CSE 546 — Project 1 Report

Harsh Patel (1221434273),
Jagdeesh Basavaraju (1213004713),
Shiksha Patel (1211045628),
Suraj S Kattige (1211230381)

1. Architecture

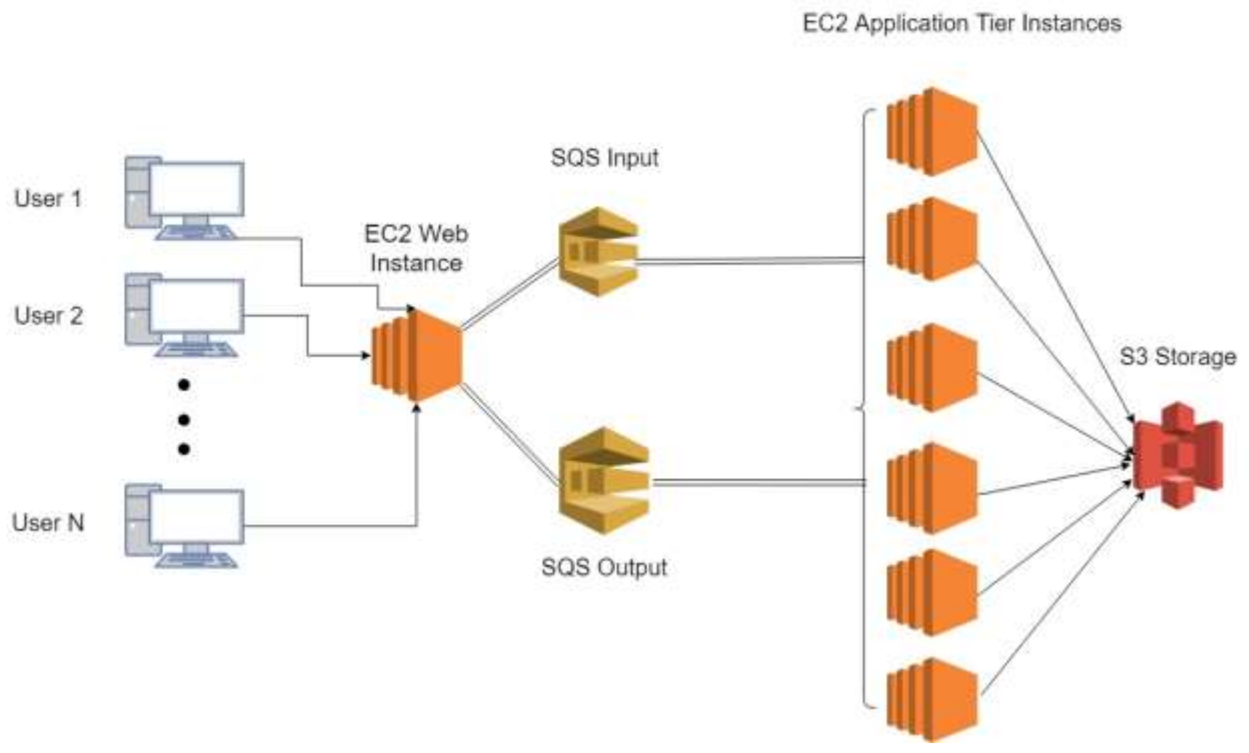


Figure 1.1 Architecture

1.1 AWS Services used in the project:

1. **AWS EC2**
   EC2 allows users to create their virtual machine in Amazon cloud for running any application. In our project, we are using EC2 (Linux images with deep learning application environment) for implementing our application at Web Tier and App Tier.
2. **AWS SQS**
   We are using standard SQS (loosely coupled FIFO) for outgoing and incoming messages in our architecture. SQS makes it easy to scale and decouple microservices.
3. **AWS S3**
   We are using S3 for storing result from App instances for persistency. S3 supports scalable storage through application programming interfaces. We can store in buckets with (key, value) object format with any kind of file.

1.2 Frameworks used:

We have used JAVA Spring Boot framework for our REST application at web tier and App Tier. We have shell scripts for running our logic for auto-start and termination of App Instances.

