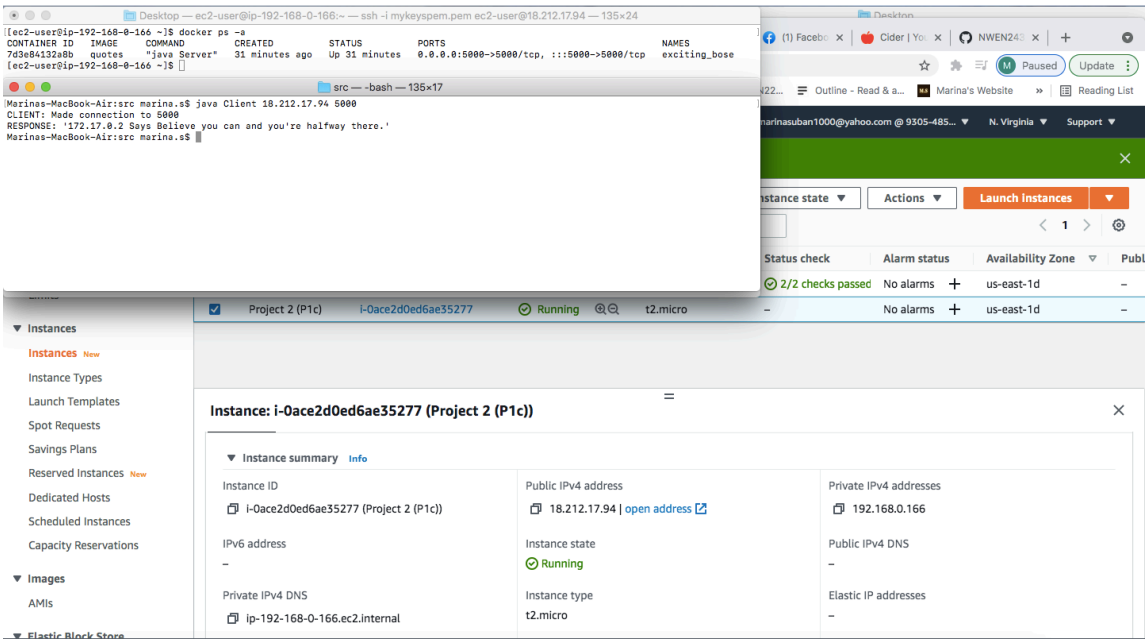
SUBANKAMO 300471606

NWEN P3 P1

MAKING A DOCKER IMAGE

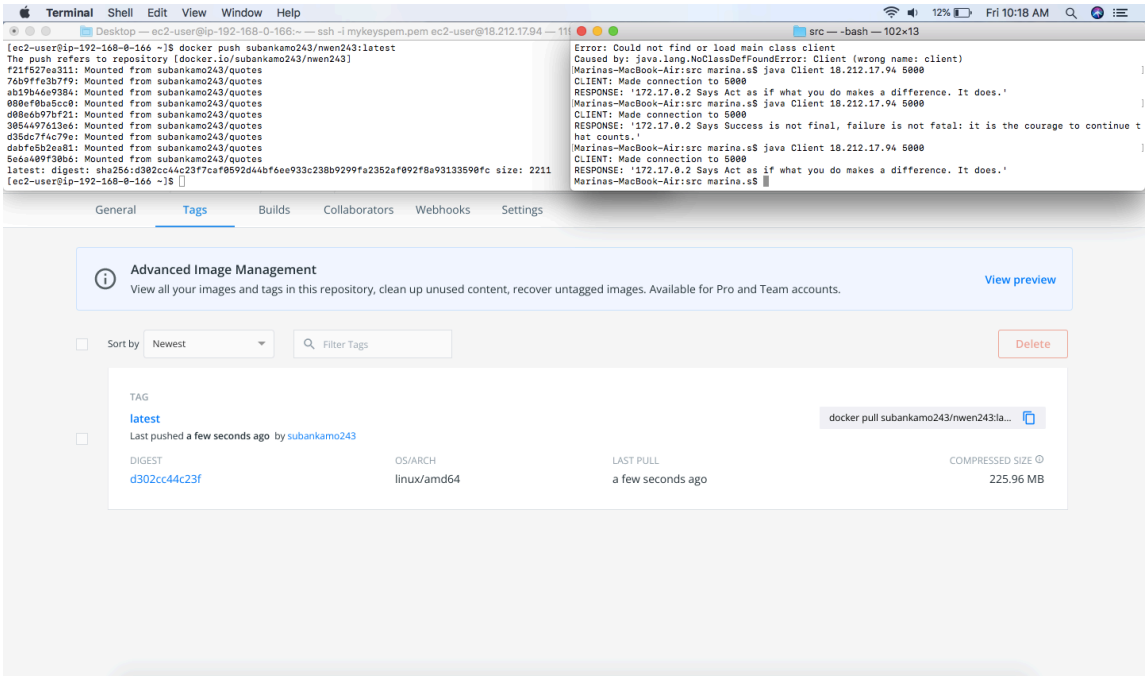


What is going on:
After ssh into the instance we made for project 2 and applying necessary updates. We created a dockerfile and upload it to our machine. We then get docker to build the image and we then run the image.

What each screen shot reveals:
