CSE 546 —Auto Scaling Image Classifier

Soham Sahare, Sai Srinivas Somarouthu, Shreya Amit Patel

1. Problem statement

In the age of the internet and increasing user demands any application should be robust, fault-tolerant, and ready to deal with the unexpected/ unprecedented loads to avoid disruption of services to the users. Moreover, to meet the user demands our application should elastically scale-up and scale-down based on the load. The fault-tolerant and autoscaling architecture of Amazon Web services (AWS) enables us to configure tiers of nodes (Instances) that self-monitor whether they need to scale based on a defined policy. This provides us a cost-effective Infrastructure as a service (IaaS) solution to address the above problem.

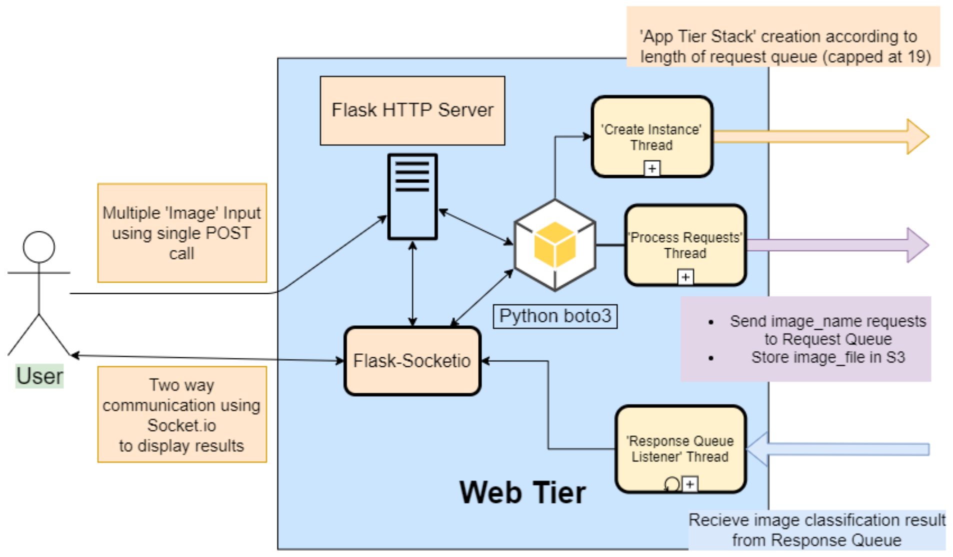
In this project, we are going to implement a fault-tolerant and auto-scalable architecture for an image recognition service to users, by using AWS cloud resources (EC2, SQS, S3) to perform image recognition on the images provided by the users using a deep learning model and return the result. The idea is to have an Auto-Scalable architecture of the image classifier model to satisfy the input demand and execute results in parallel, which supports the scale-up and scale-down of dedicated deep learning nodes to save resources and processing costs for the service provider based on the input images.

2. Design and implementation

2.1 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 a User Interface to upload multiple images for classification.

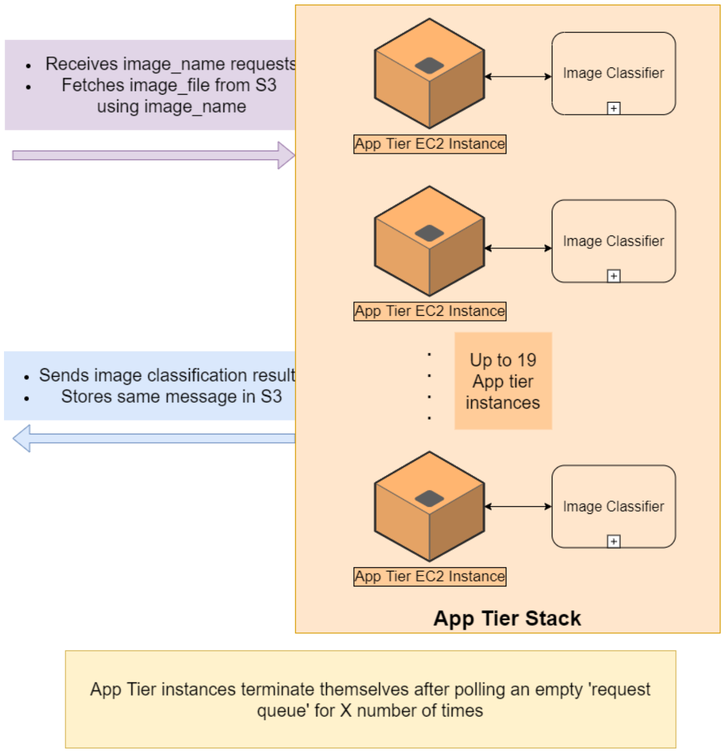
1. **Web Tier ->**


'App Tier Stack' creation according to length of request queue (capped at 19)
Flask HTTP Server
'Create Instance' Thread
Multiple 'Image' Input using single POST call
'Process Requests' Thread
Python boto3
User
Flask-Socketio
Send image_name requests to Request Queue
Store image_file in S3
Two way communication using Socket.io to display results
'Response Queue Listener' Thread
Web Tier
Recieve image classification result from Response Queue