1.3 Architecture Explanation

Our architecture employs an EC2 Web tier instance (1 instance) that takes the requests from users in the system. These requests will be pipelined into an SQS which will be fed into application tier EC2 instances (19 max,1 min). These instances will be scaled in and out through the web tier instance. Load balancing algorithm will be used to decide whether the additional application tier instances will be required. We can have a maximum of 19 application tier instances for scaling out. Application tier instances will run deep learning application and take image url as input from SQS and produce predicted output label for the image. These requests and its subsequent results will be stored on a S3 storage for persistency.
Initially we have 1 web tier and 1 App instance running to handle quick response. Web tier takes care of starting the other App instances as per demand. Scaling in is handled by App instances themselves as they auto terminate when they do not find messages in SQS.

Once the output has been obtained, they will be fed into another SQS output queue which would push these results to the EC2 web instance. We have designed a hashtable on the web instance so that the SQS output can quickly push all the results into it. When the user requests for the result of their subsequent request, the output will be pushed from the hashtable to the user and deleted from the hashtable. This simplifies the architecture and improves the overall efficiency of the architecture.

1.4 Fault Tolerance

At App Tier: Web tier keeps message in SQS from where the App Tier picks up the image url. AWS SQS provides nice feature of visibility timeout which makes a message invisible to other App EC2 instances when one App EC2 instance is reading from the SQS.

We have set visibility timeout value taking into consideration maximum time an App EC2 instance can take to run a deep learning application. So, if anything bad happens to current App EC2 instance and it terminates, then after visibility timeout, that message in SQS will be visible to other App EC2 instances as well. If current App EC2 instances runs successfully the deep learning model, it deletes message from SQS. Visibility timeout feature provides contention avoidance as well as fault tolerance.

At Web Tier: At web tier, if request load increases beyond a certain capacity, we can bring in another Web tier instance to handle the additional user requests. But in our project, due to resource constraint, we implemented only one Web Tier which ensures response to each user request.

1.4 How to ensure minimum SQS requests?

AWS SQS (free tier) can put limitation on number of API requests.
SQS provides a feature called “Receive Message Wait Time” which we are using for ensuring minimum SQS requests. Receive Message Wait Time ensures maximum amount of time which a long polling receive call to SQS will wait for a message to become available before returning an empty response. This feature gives leverage to listen to SQS queue for a longer time without making continuous requests to SQS.

2. Autoscaling

Scaling in and scaling out are done at the application tier and web tier respectively. When the web tier instance gets a new request, it puts it into the input SQS as shown in figure 1.1. Most difficult part for the autoscaling was to create the environment which is thread-safe. As in our web application there can be concurrent request served by different threads and they individually cannot do auto scaling as they don’t share memory, so keeping shared variable who keeps track of running instance is not possible. For that we used the parallel main thread which runs along with other java threads in web-tier.

In this parallel thread we have implemented the Scale out logic using the two inputs, number of running instances and approximate number of visible SQS messages. First thing that we need to make sure is that scale out logic does not try to create the instance beyond number 20. The aim of our logic is to match the number of running instances and the number of visible SQS messages, if we can do so in the given maximum 20 app instances. If number of SQS messages is higher than the 20, then we are not going to match with the number of the instances, but we will create the instances so that overall 20 app instances will be running. This is all about the scale out logic that we implemented in the Web Tier. We can say that it is almost like the greedy solution to provide the service to users as fast as possible. Here in our custom AMI which we are using to create the app instances have the capability of running the java application using shell script every time it boots up. Then app instance will start doing the work that it needs to do.

Scaling in works at the application tier. After an application tier instance predicts the output a given request, it feeds this output to the SQS output and put the key-value object pair in the S3 bucket. After completing the one request that was in SQS, it again looks for the message in the SQS. If it gets the message, then it will do whole deep learning process again. If there are no messages in the queue it instantly comes out of the loop and terminates itself, which scales in the instances. We can also use one of the SQS services which is the “Wait time”. If we give wait time as 20 seconds then receive message request from app instance will wait for 20 seconds looking for the messages in the SQS in single request, but as per the project instructions we kept it 0 wait time. So, in a nutshell, app instances will run in loop till they are able to find some request in the SQS, and if not, then it automatically terminate itself or overall scale in our whole app instances setup. But for one of the app-instance that will continuously be running to serve the requests, we have kept the wait time as 20 seconds to do continuous polling without requiring many requests.