In the screen shot we run the command ‘docker ps -a’ which gives us a log of prior docker executions.
We also have our client which demonstrate that the server is active.

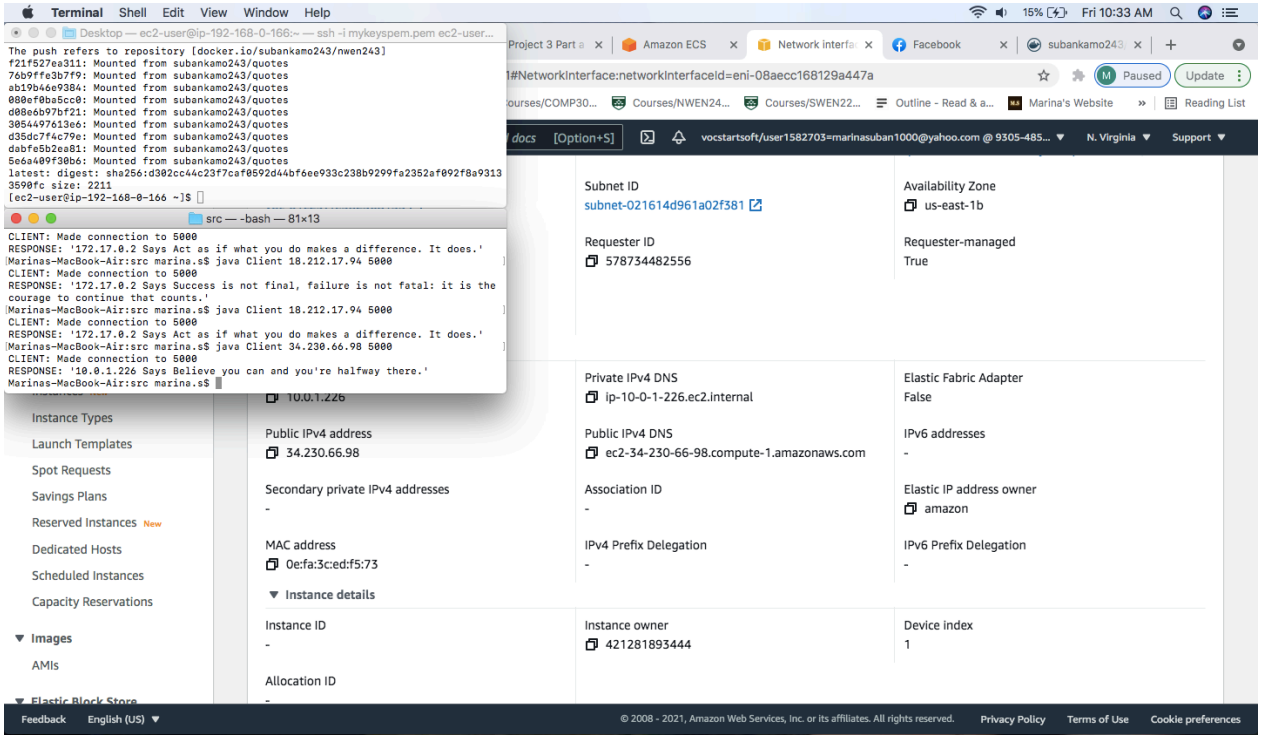
How they line up to the sections in this Project:
This is the end of ‘MAKING A DOCKER IMAGE’ section of Project 3 Part (a) - Introducing Containers