   a. The web tier is hosted on an EC2 Ubuntu Instance which hosts the *HTTP server* as well as *WebSocket connection* using Flask in Python

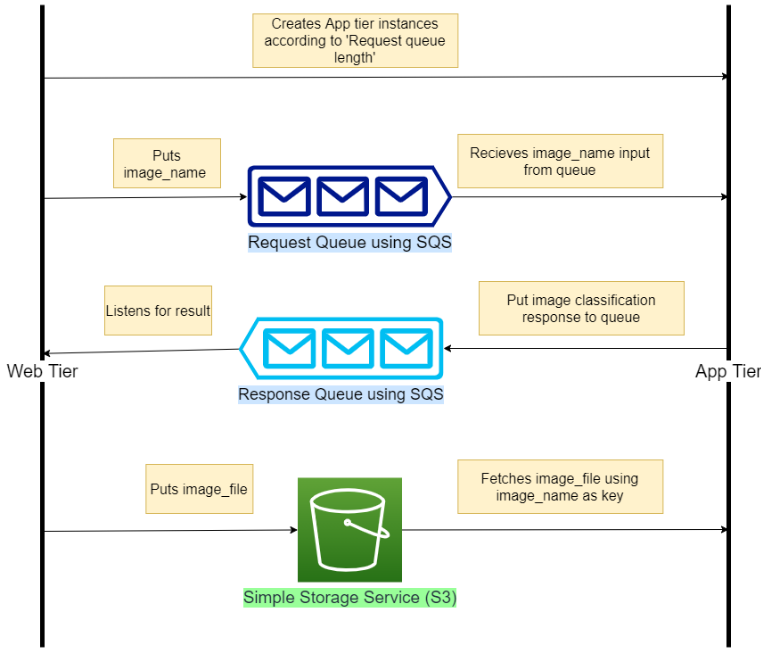
- **b.** A front-end UI provides capabilities to the user to upload multiple images as input to the image classifier model.
- **c.** These images are received through a POST call onto the HTTP server. *We only allow .JPEG, .JPG and .PNG image file extensions and we run a check on the file_name to prevent any attacks on to the server or exposing of private files.*
- **d.** These images are processed , stored in S3, and sent to the SQS 'request queue'.
- **e.** Meanwhile, we create two threads that do the following tasks →
    - **i.** calculates the length of the SQS 'response queue' and creates the 'App tier stack'
    - **ii.** 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 ->


Receives image_name requests
Fetches image_file from S3 using image_name
App Tier EC2 Instance
Image Classifier
App Tier EC2 Instance
Image Classifier
Up to 19 App tier instances
Sends image classification result
Stores same message in S3
App Tier EC2 Instance
Image Classifier
App Tier Stack
App Tier instances terminate themselves after polling an empty 'request queue' for X number of times

- **a.** App tier stack comprises of multiple app-tier instances that process the input image file and outputs the classification result.
- **b.** When an app tier instance comes online, it starts long polling to the request queue for input requests.
- **c.** 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'
- **d.** 'App-tier stack' continues to process the image requests until all messages in the request queue are exhausted.
- **e.** Instances are terminated on their own after polling an empty 'request queue' for X number of times (X is a configurable constant)

3. Queuing and Storage Tier ->


Creates App tier instances according to 'Request queue length'
Puts image_name
Recieves image_name input from queue
Request Queue using SQS
Listens for result
Put image classification response to queue
Web Tier
App Tier
Response Queue using SQS
Puts image_file
Fetches image_file using image_name as key
Simple Storage Service (S3)

Both SQS queues are **FIFO** so that order of messages is maintained

a. *'Queuing and Storage Tier'* comprise of Two First-In-First-Out (FIFO) SQS Queues and a storage bucket using S3.

b. Their responsibility is to facilitate a loosely coupled architecture by acting as message brokers and provide persistent storage using buckets

c. Both FIFO queues are setup to enable *long polling* when receiving messages and provide *visibility-timeout* for each message, they aid by ->

   i. **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.

   ii. **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.

d. 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.

e. 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.

f. S3 bucket provides persistent storage for original image files and the classification output of each image for auditing purposes.