3. Code

3.1 Code Modules

a) web-Tier-AWS module

a.1) Rest Controller (project1Controller.java): This controller maps the http request to the function inputURL present in ImageURLService.

a.2) Send message to the input queue using SQSService: This service uses AWS APIs to place the message in the input queue.

a.3) Now, the control goes to getFromHashOrQueue function. This function uses a static hashtable which store the image name and the result. So, the input URL will be first checked in hashtable and if there is a match, the entry from the hashtable will be removed and result will be returned. If there is no match, then the message will be queried from the output queue and checked against the input URL and if it is a match, then it will delete the entry from the OutputQueue and return the result. If not a match, then it will just write into the hashtable and delete the message in the queue.

a.4) Return the result from the getFromHashOrQueue function to the controller class which will in turn return as a http response to the user.

b) ListenerAndDispatchingService module

b.1) This module is executed by the individual App instances.

b.2) It will receive the message from the input queue and if the message is not available then the App instance will terminate itself. If it receives a message from the input SQS queue, it will provide the message (i.e image URL) to the deep learning module for classification. A process builder will execute the deep learning computation instructions.

b.3) The classification result will be written to the S3 storage in bucket using AWS S3 APIs.

b.4) The classification result will also be sent to the output SQS queue. After that message is deleted from the input SQS queue.

c) LoadBalancerService module

c.1) This is a module running in one of the threads of web-tier application.

c.2) This module queries input SQS queue for the approximate number of messages.

c.3) It also queries EC2 for the number of running instances at a given point in time.

c.4) Based on the above two query results, appropriate number of instances will be spawned by this loadbalancer. This ensures proper scale out of the App instances. Scaling in is taken care by the individual App instances.

3.2 Instructions for Project Execution

3.2.1 Setup

a). Setup of Web-tier-EC2 (create only one web-tier instance)

a.1) Follow instructions from AWS Instruction pdf shared on project group, while doing aws configure - set default location as us-west-1, we have used ami-07303b67 mentioned in project.pdf for creating instance for web-tier.

a.2) Create a jar file of web-tier application submitted in blackboard in your local machine using maven build.

a.3) Now transfer this web-tier jar to web-tier ec2 instance which was created earlier and give file permission (chmod +x jar name).

a.4) Create the java environment by installing default-jre (jdk) packages in Web tier EC2 instance
scp -i security_key.pem /path_of_webtier_jar ubuntu@ec2-public_dns:~

b) Setup of Application Tier-EC2

b.1) Follow instructions from AWS Instruction pdf shared on project group, while doing AWS configure-set default location as us-west-1, we have used ami-07303b67 mentioned in project.pdf for creating instance for app tier.

b.2) Create a jar file of listener-running application submitted in blackboard in your local machine using eclipse maven build.

b.3) Now transfer this listener-running jar to app ec2 instance which was created earlier and give file permission (chmod +x jar name)
scp command: scp -i security_key.pem /path_of_listner_jar ubuntu@ec2-public_dns:~

b.4) Create the java environment by installing default-jre (jdk) packages in App instance

b.5) Run the jar: java -jar /path/jar_file_name (this will start App instance which will always be in running state)

b.6) For creating ami image for other instances, follow steps from b.1 to b.4 with the listener application 2 submitted on blackboard.

b.7) Transfer autoStartListener.sh from your local machine to ec2 app instance at specific path:
scp command: scp -i security_key.pem /path_of_autoStartListener.sh ubuntu@ec2-public_dns:/home/ubuntu/var/lib/cloud/scripts/per-boot/

Give file permission: chmod +x file

b.8) Now, create an AMI image of the listener application EC2 instance. This will create one snapshot of the same size as ec2 app instance with the required environment. Now this will be the base AMI for all other Listener App EC2 instances later (we are using this created AMI for 18 App instances in our code).