CREATING YOUR DOCKER REGISTRY



What is going on:
We created a docker repository and pushed our docker image to the repository with the tag latest as outlined by the brief.
What each screen shot reveals:
In the screen shot we run the command ‘docker push subankamo243/nwen243:latest’ which push our image to the repository we created.
We also have our client which demonstrate that the server is active.
We have the repository in the background to allow for comparison to our push.
How they line up to the sections in this Project:
This is the end of ‘CHALLENGE I: PUSHING THE DOCKER IMAGE TO A CONTAINER REGISTRY’ section of Project 3 Part (a) - Introducing Containers

CHALLENGE II: PULLING THE DOCKER IMAGE TO ECS AND LAUNCHING IT!

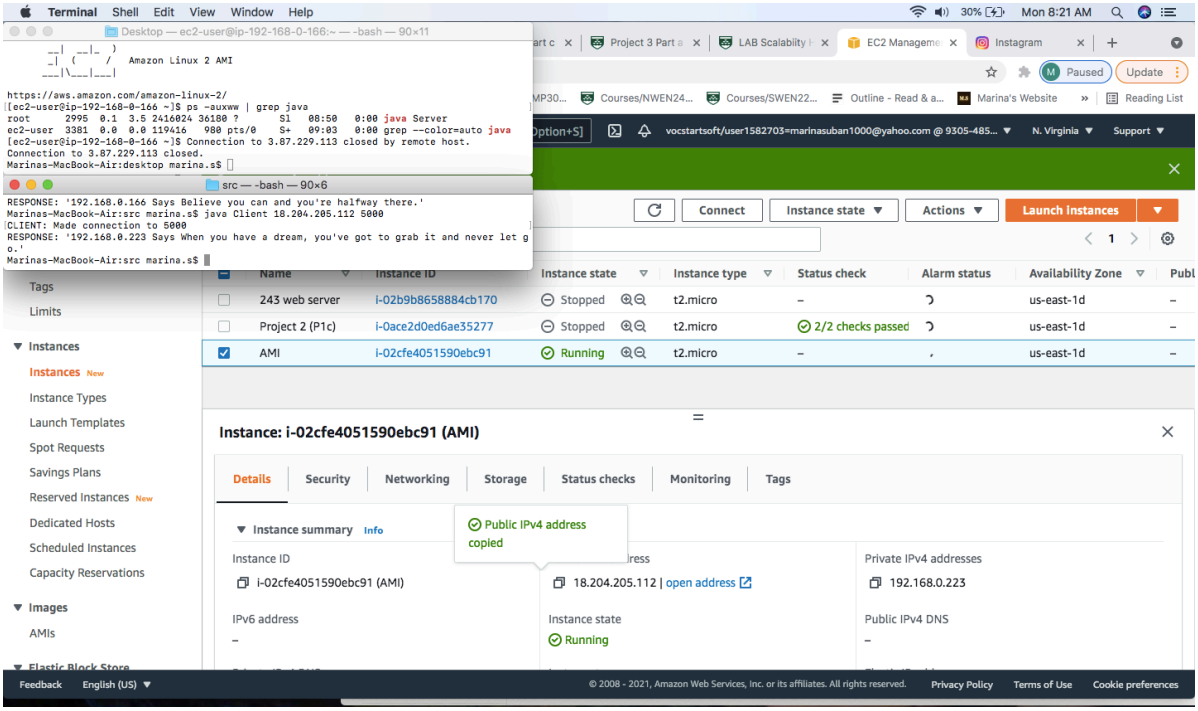


What is going on:
Created a docker container image, put it in a repository and then had AWS provision a server-less FARGATE service for our quotes server.
What each screen shot reveals:
The screenshot demonstrates a successful connection between the client and the docker container image.
How they line up to the sections in this Project:
This is the end of ‘CHALLENGE II: PULLING THE DOCKER IMAGE TO ECS AND LAUNCHING IT’ section of Project 3 Part (a) - Introducing Containers

