CSE 546 - Project1 Report

Qiang Fu, Kiran Shanthappa, Sreshta Chowdary Kampally

1. Problem statement

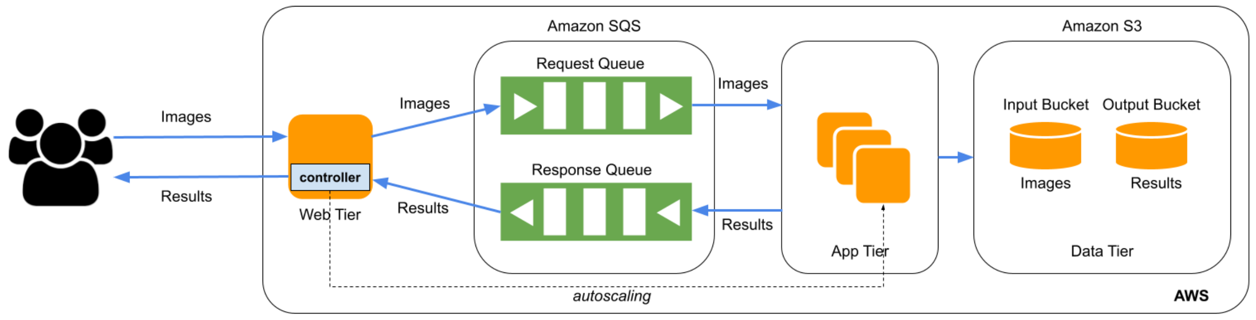
In the modern world, image-recognition is having more and more use cases worldwide, and the usage of machine learning models to perform such a task is a key for this service. However, in many scenarios, the user may not have a device strong enough to run such a ML model or the developers may not want to release the model to the public, in such cases, using cloud computing to run models and provide services can be very beneficial. In our project, we used Amazon Web Services to implement a cloud computing model that is capable of providing simple image recognition services, as well as being able to automatically scale in and out according to the current workload.

2. Design and implementation

2.1 AWS Resources

1. **Amazon EC2**
   Amazon Elastic Cloud Compute (EC2) used for creating the virtual machine of the provided linux image for face recognition at the app-tier.
2. **Amazon S3**
   Amazon Simple Storage Service (S3) is used for storing both input and output data. The interaction to S3 is only from the app-tier.
3. **Amazon SQS**
   Amazon Simple Queue Service (SQS) provides a reliable, scalable and fault tolerant buffer to exchange the messages between app-tier and web-tier.

2.2 Architecture



For this project, we implemented the cloud service according to this architecture shown in the diagram above. We have a web tier instance with a public ip address capable of receiving requests. The web tier instance then processes the request and sends them through Amazon SQS in a request queue. App tier instances keep reading messages from the request queue, process them and store data to the S3 storage, then submit the result through another response queue with SQS. The web tier instance then reads the results from SQS and replies to the user with the result. The web tier instance also has an autoscaling controller that keeps running and creates app instances based on the current demand.

2.3 Autoscaling

We had the auto scaling done in two parts, scaling out and scaling in in our implementation.

We are implementing the scaling out approach by using an autoscaling controller on the web tier instance, running in the background. Here, we check the number of messages in the request queue and the number of app tier instances running. We are using only 19 app tier instances, i.e., the maximum number of app tier instances we can have is 19. If the number of instances is less than 19 and there are still messages in the request queue, we start the required number of instances until we reach 19.

For the scaling in part, it is done automatically on the app tier instances. After processing a message and sending back the output to the response queue, the instance waits for the messages from the request queue, and if it does not receive any message, it self terminates.

