**CSE 546 —Auto Scaling Image Classifier**

*Soham Sahare, Sai Srinivas Somarouthu, Shreya Amit Patel*

# Problem statement

A single Deep Learning Image Classifier can yield long waiting times to provide classification results for a humongous input dataset. Moreover, there are chances that the image classification model might run into errors and never give a response back. These impediments do not offer a good customer experience and can cause huge losses to a service provider / company.

It is of great importance to take necessary steps to tackle the problems mentioned above. One good solution is to have multiple replicas of the same image classifier model processing the input dataset in parallel. This decreases the wait times tremendously and is very fault tolerant to errors or unexpected system shutdowns that might happen during processing.

The idea is to have an Auto-Scalable architecture which can replicate the image classifier model to satisfy the input demand and execute results in parallel. When the input load decreases it can scale down to save resources and processing costs for the service provider. In this project we will present an architecture and a working model to demonstrate the solution mentioned above.

# Design and implementation

## Architecture

Our technology stack consists of **Elastic Compute Cloud (EC2), Simple Storage Service (S3) and Simple Queue Service (SQS)** provided by **Amazon Web Services.** We use **Python’s boto3** library in conjunction with **Flask** to interact with these services. We host a simple front-end application written in **HTML, CSS and JavaScript** using Flask’s hosting capabilities to provide an User Interface to upload multiple images for classification.

1. **Web Tier ->  
   Diagram

   Description automatically generated**
   1. The web tier is hosted on an EC2 Ubuntu Instance which hosts the *HTTP server* as well as *WebSocket connection* using Flask in Python
   2. A front-end UI provides capabilities to the user to upload multiple images as input to the image classifier model.
   3. These images are received through a POST call onto the HTTP server. *We only allow .JPEG, .JPG and .PNG image file extensions and we run a check on the file\_name to prevent any attacks on to the server or exposing of private files.*
   4. These images are processed , stored in S3, and sent to the SQS ‘request queue’.
   5. Meanwhile, we create two threads that do the following tasks 🡪
      1. calculates the length of the SQS ‘response queue’ and creates the ‘App tier stack’
      2. starts listening to the SQS ‘response queue’ to receive the processed results. These results are sent to the front-end using the WebSocket connection and are displayed to the user as they are processed.
2. **Application Tier ->**

**Diagram

Description automatically generated**

* 1. App tier stack comprises of multiple app-tier instances that process the input image file and outputs classification result.
  2. When an app tier instance comes online, it starts long polling to the request queue for input requests.
  3. App-tier instance then fetches the image from S3 using ‘image\_name’ message, passes the image file to the ‘Image Classifier’ which processes the result and then sends the output as a message to the ‘response queue’
  4. ‘App-tier stack’ continues to process the image requests until all messages in the request queue are exhausted.
  5. Instances are terminated on their own after polling an empty ‘request queue’ for X number of times (X is a configurable constant)

1. **Queuing and Storage Tier ->**

**Diagram

Description automatically generated**

* 1. *‘Queuing and Storage Tier’* comprise of Two First-In-First-Out (FIFO) SQS Queues and a storage bucket using S3.
  2. Their responsibility is to facilitate a loosely coupled architecture by acting as message brokers and provide persistent storage using buckets
  3. Both FIFO queues are setup to enable *long polling* when receiving messages and provide *visibility-timeout* for each message, they aid by ->
     1. **long polling:** each ‘receive message’ request waits for a maximum of ‘long\_poll\_duration’ (5 secs) before returning with NULL if no message is in the queue. This helps in reducing null responses.
     2. **message visibility timeout:** when a message is read by an app tier instance, it is hidden from other app-tier instances for the given duration. This helps in making sure only one instance processes a job and aids in improving fault tolerance.   
        If the processing instance crashes or fails to process the job, it will be available in the queue after the ‘visibility timeout’ duration has expired and some other app-tier instance can take in the request and process it to give the result.
  4. Request Queue contains Input jobs as messages that are consumed by the ‘App Tier Stack’ . Each message consists image\_name in the body, which is the key to fetch the original image from the persistence storage.
  5. Response Queue contains the classification result as messages that is consumed by the Listener Thread created by the Web tier when the user sends the input request at the beginning of the cycle.
  6. S3 bucket provides persistent storage for original image files and the classification output of each image for auditing purposes.