SUBANKAMO 300471606

NWEN P3 P2

(A) MODIFYING YOUR CODE + (B) CREATING AN AMAZON MACHINE IMAGE



What is going on:

We are add a sh file to the AWS instance. A sh file is a shell script that any Unix system can run with the purpose of our “run.sh” file is to store the commands needed to start the java Server program.

We then set up a cron job using the cron daemon. The purpose of this is to have the cron job automatically run our “run.sh” file every time the AWS instance is restarted.

Lastly, we created an AMI image; this allows us to create instances off of it.

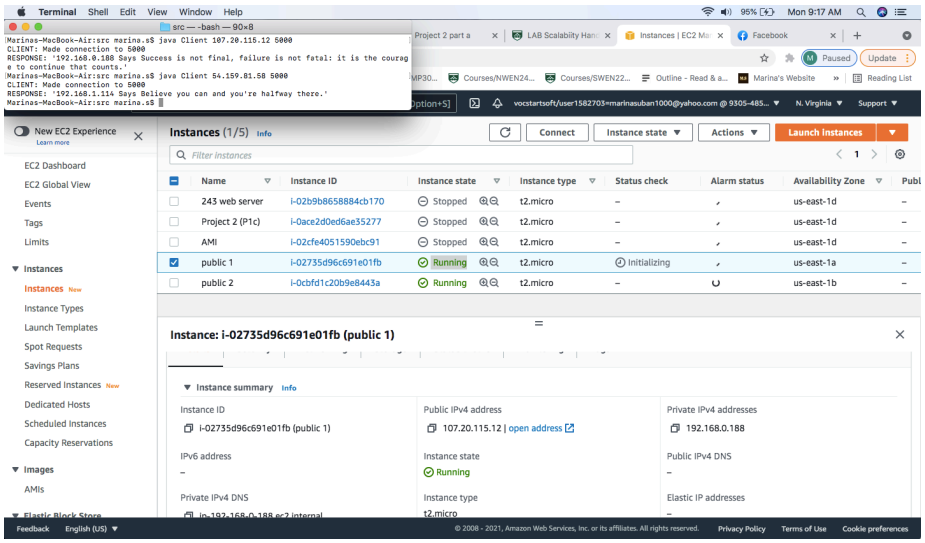
What each screen shot reveals:

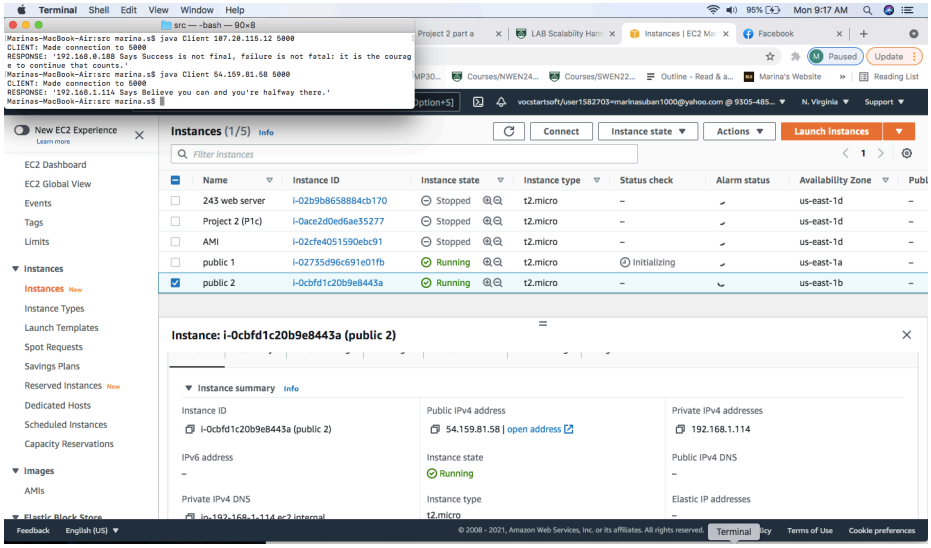
The screenshot demonstrates a successful connection between the client and the AMI instance - This image serves as evidence to show that the cron job did work properly.

