CSE 546 — Project1 Report

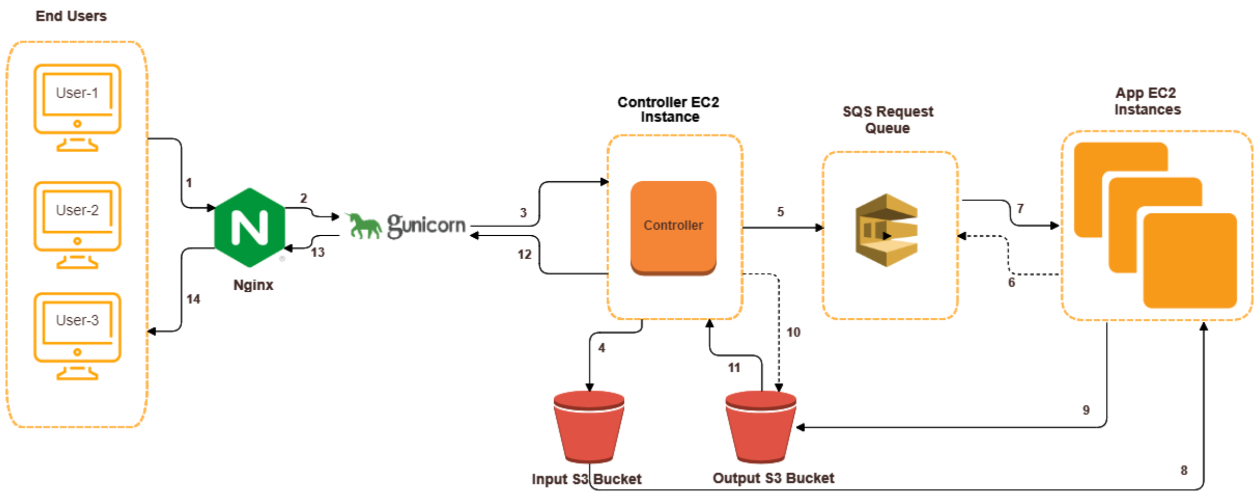
Risabh Raj (1219182289)
Sindhu Sree Aita (1220083085)
Bhavya Swetha Beri (1221768860)

1. Problem statement

To build an application which takes an image input from the user, recognizes the image using different AWS cloud services, the deep Learning algorithm and displays the result back to the user. It should be designed as such that multiple number of images and concurrent requests are to be handled.

2. Design and implementation

2.1 Architecture



End-to-End flow:

- ➢ The user uploads multiple images using the hosted website.
- ➢ The POST request is routed to Nginx server which acts as a reverse proxy for Gunicorn web server. Nginx is used to directly serve the static HTML and CSS files **(1)**.
- ➢ Nginx routes the request to Gunicorn WSGI HTTP server **(2)**. Both Nginx and Gunicorn are used to deploy the controller application.
- ➢ The controller is the master node of our application. After receiving the request **(3)**, it assigns a unique prefix to the image name which acts as the image-id.
- ➢ The controller then uploads the images into the input S3 bucket with key: image-id and value: Image **(4)**.
- ➢ The controller sends all the image-ids to the SQS request queue **(5)**.

- After that, the controller starts up the required Application EC2 instances based on the total messages in the SQS request queue, max images each instance can handle and total possible maximum instances. Max images per instance and total maximum instances are configurable at the Controller's end.
- The Application EC2 instances after booting up automatically execute the worker application. The worker applications start request queue polling to fetch the image-ids to process **(6)**.
- After getting the image-ids from the request queue **(7)**, the application fetches the corresponding image from the Input S3 bucket **(8)**.
- The application instance then runs the Deep Learning classifier model on the image to compute the prediction result.
- After processing the image and getting the prediction result, the app instance deletes the image-id from the request queue. Then it serializes the prediction result string into an object and uploads it to the Output S3 bucket where key: image-id and value: Serialized prediction object **(9)**.
- In controller, every concurrent request spawn a new handler thread. Each thread iterates over the set of image-ids it received in the POST request. Then for each image-id, the controller node thread queries the output S3 bucket to get the prediction output **(10)**. This polling mechanism is run continuously by each thread until each of them receives the prediction output for all the images of the request **(11)**.

Design Choice:

There were two ways for the controller to get the prediction output.

1. To fetch it from a Response queue,
2. To fetch it from Output S3 bucket.

- *We implemented and tested both. For a single request with multiple images, fetching prediction from response queue outperformed the S3 implementation. But, for concurrent requests, we noticed a race condition among the Controller threads to consume data from the response queue. The race condition was because each controller thread receives an independent set of images by the user. So, while one controller thread fetches some image-id from the response queue and figures out that this image-id does not belong to it, the image-id can no longer be accessed by the other threads. The image-id is currently in flight and not visible to other threads. This caused a noticeable delay in the overall processing of concurrent requests. Eventually after comparing the time taken by both implementations, we decided to poll the S3 bucket to get prediction result as it turned out to be more robust.*
- After receiving the prediction output of all images, the controller node converts it to the HTTP response HTML page and sends it to the user **(12, 13, 14)**.

2.2 Key Terms:

- Elastic Cloud Compute (EC2 Instance): It is a virtual server in the AWS cloud, that is used for hosting and running applications.
- Simple Storage Service(S3): It is a scalable storage system used for object storage in the form of key-value pairs.
- Simple Queue Service (SQS): Message channel that is used for transmitting and consuming messages across applications and promotes decoupling.

- Amazon Machine Image (AMI): Images are built out of applications and use to spin up virtual servers with the same specified configuration.
- Security Group: Used for providing security at the port level through rules established that filter traffic coming and going out of the EC2 Instance.
- IAM Role: Role created in the AWS with specific permissions and policies.

2.3 Autoscaling

Autoscaling is the ability of any distributed application to scale-out and scale-in based on the amount of load. In our architecture, the number of worker App-Tier EC2 instances are spawned and terminated based on the number of images we are getting.

