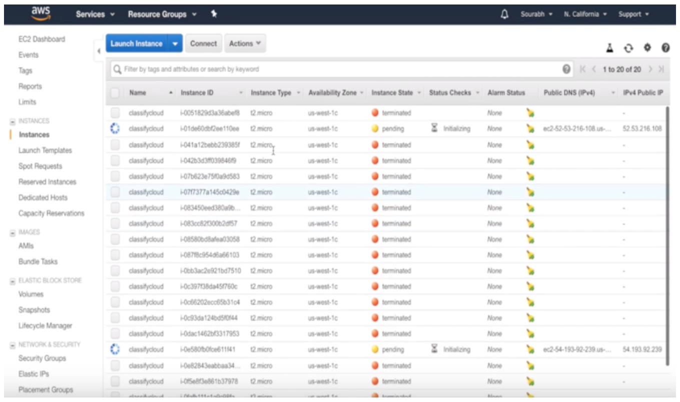
```
Sourabhs-MacBook-Pro:Web-Server sourabh$ mvn spring-boot:run
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------< edu.asu.cc:ccImageRecognition >-------------------
[INFO] Building ccImageRecognition 0.0.1-SNAPSHOT
[INFO] --------------------------------[ war ]--------------------------------
[INFO]
[INFO] >>> spring-boot-maven-plugin:2.0.0.RELEASE:run (default-cli) > test-compile @ ccImageRecognition >>>
[INFO]
[INFO] --- maven-resources-plugin:3.0.1:resources (default-resources) @ ccImageRecognition ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] Copying 1 resource
[INFO] Copying 1 resource
[INFO]
[INFO] --- maven-compiler-plugin:3.7.0:compile (default-compile) @ ccImageRecognition ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] --- maven-resources-plugin:3.0.1:testResources (default-testResources) @ ccImageRecognition ---
[INFO] Using 'UTF-8' encoding to copy filtered resources.
[INFO] skip non existing resourceDirectory /Users/sourabh/work/projects/cloud/ImageRecognition-as-a-Service-master-backup/Web-Server/src/test/resources
[INFO]
[INFO] --- maven-compiler-plugin:3.7.0:testCompile (default-testCompile) @ ccImageRecognition ---
[INFO] Nothing to compile - all classes are up to date
[INFO]
[INFO] <<< spring-boot-maven-plugin:2.0.0.RELEASE:run (default-cli) < test-compile @ ccImageRecognition <<<
[INFO]
[INFO]
[INFO] --- spring-boot-maven-plugin:2.0.0.RELEASE:run (default-cli) @ ccImageRecognition ---
[INFO] Attaching agents: □
```

Figure 21

6) *Neural network*
   Type- Convolutional Neural Networks
   Model used- Inception
   Dataset- ImageNet, CIFAR-10

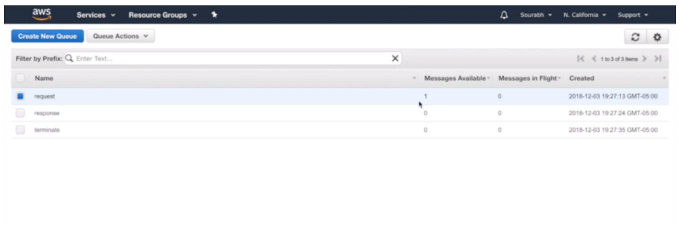
B. Performance Metrics

We tested ClassifyCloud by sending 40 parallel requests with 40 different image urls. Only 2 requests timed out when the request timeout was set to 120 seconds. We fixed this number as timeout considering the number of parallel instances that cane be created and the probable number of parallel requests that we might get. ClassifyCloud resulted in only 95% success rate in accurately classifying the input image. The average time taken to send the response back to the user and storing it in S3 bucket is ~50 seconds. Majority of this time is taken in initializing workers.