2.2 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.

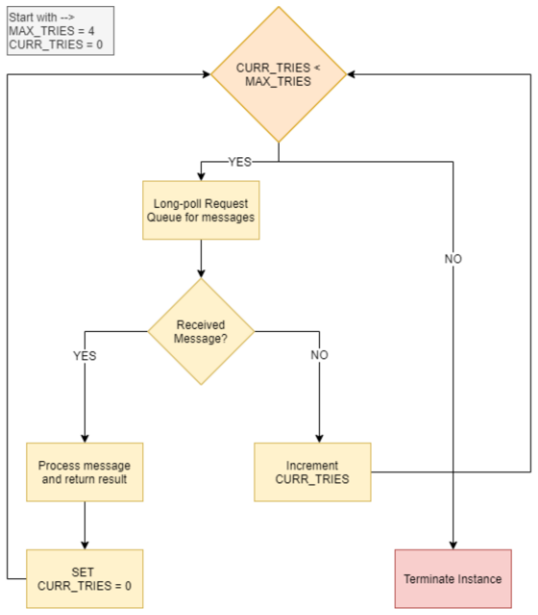
After the images are processed by the web tier and job requests are added the queue, an independent thread is created by the web tier which queries the request queue for the number of messages that are visible and available for processing. Then it calculates the number of instances needed to process 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.

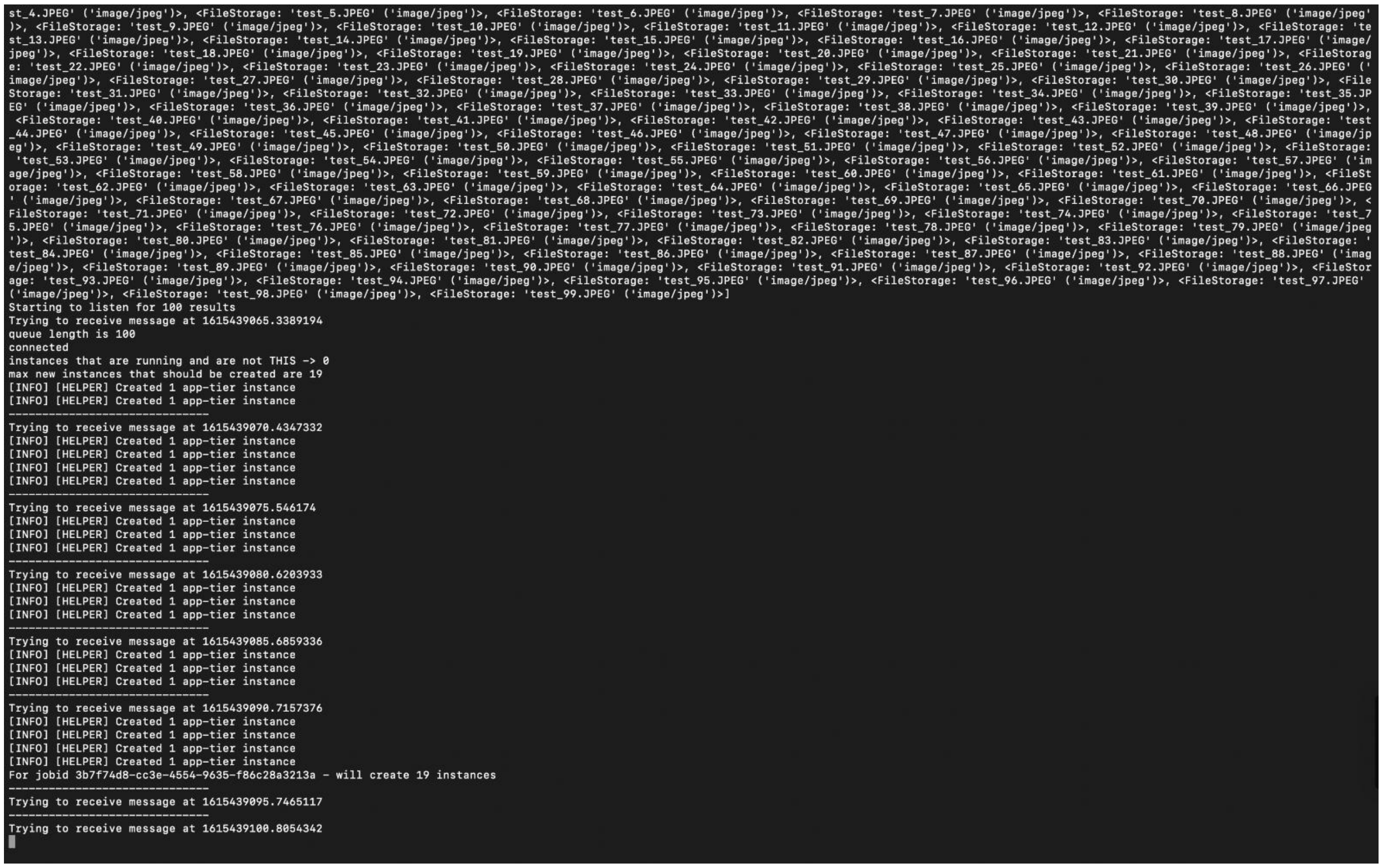
For the app tier instance to terminate itself, it requires it's *ec2 instance id* which it fetches using the Python boto3 library and then uses the said library to create a terminate request to the AWS python API.