Note: If you want to change credentials (such as access key, you should change the values of those credentials in constants.java in the code module)

3.2.2 Execution Steps

a.1) First we need to start web-tier application so that it can respond to http requests from user. Web tier can be started automatically or manually. To see the debug log, we do it manually as:

```
Java -jar jar_file_name
```

Then the web tier server is up.

a.2) On your local machine, do aws configure before running any test script.

a.3) Now you can see that, atleast one App instances will be already running.

a.4) Now as you start running test.sh and other test scripts, you can see the output response back on the console and result stored in S3 as well.

4. Project status

We have implemented and met all the project requirements.

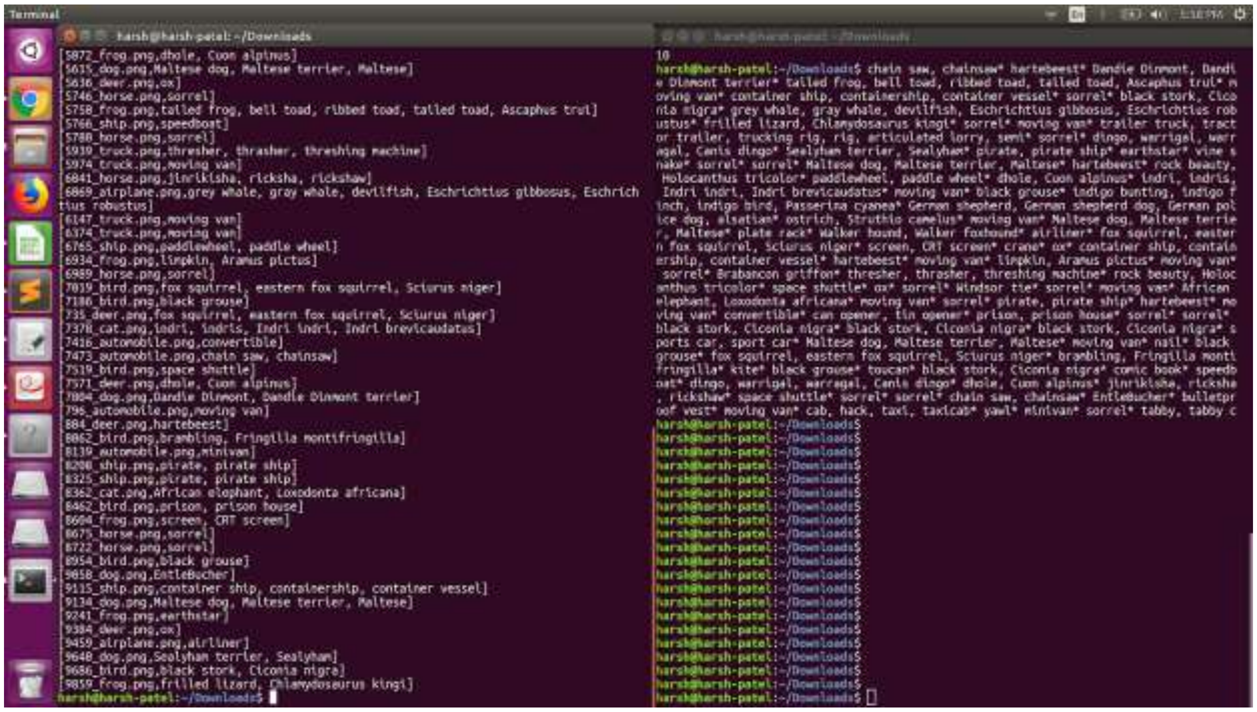
a.1) Web-tier can accept multiple http requests, send it to App tier and fetch the results back and send responses to the user.

a.2) We have implemented Autoscaling and Load balancing successfully.

a.3) For persistency, you can see the unique output stored in S3.

Screenshots of output of test scripts:

b.1) Figure 4.1 shows the output when test.sh (with 10 as concurrent requests and 100 as total requests, right half image) is run and list_data.sh (first half image) is run. As you can see unique responses in the right half image. We put * to make it distinguishable on screen (we have removed * from our code for the actual submission).