Scaling-Out:

The Controller is responsible for scaling-out the app-tier instances. The amount of load each app instance will handle is configurable. The Controller determines the load by the number of image processing requests in the SQS Request queue. Based on the queue size and the Request handling capacity, the Controller spawns the required number of EC2 app instances. The maximum possible number of App-Instances is 19.

Scaling-In:

After sending all images to the queue, the Controller starts polling the Output S3 bucket for prediction results. Every App-Tier instance runs the Deep-Learning model on the image to get the classification prediction. Then, the App instance uploads the prediction result in the output S3 bucket. The Controller's polling mechanism fetches the prediction result for all the images sent in the request. After getting the final response i.e., the key-value pairs of image and prediction, the controller long polls the request queue. If there are no messages in the Request queue and no current messages in flight, the controller starts terminating all instances which were spawned for this request.

3. Testing and evaluation

3.1 Unit Testing:

All the components developed were unit tested first to check if all of them satisfies requirements. With S3 Manager and SQS Manager, S3 buckets were made with corresponding setups.

A) Elastic Cloud Compute Instance and Load Balancer

- In the given region, create an AWS instance and start it.
- Test total number of running instances and total no. of stopped instances.
- Methods to initiate stopped instances & wait till they all are running.
- Test if all the AWS Resources can be accessed by EC2 client.
- Able to terminate/stop an instance.
- Automatically shut-down the App-tier instances.
- Method to load the input image into the input bucket in S3 and add it to the SQS request queue.
- Method to store the predicted results to S3 output bucket.
- Method to poll all messages, from the request queue.

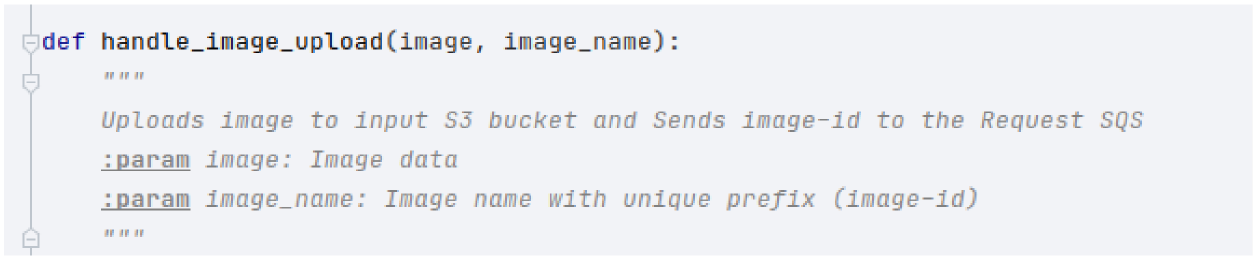
3.2 Integration Testing :

- Once unit testing was successful, we began integration testing in the following way,
- By populating messages in the queue, we tested the functionality of the EC2 application to poll the message from the queue, extract the image name from the message recieved, get the image from the S3 input bucket and execute the deep learning algorithm, push the processed result to the output S3 bucket.

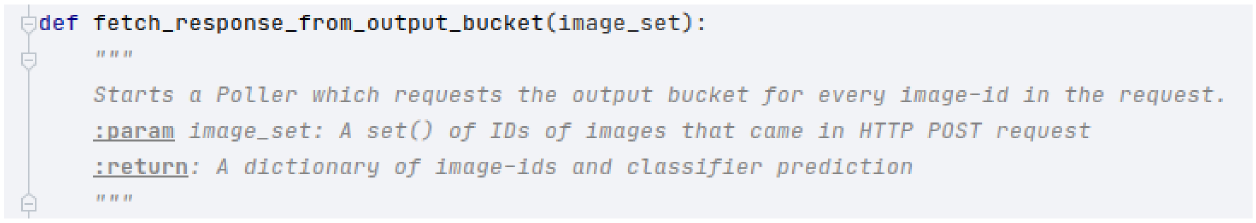
4. Code

Web-Tier:

1. app.py

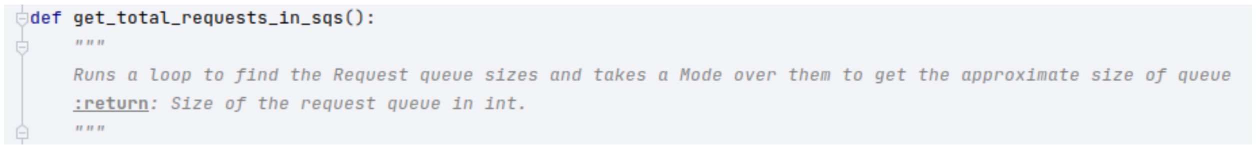


```
def handle_image_upload(image, image_name):
    """
    Uploads image to input S3 bucket and Sends image-id to the Request SQS
    :param image: Image data
    :param image_name: Image name with unique prefix (image-id)
    """
```

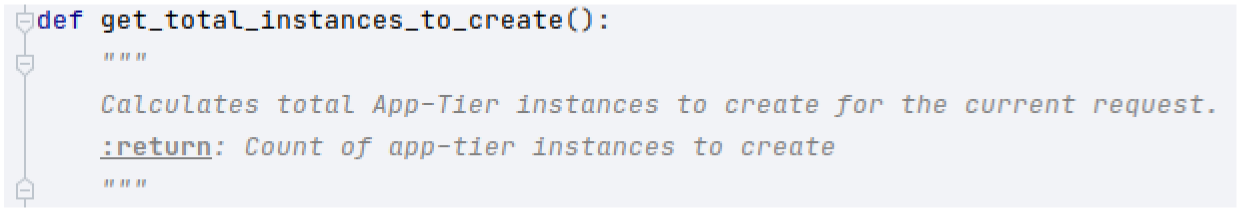


```
def fetch_response_from_output_bucket(image_set):
    """
    Starts a Poller which requests the output bucket for every image-id in the request.
    :param image_set: A set() of IDs of images that came in HTTP POST request
    :return: A dictionary of image-ids and classifier prediction
    """
```