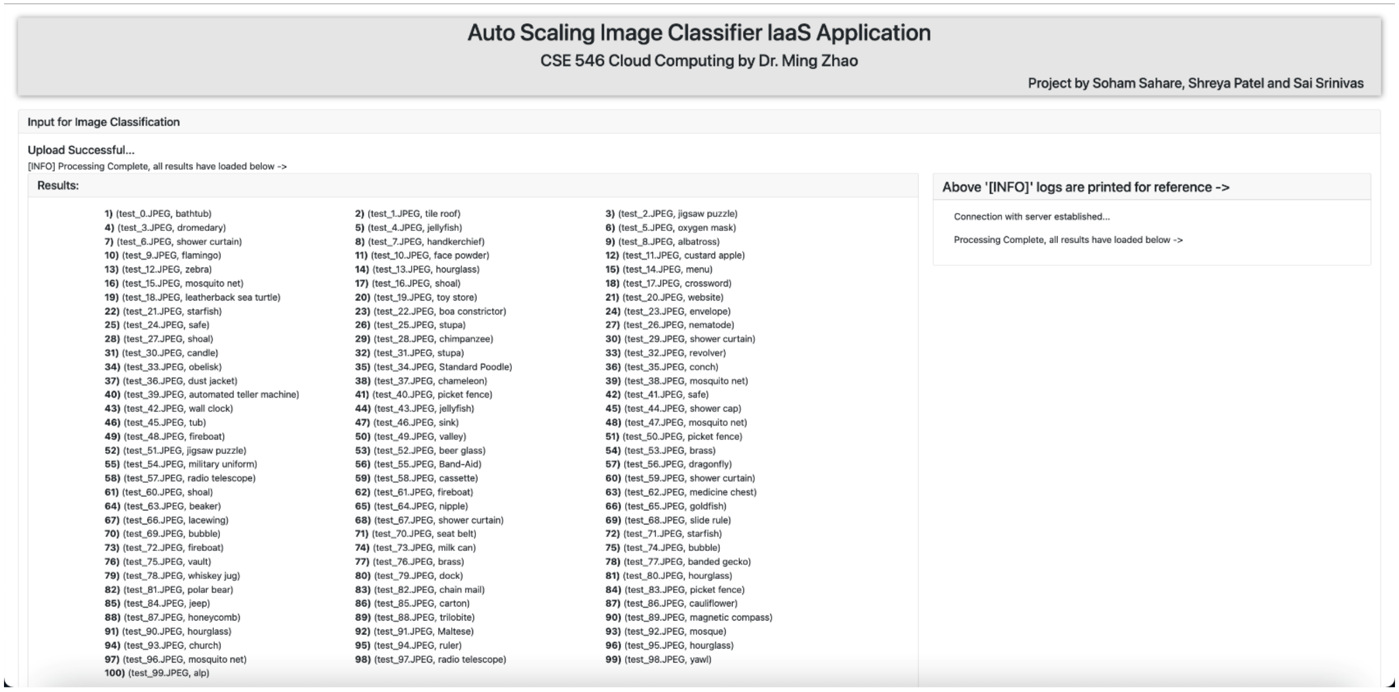
In conclusion, the length of the request queue with the long polling duration plays a major role in the auto-scaling and termination architecture.