3. Testing and evaluation

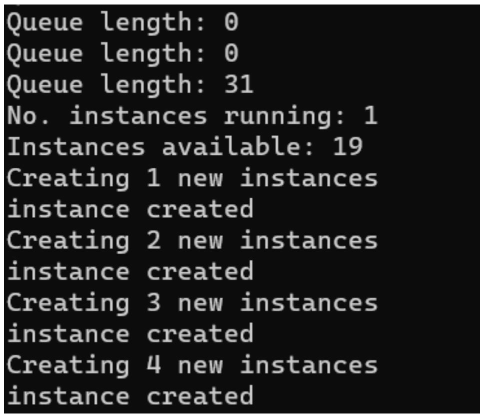
For testing, we used the multiprocessing workload generator provided in the project document, and recorded the response text. We also wrote a script result_checker.py to check if the output matches the provided dataset result. The script can be found in the repository.

The command we used to call the workload generator is as follow:

```
python multithread_workload_generator_verify_results_updated.py --num_request 100 --url "http://35.174.113.237/" --image_folder "./face_images_1000/"
```

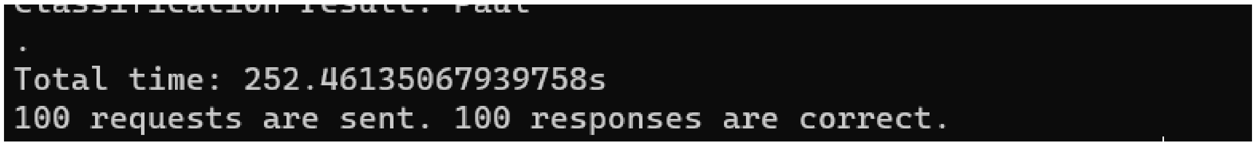
The testing result showed our cloud service handled all the requests correctly, but needs 2 to 5 minutes to start up when there is no instance running initially.

Controller automatically creating instances:



```
Queue length: 0
Queue length: 0
Queue length: 31
No. instances running: 1
Instances available: 19
Creating 1 new instances
instance created
Creating 2 new instances
instance created
Creating 3 new instances
instance created
Creating 4 new instances
instance created
```

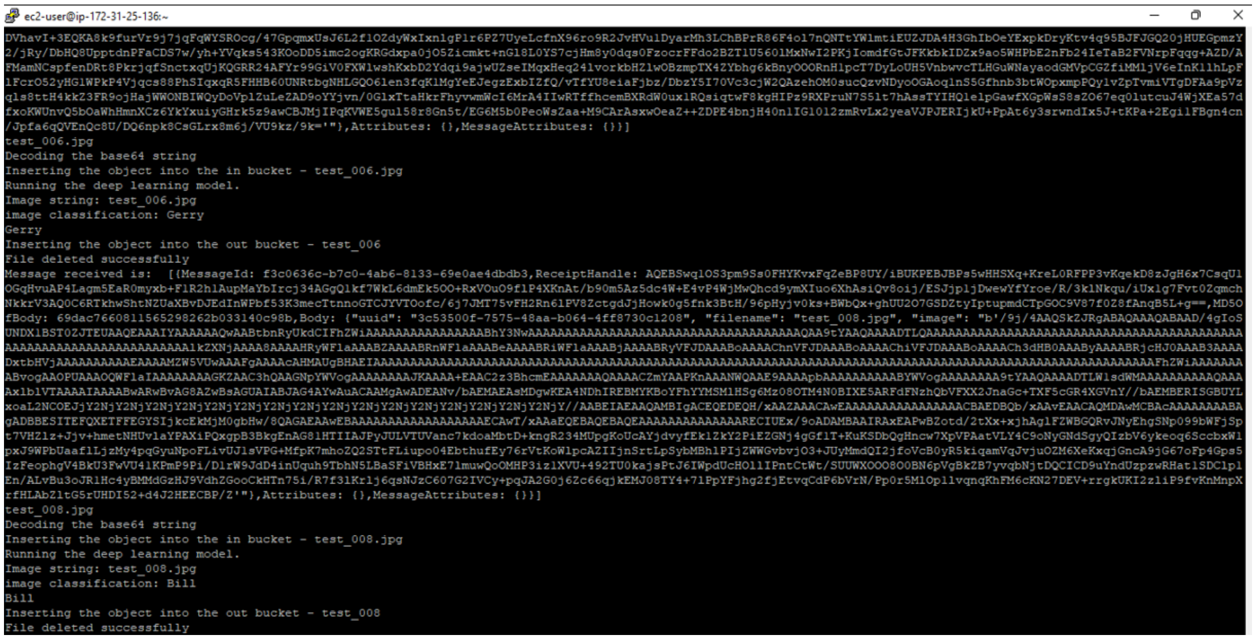
Test Result:



```
Classification result: Paul
.
Total time: 252.46135067939758s
100 requests are sent. 100 responses are correct.
```