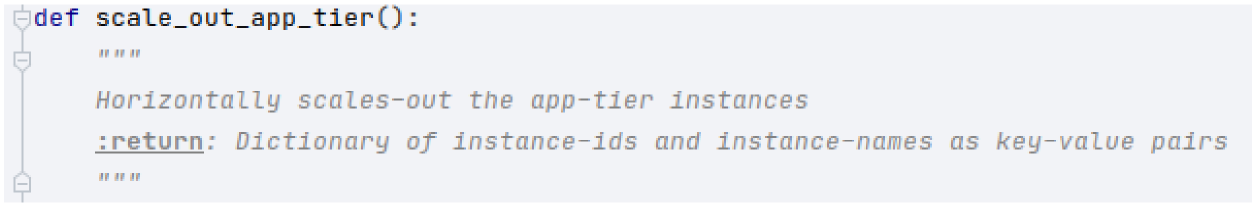
2. autoscaler.py



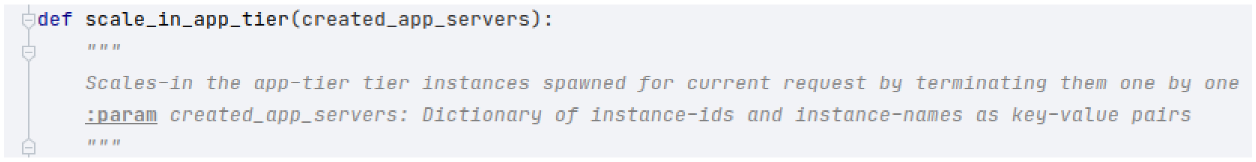
```
def get_total_requests_in_sqs():
    """
    Runs a loop to find the Request queue sizes and takes a Mode over them to get the approximate size of queue
    :return: Size of the request queue in int.
    """
```



```
def get_total_instances_to_create():
    """
    Calculates total App-Tier instances to create for the current request.
    :return: Count of app-tier instances to create
    """
```

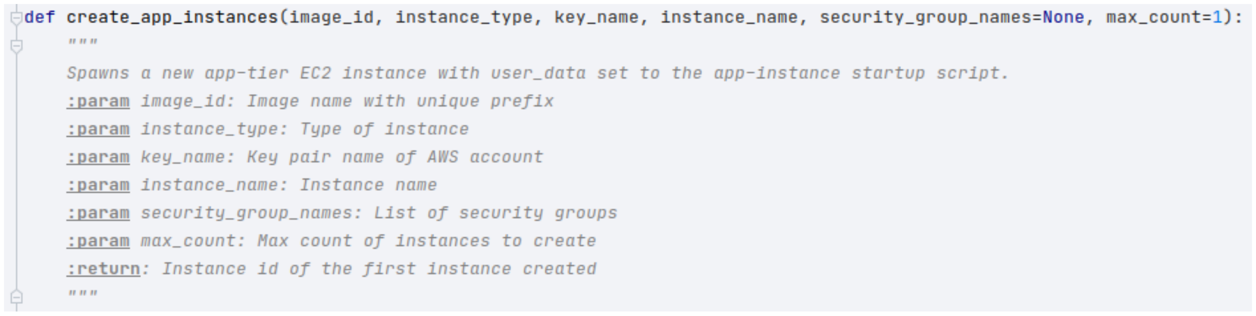


```
def scale_out_app_tier():
    """
    Horizontally scales-out the app-tier instances
    :return: Dictionary of instance-ids and instance-names as key-value pairs
    """
```

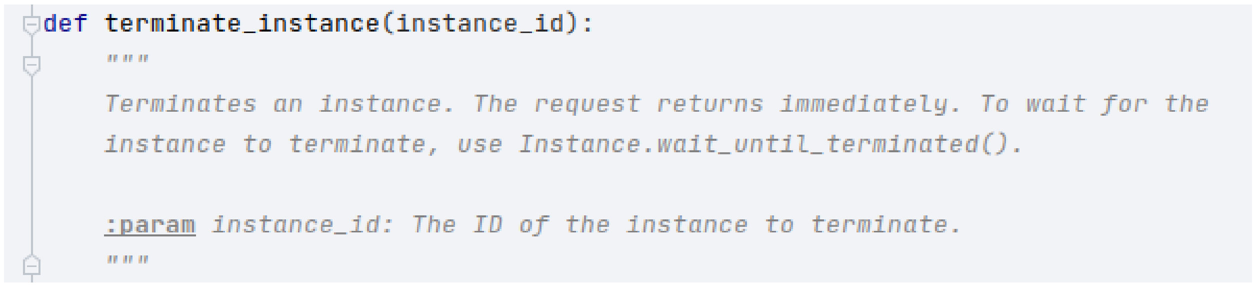


```python
def scale_in_app_tier(created_app_servers):
    """
    Scales-in the app-tier tier instances spawned for current request by terminating them one by one
    :param created_app_servers: Dictionary of instance-ids and instance-names as key-value pairs
    """
```

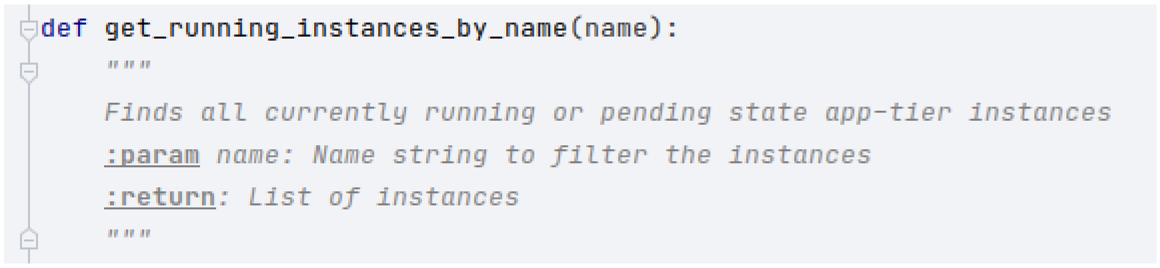
3. ec2_manager.py



```python
def create_app_instances(image_id, instance_type, key_name, instance_name, security_group_names=None, max_count=1):
    """
    Spawns a new app-tier EC2 instance with user_data set to the app-instance startup script.
    :param image_id: Image name with unique prefix
    :param instance_type: Type of instance
    :param key_name: Key pair name of AWS account
    :param instance_name: Instance name
    :param security_group_names: List of security groups
    :param max_count: Max count of instances to create
    :return: Instance id of the first instance created
    """
```