3. Testing and evaluation

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



```
st_4.JPEG' ('image/jpeg')>, <FileStorage: 'test_5.JPEG' ('image/jpeg')>, <FileStorage: 'test_6.JPEG' ('image/jpeg')>, <FileStorage: 'test_7.JPEG' ('image/jpeg')>, <FileStorage: 'test_8.JPEG' ('image/jpeg'
)>, <FileStorage: 'test_9.JPEG' ('image/jpeg')>, <FileStorage: 'test_10.JPEG' ('image/jpeg')>, <FileStorage: 'test_11.JPEG' ('image/jpeg')>, <FileStorage: 'test_12.JPEG' ('image/jpeg')>, <FileStorage: 'te
st_13.JPEG' ('image/jpeg')>, <FileStorage: 'test_14.JPEG' ('image/jpeg')>, <FileStorage: 'test_15.JPEG' ('image/jpeg')>, <FileStorage: 'test_16.JPEG' ('image/jpeg')>, <FileStorage: 'test_17.JPEG' ('image/
jpeg')>, <FileStorage: 'test_18.JPEG' ('image/jpeg')>, <FileStorage: 'test_19.JPEG' ('image/jpeg')>, <FileStorage: 'test_20.JPEG' ('image/jpeg')>, <FileStorage: 'test_21.JPEG' ('image/jpeg')>, <FileStorag
e: 'test_22.JPEG' ('image/jpeg')>, <FileStorage: 'test_23.JPEG' ('image/jpeg')>, <FileStorage: 'test_24.JPEG' ('image/jpeg')>, <FileStorage: 'test_25.JPEG' ('image/jpeg')>, <FileStorage: 'test_26.JPEG' ('
image/jpeg')>, <FileStorage: 'test_27.JPEG' ('image/jpeg')>, <FileStorage: 'test_28.JPEG' ('image/jpeg')>, <FileStorage: 'test_29.JPEG' ('image/jpeg')>, <FileStorage: 'test_30.JPEG' ('image/jpeg')>, <File
Storage: 'test_31.JPEG' ('image/jpeg')>, <FileStorage: 'test_32.JPEG' ('image/jpeg')>, <FileStorage: 'test_33.JPEG' ('image/jpeg')>, <FileStorage: 'test_34.JPEG' ('image/jpeg')>, <FileStorage: 'test_35.JP
EG' ('image/jpeg')>, <FileStorage: 'test_36.JPEG' ('image/jpeg')>, <FileStorage: 'test_37.JPEG' ('image/jpeg')>, <FileStorage: 'test_38.JPEG' ('image/jpeg')>, <FileStorage: 'test_39.JPEG' ('image/jpeg')>,
 <FileStorage: 'test_40.JPEG' ('image/jpeg')>, <FileStorage: 'test_41.JPEG' ('image/jpeg')>, <FileStorage: 'test_42.JPEG' ('image/jpeg')>, <FileStorage: 'test_43.JPEG' ('image/jpeg')>, <FileStorage: 'test
_44.JPEG' ('image/jpeg')>, <FileStorage: 'test_45.JPEG' ('image/jpeg')>, <FileStorage: 'test_46.JPEG' ('image/jpeg')>, <FileStorage: 'test_47.JPEG' ('image/jpeg')>, <FileStorage: 'test_48.JPEG' ('image/jp
eg')>, <FileStorage: 'test_49.JPEG' ('image/jpeg')>, <FileStorage: 'test_50.JPEG' ('image/jpeg')>, <FileStorage: 'test_51.JPEG' ('image/jpeg')>, <FileStorage: 'test_52.JPEG' ('image/jpeg')>, <FileStorage:
 'test_53.JPEG' ('image/jpeg')>, <FileStorage: 'test_54.JPEG' ('image/jpeg')>, <FileStorage: 'test_55.JPEG' ('image/jpeg')>, <FileStorage: 'test_56.JPEG' ('image/jpeg')>, <FileStorage: 'test_57.JPEG' ('im
age/jpeg')>, <FileStorage: 'test_58.JPEG' ('image/jpeg')>, <FileStorage: 'test_59.JPEG' ('image/jpeg')>, <FileStorage: 'test_60.JPEG' ('image/jpeg')>, <FileStorage: 'test_61.JPEG' ('image/jpeg')>, <FileSt
orage: 'test_62.JPEG' ('image/jpeg')>, <FileStorage: 'test_63.JPEG' ('image/jpeg')>, <FileStorage: 'test_64.JPEG' ('image/jpeg')>, <FileStorage: 'test_65.JPEG' ('image/jpeg')>, <FileStorage: 'test_66.JPEG