How they line up to the sections in this Project:

This is the end of ‘Creating an Amazon Machine Image’ section of Project 3 Part (b) - Scale

(C) VPC TO SPAN AVALIABILITY ZONE + (D) CREATING INSTANCES TO LOAD BALANCE



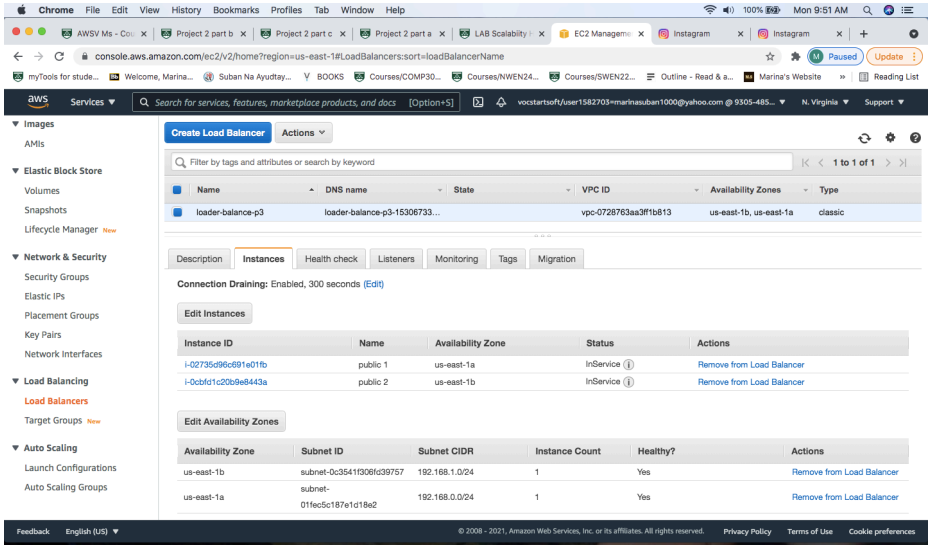


What is going on:
Created VPC and 2 public subset in different zones. The purpose of this is so we can have the load balancer allocate instances in different regions

What each screen shot reveals:
The screenshot demonstrates a successful connection between the client and the 2 new subnets.

How they line up to the sections in this Project:
This is the end of 'Creating Instances to Load Balance' section of Project 3 Part (b) - Scale