## Autoscaling

We use the Python boto3 library and SQS Request Queue to facilitate with auto-scaling capabilities. Concept is to use the length of the request queue to calculate the number of app tier instances needed to process the input dataset and when the request queue is depleted, the app tier instances should terminate on their own without any third party / human intervention.

After the images are processed by the web tier and job requests are added the queue, an independent thread is created by the web tier which queries the request queue for the number of messages that are visible and available for processing. Then it calculates the number of instances needed to process the job request and creates a maximum of 19 app tier instances, using Python boto3, to process the input dataset. We cap the number of app tier instances to 19 so that we do not exceed AWS Free tier usage, but this can be adjusted accordingly when not using AWS free tier.

Few examples of the above description are ->

* User uploads 5 images, 5 jobs are queued in the request queue and 5 app tier instances are created
* User uploads 50 images, 50 jobs are queued in the request queue and 19 app tier instances are created

When all requests are done processing by the App tier stack, each app tier instance continues to long-poll the request queue for a maximum of X times. This is done to use the existing App Tier stack in case more requests come into the queue. If they do, App tier stack will continue processing the results and output the classification result. If no new message comes in even after the Xth long-poll then the app tier instance prepares to terminate itself.

For the app tier instance to terminate itself, it requires it’s *ec2 instance id* which it fetches using the Python boto3 library and then uses the said library to create a terminate request to the AWS python API.

Diagram

Description automatically generated

In conclusion, the length of the request queue with the long polling duration plays a major role in the auto-scaling and termination architecture.

# Testing and evaluation

We tested our application using the imagenet-100 dataset provided by the Teaching Assistants. The results were as follows 🡪

<Insert testing example with 10 image>

<Insert testing example with 30 images>

<Insert testing example with 100 images>

# Code

Our project directory is as follows ->

* cloud\_project/
  + cloud\_iaas\_project/
    - templates/
      * index.html
    - static/
      * css/
        + style.css
      * js/
        + socketio\_client.js
    - \_\_init\_\_.py
    - helper.py
    - apptier.py
    - webtier.py
    - webtier\_helper.py
    - constants.py
    - imagenet-labels.json
  + architectural\_diagrams/

Purpose of each files 🡪

* **Cloud\_project**: Main project repository that is hosted on each web\_tier and app\_tier instances. This contains all the working logic for the application to work
  + **Templates/**: Contains the front-end page written in HTML, CSS and JavaScript
  + **Static**: Contains helper CSS and JavaScript files which aid the front-end page
    - Socketio\_client.js: Makes a web socket connection with the Flask HTTP server written in python to facilitate the receiving of image classification results as and when they arrive after processing.  
      Web sockets provide a real time communication channel which helps the user and back-end server to communicate as and when needed. Server can asynchronously post messages to the front end to display the image classification results
  + *\_\_init\_\_.py*: Creates a module package
  + *helper.py*: consists a set of modular functions that help in interacting with AWS such as S3, SQS and EC2. They do the following ->
    - S3 : Create a new bucket, upload a file to the bucket, download a file from the bucket
    - SQS: Create a new queue using attributes passed, get queue attributes, get queue URL, send message to a queue, receive ‘N’ messages from a queue, delete a message from the queue
    - EC2: create ec2 instances, get instance id of local ec2 instance, terminate ec2 instances
  + *apptier.py*: Responsible to long poll the request queue and give classification results using image classifier model provided by the Teaching Assistants. Also terminates itself when long polling an empty request queue for more than X times
  + *webtier.py*: Responsible to create new AWS resources, host an HTTP server and WebSocket connection using Flask and Flask-socketio respectively. Hosts the front end and receives input images from the user as POST call. Processes images, sends job ids to request queue, creates instances and listens for processed results from the response queue
  + *webtier\_helper.py*: Consists helper functions which facilitate the web tier in accomplishing its responsibilities. Majorly are 🡪
    - creating AWS resources such as S3 bucket and SQS queues
    - processing images and posting to SQS and S3
    - creating of app-tier-stack by querying the request queue
    - listening to response queue on an independent thread
  + *constants.py:* Configuration file consisting of constants that are retrieved throughout the application. They include ->
    - names for S3 bucket, SQS queues and EC2 instances
    - min and max number of app instances, security group ids to associate to new instances, app tier AMI id, key name, user data script that is run on app tier start, roles to associate EC2 instances
    - allowed image extensions, max number of long polls before terminating (referred as X above), long polling duration and other queue attributes
    - other minor constants