' ('image/jpeg')>, <FileStorage: 'test_67.JPEG' ('image/jpeg')>, <FileStorage: 'test_68.JPEG' ('image/jpeg')>, <FileStorage: 'test_69.JPEG' ('image/jpeg')>, <FileStorage: 'test_70.JPEG' ('image/jpeg')>, <
FileStorage: 'test_71.JPEG' ('image/jpeg')>, <FileStorage: 'test_72.JPEG' ('image/jpeg')>, <FileStorage: 'test_73.JPEG' ('image/jpeg')>, <FileStorage: 'test_74.JPEG' ('image/jpeg')>, <FileStorage: 'test_7
5.JPEG' ('image/jpeg')>, <FileStorage: 'test_76.JPEG' ('image/jpeg')>, <FileStorage: 'test_77.JPEG' ('image/jpeg')>, <FileStorage: 'test_78.JPEG' ('image/jpeg')>, <FileStorage: 'test_79.JPEG' ('image/jpeg
')>, <FileStorage: 'test_80.JPEG' ('image/jpeg')>, <FileStorage: 'test_81.JPEG' ('image/jpeg')>, <FileStorage: 'test_82.JPEG' ('image/jpeg')>, <FileStorage: 'test_83.JPEG' ('image/jpeg')>, <FileStorage: '
test_84.JPEG' ('image/jpeg')>, <FileStorage: 'test_85.JPEG' ('image/jpeg')>, <FileStorage: 'test_86.JPEG' ('image/jpeg')>, <FileStorage: 'test_87.JPEG' ('image/jpeg')>, <FileStorage: 'test_88.JPEG' ('imag
e/jpeg')>, <FileStorage: 'test_89.JPEG' ('image/jpeg')>, <FileStorage: 'test_90.JPEG' ('image/jpeg')>, <FileStorage: 'test_91.JPEG' ('image/jpeg')>, <FileStorage: 'test_92.JPEG' ('image/jpeg')>, <FileStor
age: 'test_93.JPEG' ('image/jpeg')>, <FileStorage: 'test_94.JPEG' ('image/jpeg')>, <FileStorage: 'test_95.JPEG' ('image/jpeg')>, <FileStorage: 'test_96.JPEG' ('image/jpeg')>, <FileStorage: 'test_97.JPEG'
('image/jpeg')>, <FileStorage: 'test_98.JPEG' ('image/jpeg')>, <FileStorage: 'test_99.JPEG' ('image/jpeg')>]
Starting to listen for 100 results
Trying to receive message at 1615439065.3389194
queue length is 100
connected
instances that are running and are not THIS -> 0
max new instances that should be created are 19
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
------------------------------
Trying to receive message at 1615439070.4347332
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
------------------------------
Trying to receive message at 1615439075.546174
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
------------------------------
Trying to receive message at 1615439080.6203933
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
------------------------------
Trying to receive message at 1615439085.6859336
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
------------------------------
Trying to receive message at 1615439090.7157376
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
[INFO] [HELPER] Created 1 app-tier instance
For jobid 3b7f74d8-cc3e-4554-9635-f86c28a3213a - will create 19 instances
------------------------------
Trying to receive message at 1615439095.7465117
------------------------------
Trying to receive message at 1615439100.8054342
```