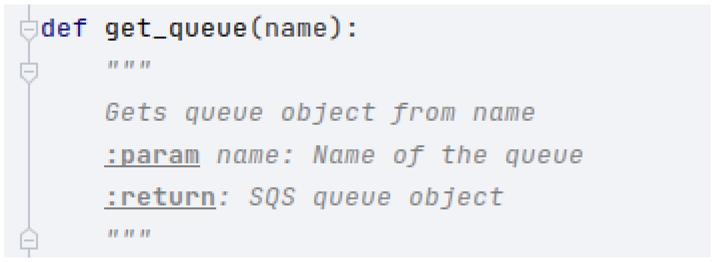


```python
def terminate_instance(instance_id):
    """
    Terminates an instance. The request returns immediately. To wait for the
    instance to terminate, use Instance.wait_until_terminated().

    :param instance_id: The ID of the instance to terminate.
    """
```

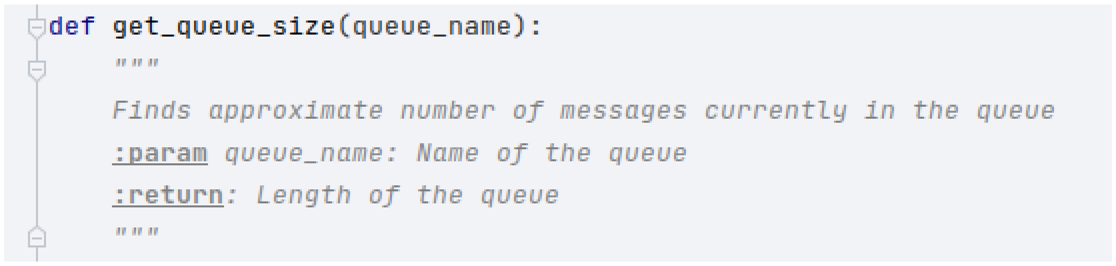


```python
def get_running_instances_by_name(name):
    """
    Finds all currently running or pending state app-tier instances
    :param name: Name string to filter the instances
    :return: List of instances
    """
```

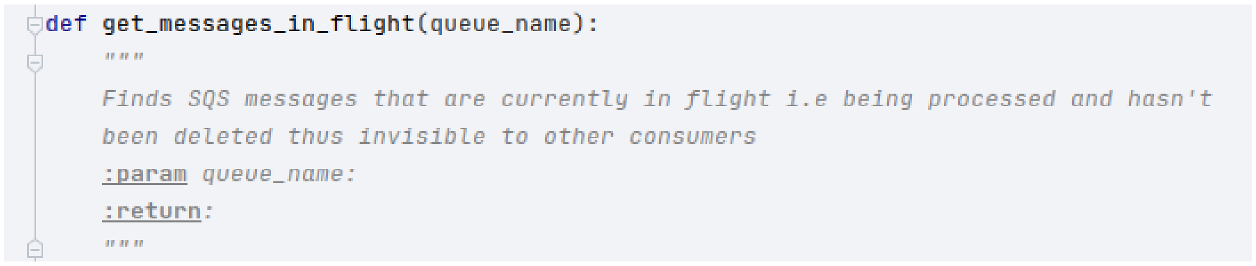
4. msg_queue.py



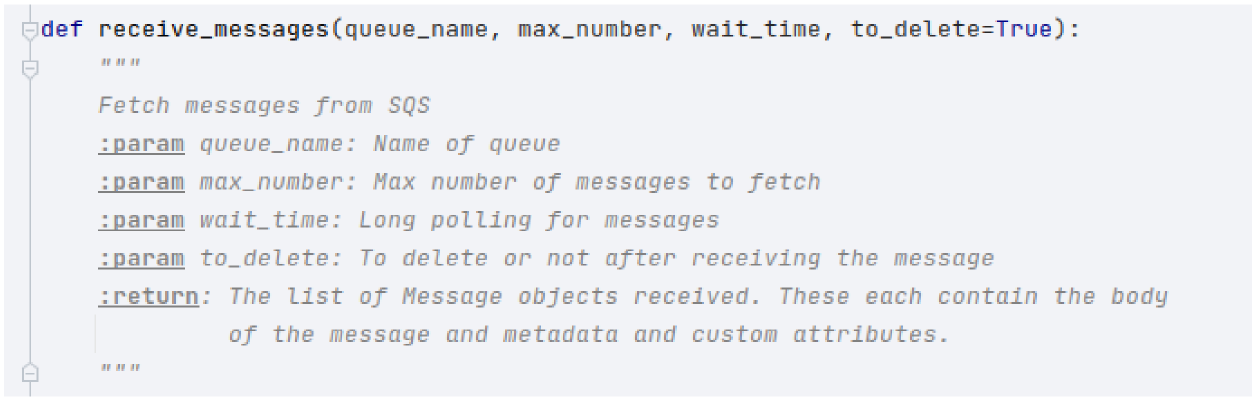
```python
def get_queue(name):
    """
    Gets queue object from name
    :param name: Name of the queue
    :return: SQS queue object
    """
```



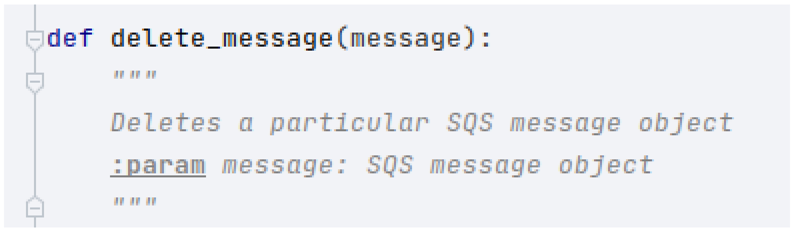
```python
def get_queue_size(queue_name):
    """
    Finds approximate number of messages currently in the queue
    :param queue_name: Name of the queue
    :return: Length of the queue
    """
```