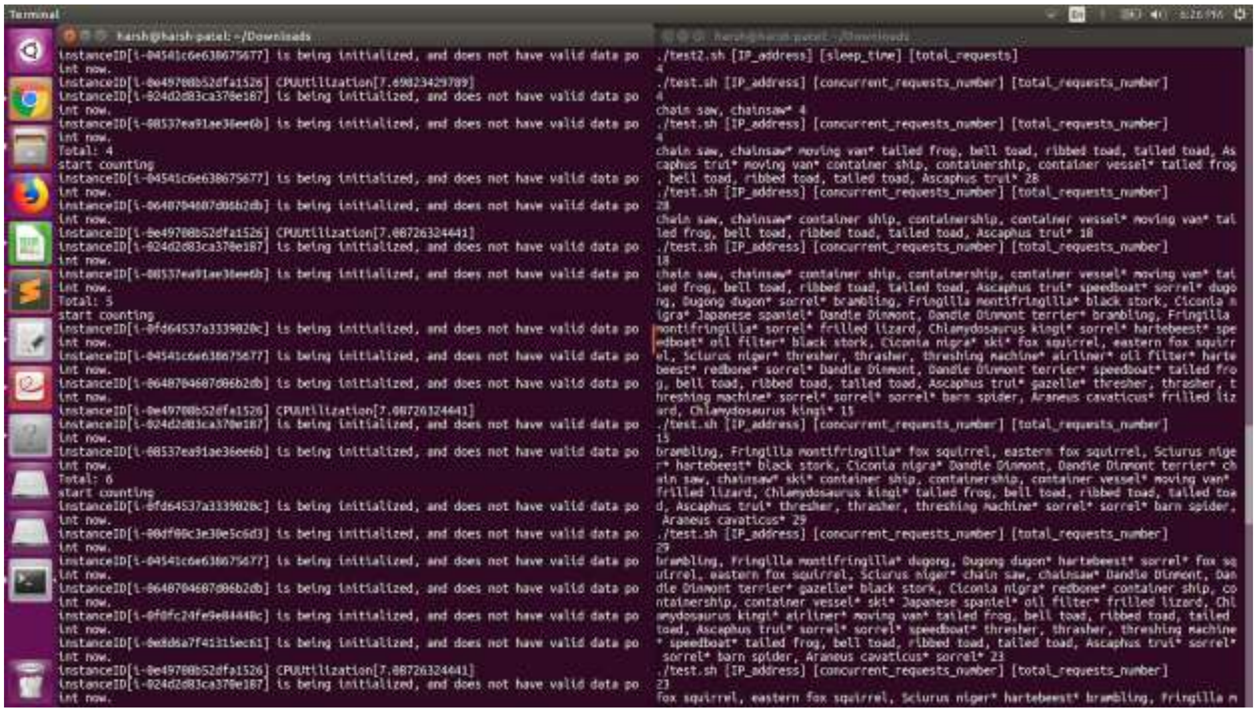


```
[5872_frog.png,dhole, Cuon alpinus]
[5615_dog.png,Maltese dog, Maltese terrier, Maltese]
[5636_deer.png,ox]
[5746_horse.png,sorrel]
[5758_frog.png,tailed frog, bell toad, ribbed toad, tailed toad, Ascaphus trui]
[5766_ship.png,speedboat]
[5788_horse.png,sorrel]
[5939_truck.png,thresher, thrasher, threshing machine]
[5974_truck.png,moving van]
[6841_horse.png,jinrikisha, ricksha, rickshaw]
[6869_airplane.png,grey whale, gray whale, devilfish, Eschrichtius gibbosus, Eschrich
tius robustus]
[6147_truck.png,moving van]
[6374_truck.png,moving van]
[6765_ship.png,paddlewheel, paddle wheel]
[6934_frog.png,limpkin, Aramus pictus]
[6989_horse.png,sorrel]
[7019_bird.png,fox squirrel, eastern fox squirrel, Sciurus niger]
[7186_bird.png,black grouse]
[735_deer.png,fox squirrel, eastern fox squirrel, Sciurus niger]
[7378_cat.png,indri, indris, Indri indri, Indri brevicaudatus]
[7416_automobile.png,convertible]
[7473_automobile.png,chain saw, chainsaw]
[7519_bird.png,space shuttle]
[7571_deer.png,dhole, Cuon alpinus]
[7804_dog.png,Dandie Dinmont, Dandie Dinmont terrier]
[796_automobile.png,moving van]
[884_deer.png,hartebeest]
[8062_bird.png,brambling, Fringilla montifringilla]
[8139_automobile.png,minivan]
[8208_ship.png,pirate, pirate ship]
[8325_ship.png,pirate, pirate ship]
[8362_cat.png,African elephant, Loxodonta africana]
[8462_bird.png,prison, prison house]
[8604_frog.png,screen, CRT screen]
[8675_horse.png,sorrel]
[8722_horse.png,sorrel]
[8954_bird.png,black grouse]
[9058_dog.png,EntleBucher]
[9115_ship.png,container ship, containership, container vessel]
[9134_dog.png,Maltese dog, Maltese terrier, Maltese]
[9241_frog.png,earthstar]
[9384_deer.png,ox]
[9459_airplane.png,airliner]
[9648_dog.png,Sealyham terrier, Sealyham]
[9686_bird.png,black stork, Ciconia nigra]
[9859_frog.png,frilled lizard, Chlamydosaurus kingi]
harsh@harsh-patel:~/Downloads$
```

```
10
harsh@harsh-patel:~/Downloads$ chain saw, chainsaw* hartebeest* Dandie Dinmont, Dandi
e Dinmont terrier* tailed frog, bell toad, ribbed toad, tailed toad, Ascaphus trui* m