If there are no previous requests, we will not have any app tier instances to start with, and will need 2-3 minutes for the auto scaling script to detect the workload, create the instances and wait for the instances to boot up. After the instances are booted and ready for query, we on average can deal with one request within 5-10 seconds and theoretically can deal with up to 1000 requests concurrently.

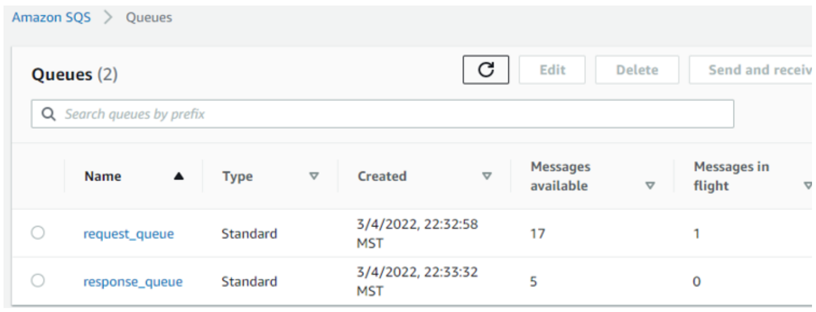
The app tier processing with the results in the SQS and S3 can be seen in below screen capture.