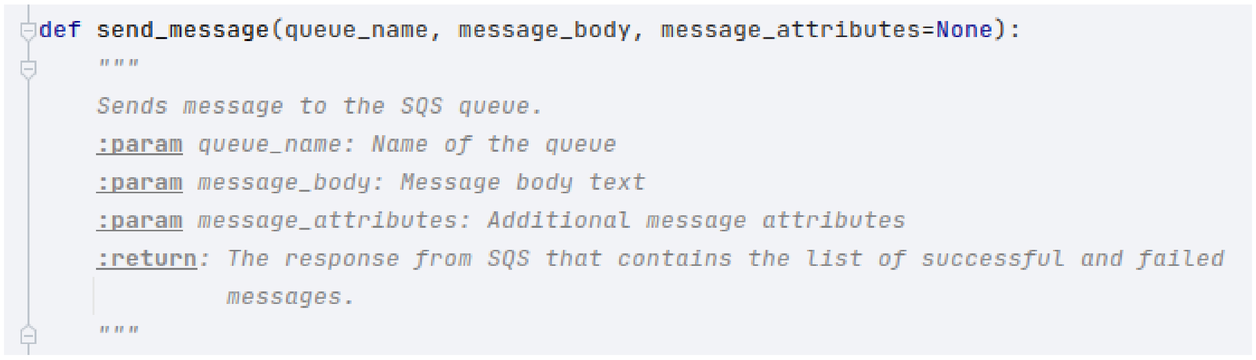
```python
def get_messages_in_flight(queue_name):
    """
    Finds SQS messages that are currently in flight i.e being processed and hasn't
    been deleted thus invisible to other consumers
    :param queue_name:
    :return:
    """
```



```python
def receive_messages(queue_name, max_number, wait_time, to_delete=True):
    """
    Fetch messages from SQS
    :param queue_name: Name of queue
    :param max_number: Max number of messages to fetch
    :param wait_time: Long polling for messages
    :param to_delete: To delete or not after receiving the message
    :return: The list of Message objects received. These each contain the body
             of the message and metadata and custom attributes.
    """
```

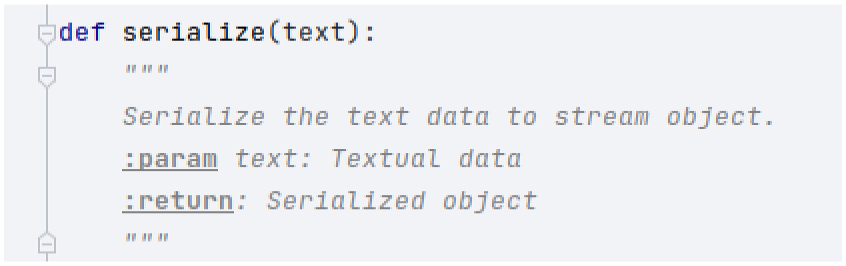


```python
def delete_message(message):
    """
    Deletes a particular SQS message object
    :param message: SQS message object
    """
```

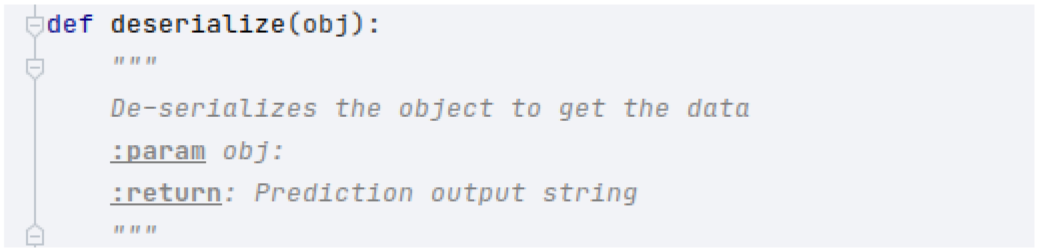


```python
def send_message(queue_name, message_body, message_attributes=None):
    """
    Sends message to the SQS queue.
    :param queue_name: Name of the queue
    :param message_body: Message body text
    :param message_attributes: Additional message attributes
    :return: The response from SQS that contains the list of successful and failed
             messages.
    """
```