*Installation steps ->*

1. *Constants.py*
   1. Create an IAM Role with the following permissions and note down the Instance profile ARN ->
      1. EC2, S3 and SQS Full Access
      2. IAM Pass Role Inline Policy for all resources
   2. Create a security group with inbound access to SSH port, TCP 80, TCP 5000. Note down the group id.
   3. The most important step is to change the constants configuration file to reflect AWS properties that are associated to your account like key-pair name, Security group IDs, Role ARN, S3 bucket/ SQS Queue names, etc.
2. AMI IDs:
   1. Post demo we will make our AMI ids public and available in us-east-1 region.
3. *App tier:*
   1. Create a web tier instance with the shared AMI ID and SSH inside the instance. Update the constants file to reflect changes for your AWS properties as mentioned in step 1. Create a new AMI for your app tier stack and save it. We will use this in our web-tier implementation
   2. Terminate the instance, it will be spawned automatically during runtime.
4. *Web tier:*
   1. Create a web tier instance using the shared AMI ID and previously created security group, SSH inside the instances and update the constants.py as given in step 1. Update the app tier stack AMI ID which you created in step 3, in the constants file.
   2. Make sure to attach the Role that you created in Step 1.a to this instance
   3. Run the webtier.py script using this command ->  
      python3 webtier.py
   4. Note down the public IP of the instance

Voila! The application will be available on public IP, port 5000.

x.x.x.x:5000/

# Individual contributions

The auto-scaling image classification project hosted using AWS resources consisted of multiple iterations of long deliberations to finalize the architectural design, module implementation and independent unit testing, integration testing and stress testing for the entire application. We tested out the AWS SDK API in multiple languages before deciding on Python. There was independent interaction testing with all three AWS resources to learn how the API interacts with each of them, viz. S3, SQS and EC2. We had to study why and how to use Identity Access management features such as Users, Roles and Policies attached to them. In and all, it was a collaborative effort to design and develop this auto-scalable application. Below are descriptions for exemplary individual contributions for each member ->

1. *Soham Sahare*
   1. *Design:* I was responsible for the complete architectural design for the application such as using FIFO queues to main the sequence of input images, Single S3 bucket to store both Input images and Output results and the logic to start the app tier logic on instance boot. I presented the idea to have multi-threading in the web tier to accomplish the various responsibilities of web-controller and tasks like interaction with AWS resources. Another requirement for the project was to receive the image classification results asynchronously without human intervention, so I suggested the use of web-sockets for an asynchronous connection with the web tier and the front end.
   2. *Implementation:* I was responsible for the complete implementation of the web tier which includes ->
      1. Writing the HTTP server using Flask in Python, setting up API GET/POST Endpoints, handling uploading of files at the web tier
      2. Setting up a web socket connection with the front-end using Flask-socketio, managing reconnection of socket connection
      3. Handling of web socket events at the front end
      4. Handling multi-threading for execution of multiple jobs in parallel which include creating EC2 instances by polling the queue and listening to output results from the response queue
   3. *Testing:*  I performed Unit testing on web tier code logic by running all functions in independent environment. Integration Testing was a collaborative effort, and we tested the application on our own personal machines using our AWS credentials. I tested the application in different loads (from 1 to 300 images) and different input formats (.jpeg and .png)
   4. *Driving the project:* I set up the project directory, GIT hub repository, Project tracking using issues, Pull requests, Peer code review, etc.
2. *Shreya Patel*
   1. *Design:*
   2. *Implementation:*
   3. *Testing:*
3. *Sai Srinivas*
   1. *Design:*
   2. *Implementation:*
   3. *Testing:*