ec2-user@ip-172-31-25-136:~

```
DVhavI+3EQKA8k9furVr9j7jqFqWYSROcg/47GpqmxUsJ6L2flOZdyWxIxnlgPlr6PZ7UyeLcfnX96ro9R2JvHVulDyarMh3LChBPrR86F4ol7nQNTtYWlmtiEUZJDA4H3GhIbOeYExpkDryKtv4q95BJFJGQ20jHUEGpmzY
2/jRy/DbHQ8UpptdnPFaCDS7w/yh+YVqks543KOoDD5imc2ogKRGdxpa0jO5Zicmkt+nGl8L0YS7cjHm8y0dqs0FzocrFFdo2BZTlU560lMxNwI2PKjIomdfGtJFKkbkIDZx9ao5WHPbE2nFb24IeTaB2FVNrpFqqg+AZD/A
FMamNCspfenDRt8PkrjqfSnctxqUjKQGRR24AFYr99GiV0FXWlwshKxbD2Ydqi9ajwUZseIMqxHeq24lvorkbHZlwOBzmpTX4ZYbhg6kBnyOOORnHlpcT7DyLoUH5VnbwvcTLHGuWNayaodGMVpCGZfiMMljV6eInKllhLpF
lFcrO52yHGlWPkP4Vjqcs88PhSIqxqR5FHHB60UNRtbgNHLGQO6len3fqKlMgYeEJegzExbIZfQ/vTfYU8eiaFjbz/DbzY5I70Vc3cjW2QAzehOM0sucQzvNDyoOGAoqlnS5Gfhnb3btWOpxmpPQylvZpTvmiVTgDFAa9pVz
qls8ttH4kkZ3FR9ojHajWWONBIWQyDoVplZuLeZAD9oYYjvn/0GlxTtaHkrFhyvwmWcI6MrA4IIwRTffhcemBXRdW0uxlRQsiqtwF8kgHIPz9RXPruN7S5lt7hAssTYIHQlelpGawfXGpWsS8sZO67eq0lutcuJ4WjXEa57d
fxoKWUnvQ5bOaWhHmnXCz6YkYxuiyGHrk5z9awCBJMjIPqKVWE5gul58r8Gn5t/EG6M5b0PeoWsZaa+M9CArAsxwOeaZ++ZDPE4bnjH40nlIGl0l2zmRvLx2yeaVJPJERIjkU+PpAt6y3srwndIx5J+tKPa+2EgilFBgn4cn
/Jpfa6qQVEnQc8U/DQ6npk8CsGLrx8m6j/VU9kz/9k='"},Attributes: {},MessageAttributes: {}}]
test_006.jpg
Decoding the base64 string
Inserting the object into the in bucket - test_006.jpg
Running the deep learning model.
Image string: test_006.jpg
image classification: Gerry
Gerry
Inserting the object into the out bucket - test_006
File deleted successfully
Message received is:  [{MessageId: f3c0636c-b7c0-4ab6-8133-69e0ae4dbdb3,ReceiptHandle: AQEBSwql0S3pm9Ss0FHYKvxFqZeBP8UY/iBUKPEBJBPs5wHHSXq+KreL0RFPP3vKqekD8zJgH6x7CsqUl
OGqHvuAP4Lagm5EaR0myxb+FlR2hlAupMaYbIrcj34AGgQlkf7WkL6dmEk5OO+RxVOuO9flP4XKnAt/b90m5Az5dc4W+E4vP4WjMwQhcd9ymXIuo6XhAsiQv8oij/ESJjpljDwewYfYroe/R/3klNkqu/iUxlg7Fvt0Zqmch
NkkrV3AQ0C6RTkhwShtNZUaXBvDJEdInWPbf53K3mecTtnnoGTCJYVTOofc/6j7JMT75vFH2Rn6lPV8ZctgdJjHowk0g5fnk3BtH/96pHyjv0ks+BWbQx+ghUU2O7GSDZtyIptupmdCTpGOC9V87f0Z8fAnqB5L+g==,MD5O
fBody: 69dac766081156529826 2b033140c98b,Body: {"uuid": "3c53500f-7575-48aa-b064-4ff8730c1208", "filename": "test_008.jpg", "image": "b'/9j/4AAQSkZJRgABAQAAAQABAAD/4gIoS
UNDXlBST0ZJTEUAAQEAAAIYAAAAAAQwAABtbnRyUkdCIFhZWiAAAAAAAAAAAAAAAAAABhY3NwAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAQAA9tYAAQAAAADTLQAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAlkZXNjAAAA8AAAAHRyWFlaAAABZAAAABRnWFlaAAABeAAAABRiWFlaAAABjAAAABRyVFJDAAABoAAAAChnVFJDAAABoAAAAChiVFJDAAABoAAAACh3dHB0AAAByAAAABRjcHJ0AAAB3AAAA
DxtbHVjAAAAAAAAAAAEAAAAMZW5VUwAAAFgAAAAcAHMAUgBHAEIAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAFhZWiAAAAAAA
ABvogAAOPUAAAOQWFlaIAAAAAAAAAGKZAAC3hQAAGNpYWVogAAAAAAAAJKAAAA+EAAC2z3BhcmEAAAAAAAAQAAAACZmYAAPKnAAANWQAAE9AAAApbAAAAAAAAAAABYWVogAAAAAAAA9tYAAQAAAADTLWlsdWMAAAAAAAAAAQAAA
AxlblVTAAAAIAAAABwARwBvAG8AZwBsAGUAIABJAG4AYwAuACAAMgAwADEANv/bAEMAEAsMDgwKEA4NDhIREBMYKBoYFhYYMSMlHSg6Mz08OTM4N0BIXE5ARFdFNzhQbVFXX2JnaGc+TXF5cGR4XGVnY//bAEMBERISGBUYL
xoaL2NCOEJjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY2NjY//AABEIAEAAQAMBIgACEQEDEQH/xAAZAAACAwEAAAAAAAAAAAAAAAAAACBAEDBQb/xAAvEAACAQMDAwMCBAcAAAAAAAABA
gADBBESITEFQXETFFEGYSIjkcEkMjM0gbHw/8QAGAEAAwEBAAAAAAAAAAAAAAAAAAECAwT/xAAaEQEBAQEBAQEAAAAAAAAAAAAAARECIUEx/9oADAMBAAIRAxEAPwBZotd/2tXx+xjhAglFZWBGQRvJNyEhgSNp099bWFjSp
t7VHZlz+Jjv+hmetNHUvlaYPAXiPQxgpB3BkgEnAG8lHTIIAJPyJULVTUVanc7kdoaMbtD+kngR234MUpgKoUcAYjdvyfEklZkY2PiEZGNj4gGflT+KuKSDbQgHncw7XpVPAatVLY4C9oNyGNdSgyQIzbV6ykeoq6SccbxWl
pxJ9WPbUaaflLjzMy4pqGyuNpoFLivUJlsVPG+MfpK7mhoZQ2STtFLiupo04EbthufEy76rVtKoWlpcAZIIjnSrtLpSybMBhlPIjZWWGvbvjO3+JUyMmdQI2jfoVcB0yR5kiqamVqJvjuOZM6XeKxqjGncA9jG67oFp4Gps5
IzFeophgV4BkU3FwVU4lKPmP9Pi/DlrW9JdD4inUquh9TbhN5LBaSFiVBHxE7lmuwQoOMHP3izlXVU+492TU0kajsPtJ6IWpdUcHOllIPntCtWt/SUUWXOOO8O0BN6pVgBkZB7yvqbNjtDQCICD9uYndUzpzwRHatlSDClpl
En/ALvBu3oJRlHc4yBMMdGzHJ9VdhZGooCkHTn75i/R7f3lKrlj6qsNJzC607G2IVCy+pqJA2G0j6Zc66qjkEMJ08TY4+7lPpYFjhg2fjEtvqCdP6bVrN/Pp0r5MlOpllvqnqKhFM6cKN27DEV+rrgkUKI2zliP9fvKnMnpX
rfHLAbZltG5rUHDI52+d4J2HEECBP/Z'"},Attributes: {},MessageAttributes: {}}]
test_008.jpg
Decoding the base64 string
Inserting the object into the in bucket - test_008.jpg
Running the deep learning model.
Image string: test_008.jpg
image classification: Bill
Bill
Inserting the object into the out bucket - test_008
File deleted successfully
```