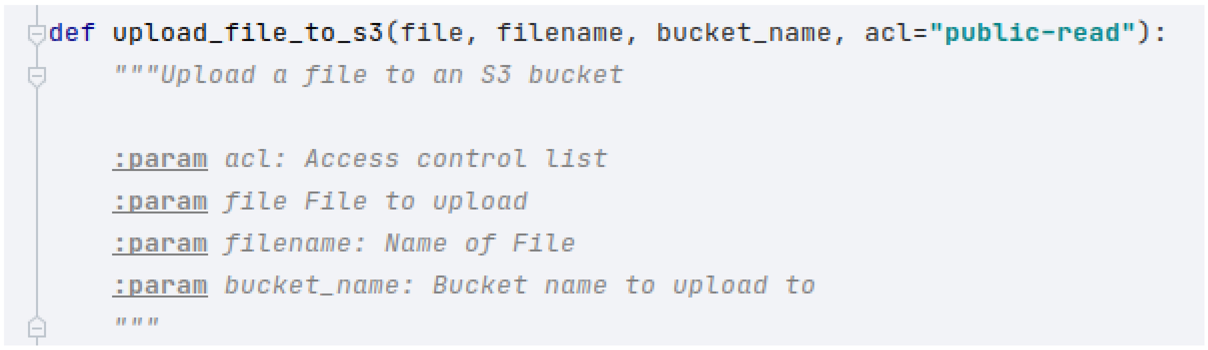
5. s3_manager.py



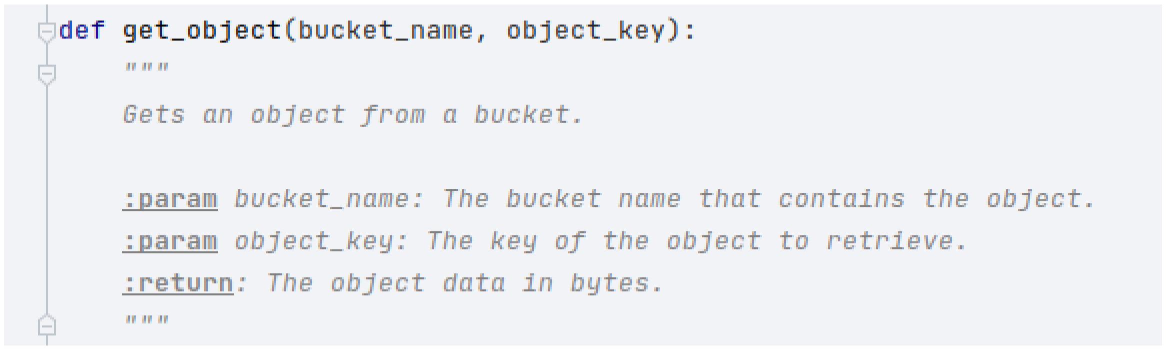
```python
def serialize(text):
    """
    Serialize the text data to stream object.
    :param text: Textual data
    :return: Serialized object
    """
```



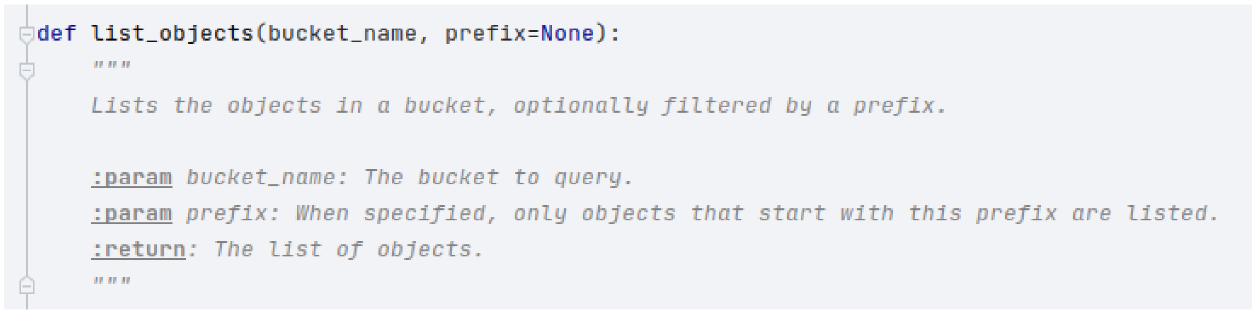
```python
def deserialize(obj):
    """
    De-serializes the object to get the data
    :param obj:
    :return: Prediction output string
    """
```



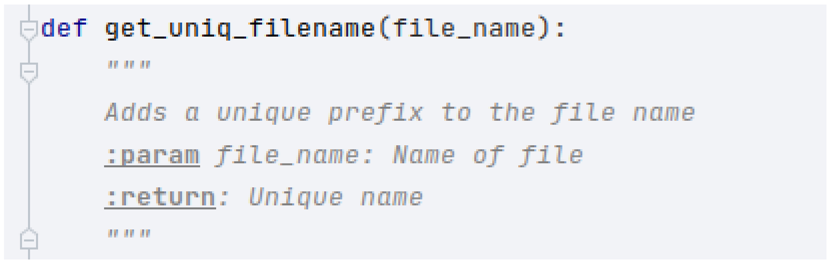
```python
def upload_file_to_s3(file, filename, bucket_name, acl="public-read"):
    """Upload a file to an S3 bucket

    :param acl: Access control list
    :param file File to upload
    :param filename: Name of File
    :param bucket_name: Bucket name to upload to
    """
```



```python
def get_object(bucket_name, object_key):
    """
    Gets an object from a bucket.

    :param bucket_name: The bucket name that contains the object.
    :param object_key: The key of the object to retrieve.
    :return: The object data in bytes.
    """
```



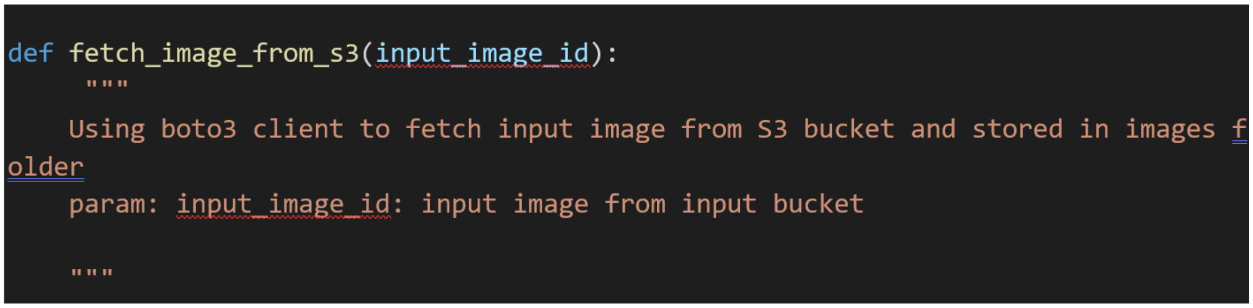
```
def list_objects(bucket_name, prefix=None):
    """
    Lists the objects in a bucket, optionally filtered by a prefix.

    :param bucket_name: The bucket to query.
    :param prefix: When specified, only objects that start with this prefix are listed.
    :return: The list of objects.
    """
```