Auto Scaling Image Classifier IaaS Application

CSE 546 Cloud Computing by Dr. Ming Zhao

Project by Soham Sahare, Shreya Patel and Sai Srinivas

Input for Image Classification

Upload Successful...

[INFO] Processing Complete, all results have loaded below ->

Results:

1) (test_0.JPEG, bathtub) **2)** (test_1.JPEG, tile roof) **3)** (test_2.JPEG, jigsaw puzzle)
4) (test_3.JPEG, dromedary) **5)** (test_4.JPEG, jellyfish) **6)** (test_5.JPEG, oxygen mask)
7) (test_6.JPEG, shower curtain) **8)** (test_7.JPEG, handkerchief) **9)** (test_8.JPEG, albatross)
10) (test_9.JPEG, flamingo) **11)** (test_10.JPEG, face powder) **12)** (test_11.JPEG, custard apple)
13) (test_12.JPEG, zebra) **14)** (test_13.JPEG, hourglass) **15)** (test_14.JPEG, menu)
16) (test_15.JPEG, mosquito net) **17)** (test_16.JPEG, shoal) **18)** (test_17.JPEG, crossword)
19) (test_18.JPEG, leatherback sea turtle) **20)** (test_19.JPEG, toy store) **21)** (test_20.JPEG, website)
22) (test_21.JPEG, starfish) **23)** (test_22.JPEG, boa constrictor) **24)** (test_23.JPEG, envelope)
25) (test_24.JPEG, safe) **26)** (test_25.JPEG, stupa) **27)** (test_26.JPEG, nematode)
28) (test_27.JPEG, shoal) **29)** (test_28.JPEG, chimpanzee) **30)** (test_29.JPEG, shower curtain)
31) (test_30.JPEG, candle) **32)** (test_31.JPEG, stupa) **33)** (test_32.JPEG, revolver)
34) (test_33.JPEG, obelisk) **35)** (test_34.JPEG, Standard Poodle) **36)** (test_35.JPEG, conch)
37) (test_36.JPEG, dust jacket) **38)** (test_37.JPEG, chameleon) **39)** (test_38.JPEG, mosquito net)
40) (test_39.JPEG, automated teller machine) **41)** (test_40.JPEG, picket fence) **42)** (test_41.JPEG, safe)
43) (test_42.JPEG, wall clock) **44)** (test_43.JPEG, jellyfish) **45)** (test_44.JPEG, shower cap)
46) (test_45.JPEG, tub) **47)** (test_46.JPEG, sink) **48)** (test_47.JPEG, mosquito net)
49) (test_48.JPEG, fireboat) **50)** (test_49.JPEG, valley) **51)** (test_50.JPEG, picket fence)
52) (test_51.JPEG, jigsaw puzzle) **53)** (test_52.JPEG, beer glass) **54)** (test_53.JPEG, brass)
55) (test_54.JPEG, military uniform) **56)** (test_55.JPEG, Band-Aid) **57)** (test_56.JPEG, dragonfly)
58) (test_57.JPEG, radio telescope) **59)** (test_58.JPEG, cassette) **60)** (test_59.JPEG, shower curtain)
61) (test_60.JPEG, shoal) **62)** (test_61.JPEG, fireboat) **63)** (test_62.JPEG, medicine chest)
64) (test_63.JPEG, beaker) **65)** (test_64.JPEG, nipple) **66)** (test_65.JPEG, goldfish)
67) (test_66.JPEG, lacewing) **68)** (test_67.JPEG, shower curtain) **69)** (test_68.JPEG, slide rule)
70) (test_69.JPEG, bubble) **71)** (test_70.JPEG, seat belt) **72)** (test_71.JPEG, starfish)
73) (test_72.JPEG, fireboat) **74)** (test_73.JPEG, milk can) **75)** (test_74.JPEG, bubble)
76) (test_75.JPEG, vault) **77)** (test_76.JPEG, brass) **78)** (test_77.JPEG, banded gecko)
79) (test_78.JPEG, whiskey jug) **80)** (test_79.JPEG, dock) **81)** (test_80.JPEG, hourglass)
82) (test_81.JPEG, polar bear) **83)** (test_82.JPEG, chain mail) **84)** (test_83.JPEG, picket fence)
85) (test_84.JPEG, jeep) **86)** (test_85.JPEG, carton) **87)** (test_86.JPEG, cauliflower)
88) (test_87.JPEG, honeycomb) **89)** (test_88.JPEG, trilobite) **90)** (test_89.JPEG, magnetic compass)
91) (test_90.JPEG, hourglass) **92)** (test_91.JPEG, Maltese) **93)** (test_92.JPEG, mosque)
94) (test_93.JPEG, church) **95)** (test_94.JPEG, ruler) **96)** (test_95.JPEG, hourglass)
97) (test_96.JPEG, mosquito net) **98)** (test_97.JPEG, radio telescope) **99)** (test_98.JPEG, yawl)
100) (test_99.JPEG, alp)

Above '[INFO]' logs are printed for reference ->

Connection with server established...

Processing Complete, all results have loaded below ->

4. 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*
   a. Create an IAM Role with the following permissions and note down the Instance profile ARN ->
      i. EC2, S3 and SQS Full Access
      ii. IAM Pass Role Inline Policy for all resources
   b. Create a security group with inbound access to SSH port, TCP 80, TCP 5000. Note down the group id.
   c. 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:
   a. Post demo we will make our AMI ids public and available in us-east-1 region.
3. *App tier:*
   a. 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
   b. Terminate the instance, it will be spawned automatically during runtime.
4. *Web tier:*
   a. 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.
   b. Make sure to attach the Role that you created in Step 1.a to this instance
   c. Run the webtier.py script using this command ->
      python3 webtier.py
   d. Note down the public IP of the instance

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

x.x.x.x:5000/

5. 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

REFER NEXT PAGE

A. Soham Sahare (ASUID:1219459592)

Design:

I was responsible for the complete architectural design for the application such as placement of the various modules in the application workflow , using FIFO queues to maintain the sequence of messages, the logic to start the python scripts on instance boot; creation and management of communication channels. I presented the idea to have multi-threading in the web tier to accomplish the various tasks such as hosting a web server, spawning new app_tier instances according to queue length and listening to the response queue. Another core requirement for the project was to receive the image classification results without human intervention, so I suggested and integrated the use of web-sockets for an asynchronous connection with the web tier and the front end.