Fig. Running of application listener



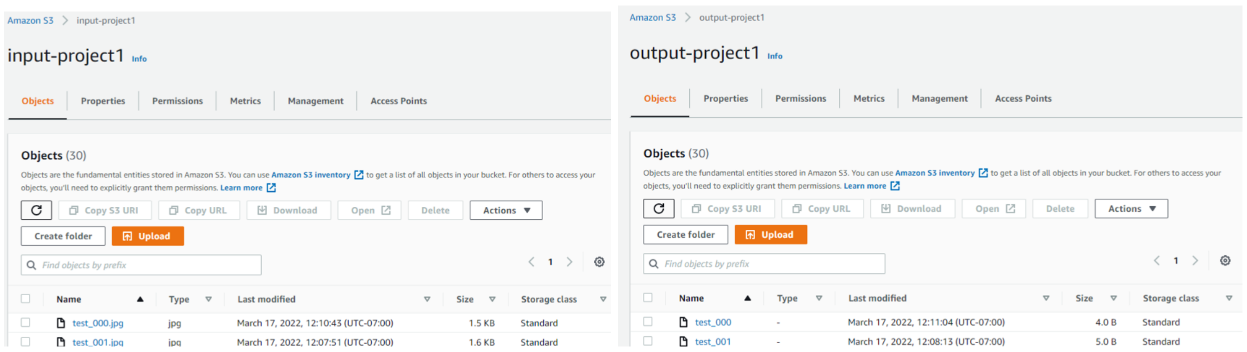
Amazon SQS > Queues

Queues (2)