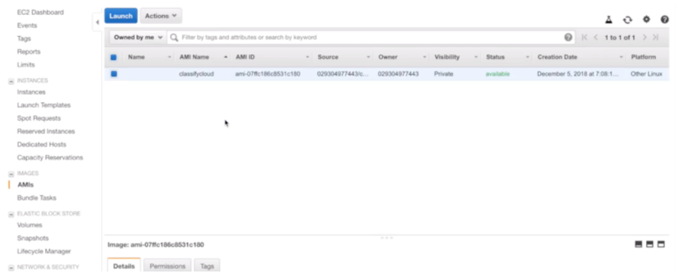
Name	Instance ID	Instance Type	Availability Zone	Instance State	Status Checks	Alarm Status	Public DNS (IPv4)	IPv4 Public IP
classifycloud	i-0051829d3a36abef8	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-01de60dbf2ee110ee	t2.micro	us-west-1c	🟡 pending	Initializing	None	ec2-52-53-216-108.us-...	52.53.216.108
classifycloud	i-041a12bebb239385f	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-042b3d3ff039846f9	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-07b623e75f0a9d583	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-07f7377a145c0429e	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-083450eed380a9b...	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-083cc82f300b2df57	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-08580bd8afea03058	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-087f8c954d6a66103	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0bb3ac2e921bd7510	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0c397f38da45f760c	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0c66202ecc65b31c4	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0c93da124bd5f0f44	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0dac1462bf3317953	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0e580fb0fce611f41	t2.micro	us-west-1c	🟡 pending	Initializing	None	ec2-54-193-92-239.us-...	54.193.92.239
classifycloud	i-0e82843eabbaa34...	t2.micro	us-west-1c	🔴 terminated		None		-
classifycloud	i-0f5e8f3e861b37978	t2.micro	us-west-1c	🔴 terminated		None		-

Figure 15



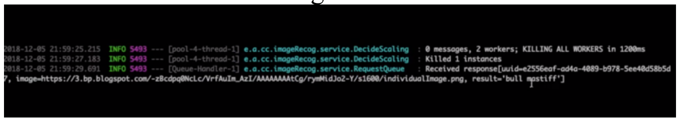
Name	Messages Available	Messages in Flight	Created
request	1	0	2018-12-03 19:27:13 GMT-05:00
response	0	0	2018-12-03 19:27:24 GMT-05:00
terminate	0	0	2018-12-03 19:27:35 GMT-05:00

Figure 16



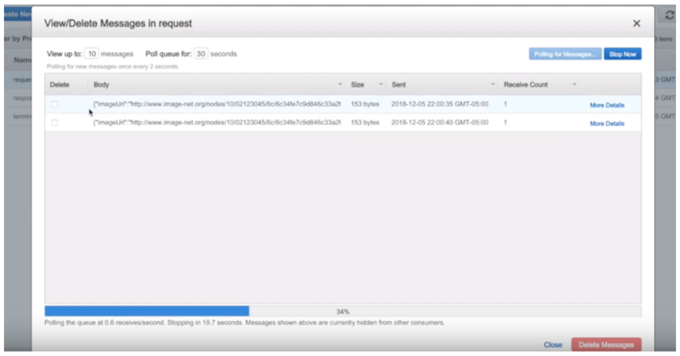
Name	AMI Name	AMI ID	Source	Owner	Visibility	Status	Creation Date	Platform
	classifycloud	ami-07ffc186c8531c180	029304977443/c...	029304977443	Private	available	December 5, 2018 at 7:08:1...	Other Linux

Figure 17



```
2018-12-05 21:59:25.215  INFO 5493 --- [pool-4-thread-1] e.a.cc.imageRecog.service.DecideScaling  : 0 messages, 2 workers; KILLING ALL WORKERS in 1200ms
2018-12-05 21:59:27.183  INFO 5493 --- [pool-4-thread-1] e.a.cc.imageRecog.service.DecideScaling  : Killed 1 instances
2018-12-05 21:59:29.691  INFO 5493 --- [Queue-Handler-1] e.a.cc.imageRecog.service.RequestQueue   : Received response[uuid=e2556eaf-ad4a-4089-b978-5ee40d58b5d
7, image=https://3.bp.blogspot.com/-zBcdpq0NcLc/VrfAuIm_AzI/AAAAAAAAtCg/rymMidJo2-Y/s1600/individualImage.png, result='bull mastiff']
```