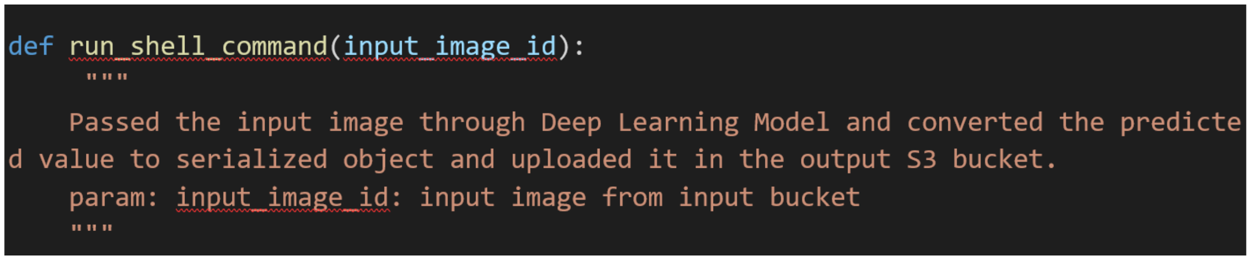
```
def get_uniq_filename(file_name):
    """
    Adds a unique prefix to the file name
    :param file_name: Name of file
    :return: Unique name
    """
```

App-Tier:

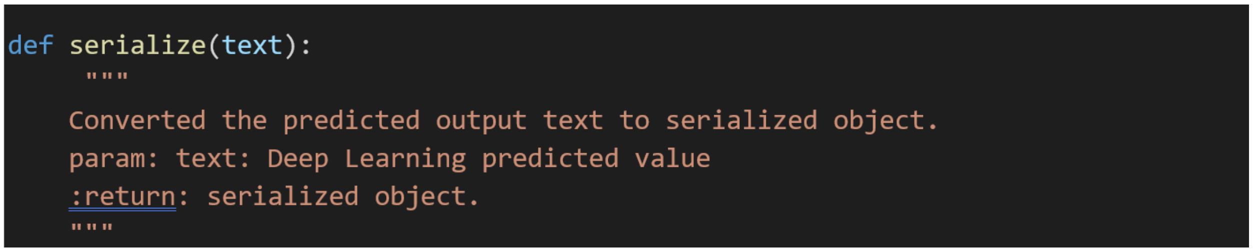
run_poller.py



```
def fetch_image_from_s3(input_image_id):
     """
    Using boto3 client to fetch input image from S3 bucket and stored in images f
older
    param: input_image_id: input image from input bucket

    """
```



```
def run_shell_command(input_image_id):
     """
    Passed the input image through Deep Learning Model and converted the predicte
d value to serialized object and uploaded it in the output S3 bucket.
    param: input_image_id: input image from input bucket
    """
```



```
def serialize(text):
     """
    Converted the predicted output text to serialized object.
    param: text: Deep Learning predicted value
    :return: serialized object.
    """
```

```python
def upload_file(input_image_id, serialized_object):
    """
    uploaded the serialized object to output S3 bucket with input image name as key value.
    param: input_image_id: input image from input bucket
    param: serialized_object: predicted value is converted to serialized object.
    """
```

```python
def fetch_imageid_from_sqs():
    """
    After getting all the messages, using a loop to send input images to fetch images from S3 then delete the message
    """
```

```python
def receive_messages(queue_name, max_number, wait_time, to_delete=True):
    """
    Get messages from the queue and run for loop to check if it needs to be deleted.
    param: queue_name: get queue by name
    param: max_number: Max number of messages.
    param: wait_time: wait time in seconds

    :return: All messages from the queue.
    """
```

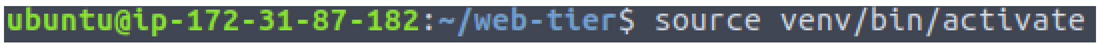
```python
def delete_message(message):
    """
    Delete the message
    param: message: message id that needs to be deleted
    """
```

```python
def get_queue(name):
    """
    Get queue by queue name from sqs resource.
    param: name: queue name
    :return: queue.
    """
```

Steps to Execute:

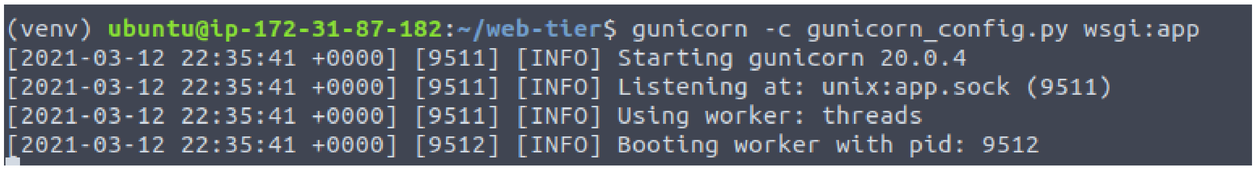
1. Start Web-Tier instance using AMI: ami-02c57bd631cb6b26c
2. SSH into the Web-Tier instance.
3. Goto /home/ubuntu/web-tier directory.

4. Activate virtual env.



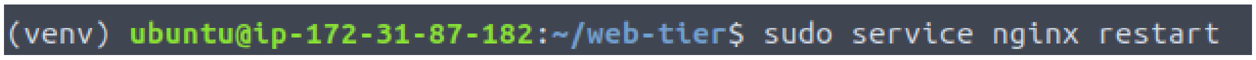
```
ubuntu@ip-172-31-87-182:~/web-tier$ source venv/bin/activate
```

5. Run Gunicorn server



```
(venv) ubuntu@ip-172-31-87-182:~/web-tier$ gunicorn -c gunicorn_config.py wsgi:app
[2021-03-12 22:35:41 +0000] [9511] [INFO] Starting gunicorn 20.0.4
[2021-03-12 22:35:41 +0000] [9511] [INFO] Listening at: unix:app.sock (9511)
[2021-03-12 22:35:41 +0000] [9511] [INFO] Using worker: threads
[2021-03-12 22:35:41 +0000] [9512] [INFO] Booting worker with pid: 9512
```

6. Restart Nginx



```
(venv) ubuntu@ip-172-31-87-182:~/web-tier$ sudo service nginx restart
```

7. Access the website using the public IP address of the web-tier instance.

5. Individual contributions

Sindhu Sree Aita, MSCS student at CIDSE school of Engineering (ASU ID –1220083085)

Design:

I was fully involved in building the application using python flask and using different AWS services like load-balancer, SQS, and end-to-end design including file uploading to input S3 bucket and retrieving predicted output from the output S3 bucket and other utility python modules. I along with the team worked on identifying the pros and cons of two different queues, FIFO queue and Simple queue. We chose to go with the Simple queue by picking mode value of the five requests sent to find the queue length so that we don't get any wrong value from the queue.