Name	Type	Created	Messages available	Messages in flight
request_queue	Standard	3/4/2022, 22:32:58 MST	17	1
response_queue	Standard	3/4/2022, 22:33:32 MST	5	0

Fig. Messages in SQS queues during processing

For persistence, we also store the input images and their respective outputs to s3 buckets input_project1 and output_project1, respectively.



Amazon S3 > input-project1

input-project1 Info

Objects (30)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test_000.jpg	jpg	March 17, 2022, 12:10:43 (UTC-07:00)	1.5 KB	Standard
test_001.jpg	jpg	March 17, 2022, 12:07:51 (UTC-07:00)	1.6 KB	Standard

Amazon S3 > output-project1

output-project1 Info

Objects (30)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

Name	Type	Last modified	Size	Storage class
test_000	-	March 17, 2022, 12:11:04 (UTC-07:00)	4.0 B	Standard
test_001	-	March 17, 2022, 12:08:13 (UTC-07:00)	5.0 B	Standard

Fig. S3 input and output buckets for persistence

4. Code

4.1 Modules

multithread_workload_generator_verify_results_updated.py is the script used to generate the workload.

a) App tier:

The app-tier is an auto-scaled instance running the application logic for face recognition. The application uses JAVA Spring Boot framework and is auto-started at the start of the EC2 instance. All the AWS resource access parameters are configured within the constants.java code. The S3, SQS and EC2 resources are accessed through the AWS APIs.

1. App-Instances are launched within the security group, and user-data is passed as parameter by the controller running on the web-tier. A single instance processes a request from the message queue.
2. The application starts with the message queue listener polling for messages in the SQS queue - “request_queue”. The message in the queue contains data of the image sent from web-tier for face recognition processing. Message data is sent in JSON format and contains uuid, filename and image (in base64 string format). Here uuid is used for uniquely identifying a request mapping to response at web-tier. The uuid helps to identify any duplicate responses.
3. The application converts the base64 to byte [ ] and stores it as a temporary .jpg file in the EC2 file system. The image file is sent to the S3 bucket “input-project1” to be stored as type “image/jpg”. The image file in the EC2 is fed to the python deep learning application - “face_recognition.py”. The python code is running from the Java code through the ProcessBuilder libraries and the result is fetched from the BufferInputStream.
4. Result from the python deep learning application is stored in the S3 bucket “output-project1” as type “text/plain”. The result is sent to the web-tier by posting a message to the SQS queue - “response_queue”. The result is sent in JSON format as below:

   ```
   {“uuid”: “<uuid_sent_in_request_message>”, “result”:
   “<face_recognition_result>”}
   ```

5. The temporary image .jpg file in the instance is deleted and followed by the deletion of the request message from the request_queue.

6. As the app instance processes one request after another, this application will loop to look for a new message in the request_queue. If it finds a new message then steps 1-5 are repeated. Otherwise, the app instance scales out by self-terminating the instance.

Initially there will be zero app instances running. Controller brings up the new free tier t2.micro EC2 instances based on the number of messages in the request queue. The new instances use custom AMI with auto start functionality setup through the user-data feature of the EC2. The script autStartrunningListener.sh is run as soon as the app instance is brought up.

During scale-in the app-instance are self terminating. If there are no messages in the request queue then all the app-instances online will be terminated. Hence, when new messages arrive to the request queue, controller has to launch new instances. The option to keep atleast few instances online entire time can done by making certain changes in the listener application code. This can be achieved by changing in the ListenerAndDispatchingServiceImpl.java and method generalFunction() by changing the loop break to continue.

b) Web tier:

/webtier/Create_s3_bucket.py is used initially to create the s3 bucket, it is not used during deployment.