Implementation:

I was responsible for the complete implementation of the web tier (webtier.py and webtier_helper.py) which included ->

- Writing the HTTP server using Flask in Python, setting up API GET/POST Endpoints, handling uploading, filtering, and saving of image files at the web tier.
- Setting up a web socket connection with the front-end using Flask-socketio, managing reconnection of web sockets, sending of socket events at appropriate places in the application logic.
- Handling of web socket events at the front end as well as back end.
- 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.

Testing:

I performed Unit testing on web tier code logic by running all functions in independent and isolated environment. Integration Testing for components was a collaborative effort, we tested the components we implemented with the other components on our own personal machines. We then proceeded to test the complete application on different loads (ranging from 1 to 300 images) and different input formats (.jpeg and .png) on personal EC2 instances created using our AWS credentials.

Driving the development:

I set up the project directory, GIT repository and workflow, order of implementation of components, project tracking using issues, pull requests, peer code review and so on.

B. <u>**Shreya Patel**</u>(ASUID:1219436127)

Design:

I was responsible for designing the independent resource implementation of instances, queues and buckets using EC2, SQS and S3, respectively. I designed the AMIs for the instances. I created IAM roles and developed a helper module to assign these IAM roles on the fly to the app_tier instances. I also designed and developed the front-end web application to display the data in an organized and concise manner while keeping in mind UI/UX concepts.

Implementation:

1. I implemented helper.py which includes modular functions for:
   - Creating and terminating EC2 instances. Creation of the FIFO queues in SQS, along with sending, receiving and deleting of messages from the queue.
   - Creating a S3 bucket, uploading to bucket, and downloading the files from the bucket.
   - Interaction with AWS resources required several permissions which I added through the policies in the IAM roles and created a module to assign IAM roles to app_tier instances on the fly
2. I implemented index.html and styles.css for the front-end web application, which includes:
   - Bootstrap components to enhance the user experience of the application, by displaying the incoming data in a systematic and organized manner.
   - I used HTML and CSS to make the application more user intuitive and interactive.
3. I implemented the AMIs design and installation:
   - I configured the web and app tier EC2 instance environment and then created respective AMIs for both tiers.
   - The challenges occurring in this process were handled by me.

Testing:

I performed unit-testing of the helper module independently and then carried out integration testing along with other components such as app_tier and web_tier. I tested the frontend with dummy data as well as real-time data. I verified all the policies and AMIs of the instances during integration testing with different test case scenarios.

C. <u>Sai Srinivas Somarouthu</u>(ASUID:1219481822)

Design:

I was also responsible for designing complete app-tier logic on how to receive images from request FIFO queues and store results in required format in S3 and response FIFO queue and designed the listener logic for App_tier to get message from request FIFO queue before it gets terminated. App_tier was completely designed from scratch with using Python, app_tier logic, the deep learning algorithm and imagenet-label.json from original AMI provided by the Teaching Assistants.

I was responsible for the design and complete development of app-tier logic, and suggested use of S3 for storing output results in a text file for persistency and auditing purposes.

I have also implemented the termination logic for elastic scale in of app tiers if the request queue is empty for a set time duration.

Implementation:

I was responsible for the complete implementation of the app tier which includes:

- I have implemented the app tier module using libraries such as
  a) PIL, Torch vision, NumPy for the deep learning module from the given original AMI
  b) boto3, helper module to interact with the web tier and the frontend.
- Implemented the logic for extracting 'image_id' from request message from the FIFO request queue which will be fed to the deep learning model and used for storing results to S3 and FIFO response queue.
- Implemented the logic to recursively long-poll the request queue for completion of all the jobs and decide the termination of the instance.
- Implemented a fault tolerant and robust approach for elastic scale out
- Our designed app tier is never exposed to the outside world through the use of proper security group policies.

Testing:

I performed Unit testing on app tier module by running all functions in independent test environment and rigorously tested the elastic scale in and termination logic. Tested storing of results in S3 buckets, writing results to the response queue, giving meaningful names to app_tier instances. I also tested the integration of the deep learning module provided on my own personal machines. Moreover, I also tested the auto scalability of App_tier stack under different user loads varying from 1 to 300 user images using my personal AWS credentials.