Implementation:

1. I initially worked on the development of the frontend for our application using HTML, CSS and python flask. Python flask is the framework we used to construct our project.
2. Implemented the method to uniquely identify the images uploaded by the user and store it into the S3 input bucket since we there will be an issue when two users concurrently upload images with same names so, I used uuid concept to differentiate each image uniquely. When user inputs the image I added a six-digit uuid to name of the image, for example, image "text.png" is given as input by the user the new name generated by the implemented method is "546j31test.png". After the unique name is generated, we rename the image name to the unique name and load it to the input S3 bucket along with the image and send the unique name to the web-tier so that it can process to the SQS request queue.
3. To differentiate between concurrent requests, I implemented a logic to maintain a set with the image names so, that after processing the entire set we can output the result to the user.

Testing:

I worked on the unit testing of uploading the images to the server and creation of input and output buckets into the S3, IAM roles, Security groups and other minor configurations. The unique name generated is also tested by uploading images into the S3 bucket. We did end-to-end testing by sending different number of images at a time and verified the results. We also worked on testing concurrent requests which resulted in expected output without any fail and verified the scale-out and scale-in functionality based on the number of images as finally we noted that out application took about 1min 59s for processing 100images in a single request and on average of 3min for 100 images of two concurrent requests.

Risabh Raj, MCS student at CIDSE school of Engineering (ASU ID- 1219182289)

Design:

I was involved in designing the End-to-End architecture of this project. During the planning phase of the Software Design Life Cycle, I went through the documentation of AWS to figure out the best components to use for the project. We utilized EC2 instances to deploy our application, SQS as a shared request queue among multiple worker app-tier instances and, S3 to store inputs and outputs. While fetching the prediction output, we had two design choices. For single request with multiple images, polling SQS response queue is a better design option, but concurrent requests can cause a race condition among Controller threads. This could increase overall response time considerably. That is why I decided to use output S3 bucket to get prediction results at Controller end rather than using the SQS response queue. Since S3 is an Object storage, to store the prediction result, I decided to use Python Pickle module to serialize the prediction string into object. To give response to user, I implemented a de-serializer at Controller end to convert the object back to the string. For development, I decided to go with Python Flask framework due to its lightweight nature. Then I chose Nginx and Gunicorn to host our web-tier which also acts as the Controller node for the architecture.

Implementation:

- **Input image upload handler:** I developed the input image upload handler which allows multiple images in a single HTTP POST request. I implemented the code to push this image data to Input S3 bucket as well to the SQS Request queue.
- **Autoscaler:** I designed and implemented the Autoscaling logic to scale-out and scale-in the worker app instances based on the number of messages in the request queue. Due to the limitations of AWS Free tier, only a maximum number of 19 App-Tier EC2 instances are allowed for the project. So, I created a configurable autoscaler which calculates the total instances to create using config properties MAX_REQUESTS_PER_INSTANCE and MAX_POSSIBLE_INSTANCES. To keep track of running app instances and assign numbers to new app instances, I used a synchronized PriorityQueue.
- **Output format design:** Since S3 is an Object storage, to store the prediction result, I decided to use Python Pickle module to serialize the prediction string into object. To give response to user, I implemented a de-serializer at Controller end to convert the object back to string.
- **Output Poller:** I developed an S3 output bucket polling mechanism which keeps on running until it fetches all the prediction results of images that came in the HTTP POST request. These prediction results are then sent as a response to the user.
- **Application Deployment:** I have setup the entire web-tier and app-tier deployment. I created the app-tier AMI after installing all the dependencies. I installed and configured Nginx and Gunicorn to deploy the web server. I configured the individual AWS component permissions and access controls.

Testing:

I unit tested every individual component of the project. I wrote a cleanup script to aid in testing which deletes all messages from SQS, all images from S3 buckets and terminates all running EC2 instances. I also ran multiple load tests with concurrent requests each containing 100 images to test the robustness of our application.

Bhavya Swetha Beri, MSCS student at CIDSE school of Engineering (ASU ID – 1221768860)

Design:

I have contributed to designing the project end to end and in the architecture of the system. After several discussions through various models, we came up with the architecture described in the project. I majorly worked on the app-tier of the project. I worked on finalizing the design for connecting app-tier and web tier. I have also worked on the designing the communication between app-tier and other components of AWS like S3 and SQS which are important pieces in completing the entire flow of the application.

Implementation:

I mainly worked on backend by using technologies python, boto3. Worked on building the app-tier which has three main functions. It has a SQS listener which will be constantly polling the messages from SQS queue. When the messages are added to SQS input queue from the web-tier the app-tier starts processing the messages which have the attributes of the images. The message has image-id which will be fetched by the app-tier to get the image form S3 bucket and once the message is processed it is deleted from the queue. The image classifiers run on the retrieved image. Once the classifier completes the processing the classification result is added to output S3 bucket as a serialized object. The classification results are also added to SQS output queue which are later processed by the web-tier to show the results in user interface.

Testing:

I have worked on unit test cases on App-tier and all the connected components. Tested if right input image is fetched from the input S3 bucket. Testing if all the input images in the request queue are added to S3 bucket then those messages are deleted from the queue. I tested whether it is possible to store input duplicate images in images folder. Testing if serialized object is stored in S3 output bucket. We have done end to end testing by going through all test cases that are possible. I have collaborated broadly in stack testing the complete application by sending distinctive number of requests to handle and checked if all components worked as expected.