/webtier/Flask-entry.py is the web request controller. It is written in flask, running as a gunicorn server then gets forwarded with nginx to receive and response to public requests to the ip address. It first receives the request, process it and send message to the sqs request queue, then read from sqs response queue and send response back to the user.

Controller.py is the autoscaling controller, it runs in the background, checking the queue periodically and creates new instances as needed.

Prop.py is a helper program for controller.py.

4.2 Installation:

a) App tier setup:

1. The initial EC2 instance is created through the AMI ami-0a55e620aa5c79a24. This image contains the setup to run face_recognition.py. All AWS configuration uses us-east-1.

2. For creating app-tier listener, open the project App-tier appInstance06 submitted on canvas in the eclipse. Build the project through maven install.
3. Use winScp to connect to the EC2 instance and upload the listener-running jar. Also transfer the autoStartrunningListener.sh to ec2-user home directory (/home/ec2-user). Change permission to 776.
4. Install oracle java 8 using wget to download the rpm and perform yum install rpm.
5. Configure the aws by running the command “aws configure”. Setup the access credentials and resource name in the constants.java code.
6. To start the application, run the auto start script - “/home/ec2-user/autoStartrunnningListener.sh”.
7. For creating custom AMI, perform the steps 1-4. Shutdown the instance and create the instance image. All the new instances started will use the custom image (in our case ami-0f25321ba5b9bf7ba)
8. To setup the user-data, send the auto start script as a parameter when starting the instance in the controller. The controller starts a maximum of 19 instances.

There is no manual startup process for app-tier as it is controlled by controller present web-tier.

b) Web tier setup:

For the web tier instance, a ubuntu server is used. The server needs to apt install python3.8, flask, pymemcache, gunicorn and nginx. After installation, a few configurations need to be changed, and their location can be found in README.md. The sample configurations are provided in the config_files folder, with the only thing needed to be changed being the path variables.

To start the local gunicorn server, run the following commands:

```
sudo systemctl daemon-reload
sudo systemctl start flask-entry.service
sudo systemctl enable flask-entry.service
```

To start the autoscaling controller, cd to the script path and run the following:

```
nohup python -u controller.py > program.out 2>&1 &
```

To forward the public traffic with nginx, run the following commands:

```
sudo systemctl daemon-reload
sudo systemctl start nginx
```

```
sudo systemctl enable nginx
```

All the commands above need to be run for the server to work properly.

5. Individual contributions (optional)

Qiang Fu (ASUID:1212546281):

For this project, Qiang is responsible for initially designing the structure, research on the best tools to use, setting up the AWS account, implementing the Web tier instance, and testing/checking the request and results.

During the planning and designing stage, Qiang has been actively involved in the designing of the overall architecture of the service, and has planned some of the timeline of the project. After some discussions, Qiang has planned the languages and tools to use for the implementation of web tier instances, including: Python, Flask, Gunicorn and Nginx.

Qiang also did the initial setup for the AWS accounts, IAM accounts/authentication for the team and TA, and security policy management, actively communicating to ensure every team member had access to the project development.

During the implementation stage, Qiang is responsible for implementing the request controller on the web tier instance, as well as setting up the server and configurations.

First, Qiang created the request controller script, and tested it as a local flask development server to ensure the functionality of the controller.

There is a problem with the flask development server however, the simple server is for testing only and can only handle one request at a time with no support for multithreading, therefore it can’t be used for the actual implementation. To solve this problem, Qiang has set up the server using gunicorn, a strong WSGI server capable of handling real-life level of requests. This creates another two issues related to the multithreading: First, gunicorn itself is a local server and does not have the public traffic routing to it. Second, because of the nature of SQS, it is impossible to get the exact message, therefore a thread may not get the correct response to its original request, and needs a way to share the data between threads, which is not possible in flask. To solve the first problem, Qiang has set up a nginx service which acts as a reverse proxy, routing the public request to the local gunicorn server, and the gunicorn response back to the public request. Qiang also tuned the config for both gunicorn and nginx to set up the proxy, and extend the timeout limit to avoid failure when the app instance takes too long to start up. For the second problem, Qiang used pymemcache, which acts as a global accessible thread-safe cache/database server, to share information between different threads, ensuring each requests will get their corresponding response.