oving van* container ship, containership, container vessel* sorrel* black stork, Cico
nia nigra* grey whale, gray whale, devilfish, Eschrichtius gibbosus, Eschrichtius rob
ustus* frilled lizard, Chlamydosaurus kingi* sorrel* moving van* trailer truck, tract
or trailer, trucking rig, rig, articulated lorry, semi* sorrel* dingo, warrigal, warr
agal, Canis dingo* Sealyham terrier, Sealyham* pirate, pirate ship* earthstar* vine s
nake* sorrel* sorrel* Maltese dog, Maltese terrier, Maltese* hartebeest* rock beauty,
 Holocanthus tricolor* paddlewheel, paddle wheel* dhole, Cuon alpinus* indri, indris,
 Indri indri, Indri brevicaudatus* moving van* black grouse* indigo bunting, indigo f
inch, indigo bird, Passerina cyanea* German shepherd, German shepherd dog, German pol
ice dog, alsatian* ostrich, Struthio camelus* moving van* Maltese dog, Maltese terrie
r, Maltese* plate rack* Walker hound, Walker foxhound* airliner* fox squirrel, easter
n fox squirrel, Sciurus niger* screen, CRT screen* crane* ox* container ship, contain
ership, container vessel* hartebeest* moving van* limpkin, Aramus pictus* moving van*
 sorrel* Brabancon griffon* thresher, thrasher, threshing machine* rock beauty, Holoc
anthus tricolor* space shuttle* ox* sorrel* Windsor tie* sorrel* moving van* African
elephant, Loxodonta africana* moving van* sorrel* pirate, pirate ship* hartebeest* mo
ving van* convertible* can opener, tin opener* prison, prison house* sorrel* sorrel*
black stork, Ciconia nigra* black stork, Ciconia nigra* black stork, Ciconia nigra* s
ports car, sport car* Maltese dog, Maltese terrier, Maltese* moving van* nail* black
grouse* fox squirrel, eastern fox squirrel, Sciurus niger* brambling, Fringilla monti
fringilla* kite* black grouse* toucan* black stork, Ciconia nigra* comic book* speedb
oat* dingo, warrigal, warragal, Canis dingo* dhole, Cuon alpinus* jinrikisha, ricksha
, rickshaw* space shuttle* sorrel* sorrel* chain saw, chainsaw* EntleBucher* bulletpr
oof vest* moving van* cab, hack, taxi, taxicab* yawl* minivan* sorrel* tabby, tabby c
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
harsh@harsh-patel:~/Downloads$
```