Figure 18



View/Delete Messages in request

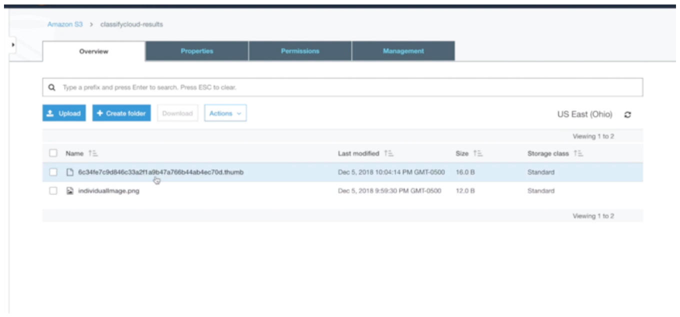
View up to: 10 messages Poll queue for: 30 seconds

Polling for new messages once every 2 seconds.

Delete	Body	Size	Sent	Receive Count	
	{"imageUrl":"http://www.image-net.org/nodes/10/02123045/6c/6c34fe7c9d846c33a2f	153 bytes	2018-12-05 22:00:35 GMT-05:00	1	More Details
	{"imageUrl":"http://www.image-net.org/nodes/10/02123045/6c/6c34fe7c9d846c33a2f	153 bytes	2018-12-05 22:00:40 GMT-05:00	1	More Details

Polling the queue at 0.6 receives/second. Stopping in 19.7 seconds. Messages shown above are currently hidden from other consumers.

Figure 19



Name	Last modified	Size	Storage class
6c34fe7c9d846c33a2f1a9b47a766b44ab4ec70d.thumb	Dec 5, 2018 10:04:14 PM GMT-0500	16.0 B	Standard
individualImage.png	Dec 5, 2018 9:59:30 PM GMT-0500	12.0 B	Standard

Figure 20

C. Discussions and Limitations

ClassifyCloud can spawn a maaximum of 19 parallel workers due to AWS free tier restrictions. There is no option to use different image classification model other than the already deployed Inception model. We are planning to provide this missing feature in out future work. We are planning to generalize our project to support other deep learning and machine learning models.

TEAM COORDINATION

Gurupad Hegde Mahabaleshwar(UFID- 78356173)

1) Webserver Part
   Developed part of load balancer implementation.
   Developed Request queue handler.
   Implementated Decide scaling logic and image recognition part.
2) Cloud Part
   Created and setup AWS accout and S3 buckets.
3) Worker
   Developed shell script and service to run at startup.

Sourabh Gopal Parvatikar(UFID- 79325142)

1) Webserver Part
   Developed part of load balancer implementation.
   Developed Response queue handler.
   Implemented the Schedule load balancer module and Metrics modules.
2) Cloud Part
   Created slave EC2 instance and SQS Queues.
   Registered the AMI.
3) Worker
   Developed worker.py.

We both worked together in designing the overall architecture of the system and coming up with the autoscaling and load balancer algorithms.

REFERENCES

[1] Article Comparing image recognition and classification APIs. https://www.upwork.com/hiring/data/comparing-image-recognition-apis/

[2] Comparing 3 best image analysis APIS by Scott Domes. https://engineering.musefind.com/we-compared-the-3-best-image-analysis-apis-here-s-what-we-learned-2d54cff5ae62

[3] Going deeper with convolutions by Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, Andrew Rabinovich. https://arxiv.org/abs/1409.4842

[4] https://searchenterpriseai.techtarget.com/definition/image-recognition

[5] https://medium.com/@ryanzhou7/running-spring-boot-on-amazon-web-services-for-free-f3b0aeec809

[6] https://www.baeldung.com/spring-boot

[7] https://creately.com/

[8] https://towardsdatascience.com/cifar-10-image-classification-in-tensorflow-5b501f7dc77c

[9] https://www.cs.toronto.edu/~kriz/cifar.html

[10] http://www.image-net.org/