For testing, Qiang runs the workload generator and wrote result_checker.py to check between the printed result text and the correct result provided.

Kiran Shanthappa(ASU ID: 1222119418):

The design step included choosing which Amazon Web Services resources to employ, where each of the different components will reside in the architecture, how they will communicate with one another, and where the deep learning image recognition module will be called, among other things. Kiran worked with the rest of the team to come up with the project's design and architecture. I worked on finalizing the image propagation as base64 string and JSON format for queue messages. In both controller and App-tier apps, decided on the organization of the various modules. For the App-tier design the based on the AWS SDK in Java and decided to develop using spring boot framework.

Kiran worked on the App-Tier implementations of Spring Boot apps. The load balancer was created as a distinct process in the Web-tier application, but the load balancing logic was discussed and determined with the entire team. Kiran used the Amazon Web Services APIs in JAVA to implement operations on SQS such as sending a message, deleting a message, receiving a message, and getting the approximate number of messages in the queue, as well as operations on S3 such as creating a bucket, writing to the bucket, and EC2 instance creation, termination, and checking the number of instances running. Used process builder for invoking the python deep learning code. Creating the AMI with auto start of the listener through the user-data. Configuring the visibility timeout.

Unit testing, integration testing, and end-to-end testing were all part of the testing phase. It also entailed executing the scripts that had been handed to us. The application running in the app-instances and AMI were subjected to unit and integration testing by Kiran. While implementing and checking the validity of the application and modules, Kiran had to do a lot of manual testing using custom scripts to post the messages to the queue.

This project has provided a tremendous opportunity to learn the various services of the AWS where I worked majorly such as EC2, S3, and SQS. The theoretical knowledge from the class was applied into the project to develop a successful app-tier. Also the project helped to understand dependencies such as AMI creation, auto start script, user-data, EBS volumes, security groups, IAM roles, and cloud monitor to complete this project successfully. Learned the different connection and configuration options such AWS Cli, AWS APIs in java/python and through the web management console.

Sreshta Chowdary Kampally(ASUID:1224031900):

Design:

1. I actively collaborated with my teammates in deciding a few changes in the architecture. We decided to go with an approach regarding the scaling out and scaling in the instances. Instead of stopping the instances, we are terminating them.
2. My main contribution in this project is designing and implementing the autoscaling controller. We decided to run the controller in the background at 30-second intervals.

Implementation:

1. Setup of SQS queues and S3 buckets required for the project:
    a. Created standard queues and fixed the visibility timeout of request queue to 30 seconds
2. Implementation of autoscaler code:
    a. Find the visible messages in the request queue.
    b. Find the number of instances that are running or are pending.
    c. Evaluate the intervals in which the autoscaler should run and fix it to 30 seconds. That is, the autoscaler runs every 30 seconds.

Testing:

1. I was actively involved in the unit testing of controller.py and modifying the parameters to ensure it was the best efficient value. For example, after trying various intervals, we decided to run the code every 30 seconds since this value was more efficient than the others.
2. We tried uploading different numbers of images to verify the correctness of our code.

Tools and Services used:

1. Libraries: used ‘Boto3’, an AWS SDK for Python
2. AWS Services: EC2,SQS,S3
3. Putty: To check the implementation of the controller code and debug it on windows.

Takeaway:

From this project, I learned about the various AWS services and practically implemented the autoscaling logic. I also got myself familiar with Boto3 and how we can create stop ec2 instances and implement many other functions in EC2, SQS, and S3 using Boto3. I also got an opportunity to learn about other tools used in this project.