Figure 4.1

b.2) Figure 4.2 shows the output of test2.sh (sleep time given 20 seconds, total requests 200, second half image) and list_instances.sh (first half image). During sleep time, we could see the minimum number of instances as 2.



```
instanceID[i-04541c6e638675677] is being initialized, and does not have valid data po
int now.
instanceID[i-0e49708b52dfa1526] CPUUtilization[7.69823429789]
instanceID[i-024d2d83ca370e187] is being initialized, and does not have valid data po
int now.
instanceID[i-08537ea91ae36ee6b] is being initialized, and does not have valid data po
int now.
Total: 4
start counting
instanceID[i-04541c6e638675677] is being initialized, and does not have valid data po
int now.
instanceID[i-0648704687d86b2db] is being initialized, and does not have valid data po
int now.
instanceID[i-0e49708b52dfa1526] CPUUtilization[7.08726324441]
instanceID[i-024d2d83ca370e187] is being initialized, and does not have valid data po
int now.
instanceID[i-08537ea91ae36ee6b] is being initialized, and does not have valid data po
int now.
Total: 5
start counting
instanceID[i-0fd64537a3339828c] is being initialized, and does not have valid data po
int now.
instanceID[i-04541c6e638675677] is being initialized, and does not have valid data po
int now.
instanceID[i-0648704687d86b2db] is being initialized, and does not have valid data po
int now.
instanceID[i-0e49708b52dfa1526] CPUUtilization[7.08726324441]
instanceID[i-024d2d83ca370e187] is being initialized, and does not have valid data po
int now.
instanceID[i-08537ea91ae36ee6b] is being initialized, and does not have valid data po
int now.
Total: 6
start counting
instanceID[i-0fd64537a3339828c] is being initialized, and does not have valid data po
int now.
instanceID[i-00df00c3e30e5c6d3] is being initialized, and does not have valid data po
int now.
instanceID[i-04541c6e638675677] is being initialized, and does not have valid data po
int now.
instanceID[i-0648704687d86b2db] is being initialized, and does not have valid data po
int now.
instanceID[i-0f0fc24fe9e84448c] is being initialized, and does not have valid data po
int now.
instanceID[i-0e8d6a7f41315ec61] is being initialized, and does not have valid data po
int now.
instanceID[i-0e49708b52dfa1526] CPUUtilization[7.08726324441]
instanceID[i-024d2d83ca370e187] is being initialized, and does not have valid data po
int now.
```

```
./test2.sh [IP_address] [sleep_time] [total_requests]
4
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
4
chain saw, chainsaw* 4
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
4
chain saw, chainsaw* moving van* tailed frog, bell toad, ribbed toad, tailed toad, As
caphus trui* moving van* container ship, containership, container vessel* tailed frog
, bell toad, ribbed toad, tailed toad, Ascaphus trui* 28
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
28
chain saw, chainsaw* container ship, containership, container vessel* moving van* tai
led frog, bell toad, ribbed toad, tailed toad, Ascaphus trui* 18
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
18
chain saw, chainsaw* container ship, containership, container vessel* moving van* tai
led frog, bell toad, ribbed toad, tailed toad, Ascaphus trui* speedboat* sorrel* dugo
ng, Dugong dugon* sorrel* brambling, Fringilla montifringilla* black stork, Ciconia n
igra* Japanese spaniel* Dandie Dinmont, Dandie Dinmont terrier* brambling, Fringilla
montifringilla* sorrel* frilled lizard, Chlamydosaurus kingi* sorrel* hartebeest* spe
edboat* oil filter* black stork, Ciconia nigra* ski* fox squirrel, eastern fox squirr
el, Sciurus niger* thresher, thrasher, threshing machine* airliner* oil filter* harte
beest* redbone* sorrel* Dandie Dinmont, Dandie Dinmont terrier* speedboat* tailed fro
g, bell toad, ribbed toad, tailed toad, Ascaphus trui* gazelle* thresher, thrasher, t
hreshing machine* sorrel* sorrel* sorrel* barn spider, Araneus cavaticus* frilled liz
ard, Chlamydosaurus kingi* 15
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
15
brambling, Fringilla montifringilla* fox squirrel, eastern fox squirrel, Sciurus nige
r* hartebeest* black stork, Ciconia nigra* Dandie Dinmont, Dandie Dinmont terrier* ch
ain saw, chainsaw* ski* container ship, containership, container vessel* moving van*
frilled lizard, Chlamydosaurus kingi* tailed frog, bell toad, ribbed toad, tailed toa
d, Ascaphus trui* thresher, thrasher, threshing machine* sorrel* sorrel* barn spider,
 Araneus cavaticus* 29
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
29
brambling, Fringilla montifringilla* dugong, Dugong dugon* hartebeest* sorrel* fox sq
uirrel, eastern fox squirrel, Sciurus niger* chain saw, chainsaw* Dandie Dinmont, Dan
die Dinmont terrier* gazelle* black stork, Ciconia nigra* redbone* container ship, co
ntainership, container vessel* ski* Japanese spaniel* oil filter* frilled lizard, Chl
amydosaurus kingi* airliner* moving van* tailed frog, bell toad, ribbed toad, tailed
toad, Ascaphus trui* sorrel* sorrel* speedboat* thresher, thrasher, threshing machine
* speedboat* tailed frog, bell toad, ribbed toad, tailed toad, Ascaphus trui* sorrel*
 sorrel* barn spider, Araneus cavaticus* sorrel* 23
./test.sh [IP_address] [concurrent_requests_number] [total_requests_number]
23
fox squirrel, eastern fox squirrel, Sciurus niger* hartebeest* brambling, Fringilla m
```

Figure 4.2

5. Individual contributions (optional)

Jagdeesh Basavaraju (ASU ID: 1213004713)

The project aims at building an elastic web application that can automatically scale out and in on demand and cost effectively by using cloud resources. The resources used were from Amazon Web Services. It is an image recognition application exposed as a Rest Service to the clients to access. The application takes the URL of the image and returns the predicted output by the deep learning model by using the AWS resources for computation, transportation and storage. So the tasks involved designing the architecture, implementing RESTful Web Services, load balancer and so on. Below is the contribution of mine to each of the phases of the project.

I. Design:

The design phase involved deciding which of the resources from the Amazon Web Services to use, where will each of the individual components reside in the architecture, how will they communicate with each other, where will the deep learning image recognition module be called and so on. I collaborated with the rest of the team to come up with the design and architecture of the project. I worked on finalizing the design of the Rest Service (Web-tier) and the App-tier. I decided the structure of the individual modules in both Web-tier and App-tier applications.

II. Implementation:

The implementation involved implementing the RESTful Web Service for web-tier, application to run the image classification and storing results and the module for load balancing. I worked on the implementation of Spring Boot applications for Web-Tier and App-Tier. I implemented the load balancer in the Web-tier application itself as a separate process, but the logic of load balancing was discussed with the entire team and decided. Operations on SQS like sending a message, deleting a message, receiving a message, getting the approximate number of messages in the queue and operations on S3 like creating a bucket, writing to the bucket and EC2 instance creation, termination, and checking the number of instances running were implemented by me using the APIs of the Amazon Web Services in Java. These operations logically combined is what makes the application.

III. Testing:

Testing phase involved unit testing, integration testing and end-end testing. It also involved running the scripts provided to us. I did the unit testing and the integration testing of the web-tier and the application running in the app-instances. I had to also do a lot of manual testing while implementing and checking the correctness of the application and the modules.

Suraj S Kattige (ASU ID: 1211230381)

I. Design:

I collaborated with my teammates in design of the architecture of the system. After brainstorming through various models, we came up with the architecture described in the project. I suggested the working of the load balancing algorithm for the system. We decided to include a hash table to improve the efficiency of the application as well.

II. Implementation:

I was responsible for figuring out the mechanism of taking snapshot of an EC2 web tier instance and using those snapshots in the scaling out process. As the EC2 application tier instances should be identical to one another on startup, I created an EC2 instance and created a listener shell script, and also stored the deep learning code. On startup, the app tier EC2 instances was designed to run a listener script to listen to the incoming requests. If the EC2 instance does not receive any requests, it terminates. This is part of the autoscaling and load balancing process. I also coded in java to create the snapshot of the required EC2 instance using the instance ID in the application-tier. I also collaborated with my team members in extensively testing the system to handle multiple requests from users and making the code more efficient to lessen the execution time of the system.

III. Testing:

I was also extensively involved in the testing phase of the project as well. My team and I performed unit, integration and regression testing of the system. Regression testing was particularly important as we had to take care of the efficiency of the system, while also checking that the code does not break the system.