(E) CREATING A LOAD BALANCER + (F) CHECK IF IT IS WORKING

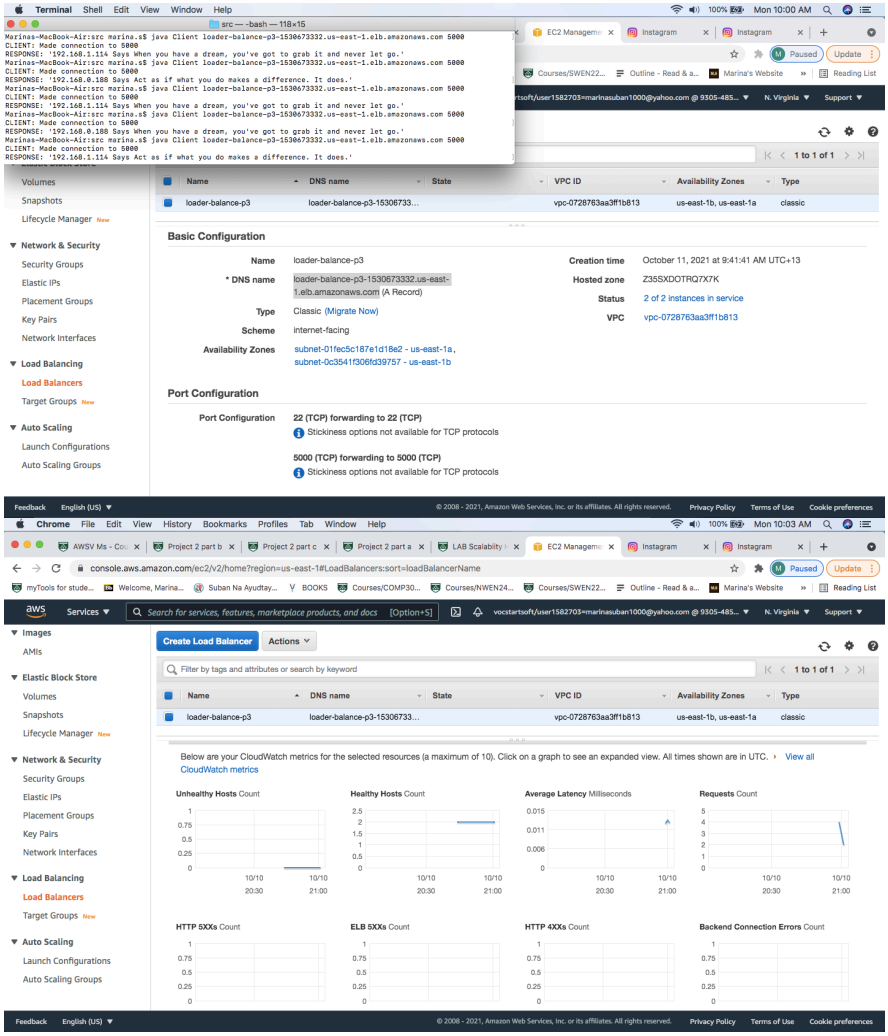


What is going on:
Created a Load Balancer, the purpose of this is to distribute the work to the two instances we had created previously when somebody tries to connect via the IP.

What each screen shot reveals:
The screenshot demonstrates that the load balancer has been created and the two instances have been added to it.

How they line up to the sections in this Project:
This is the end of Checking it is Working' section of Project 3 Part (b) - Scale

(G) TESTING



What is going on:

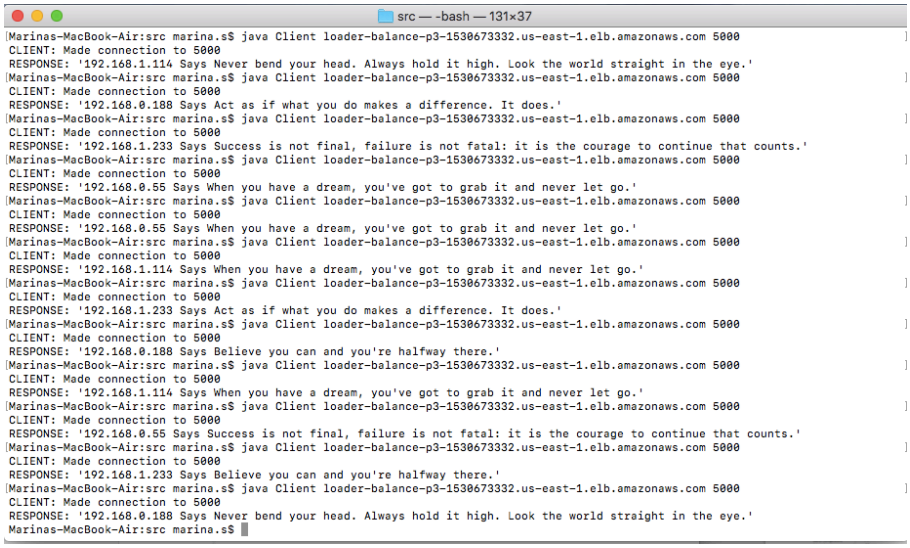
We test our loader balance with the two instances. From the client we can see that the loader balance tends to alternate between the two subnets.

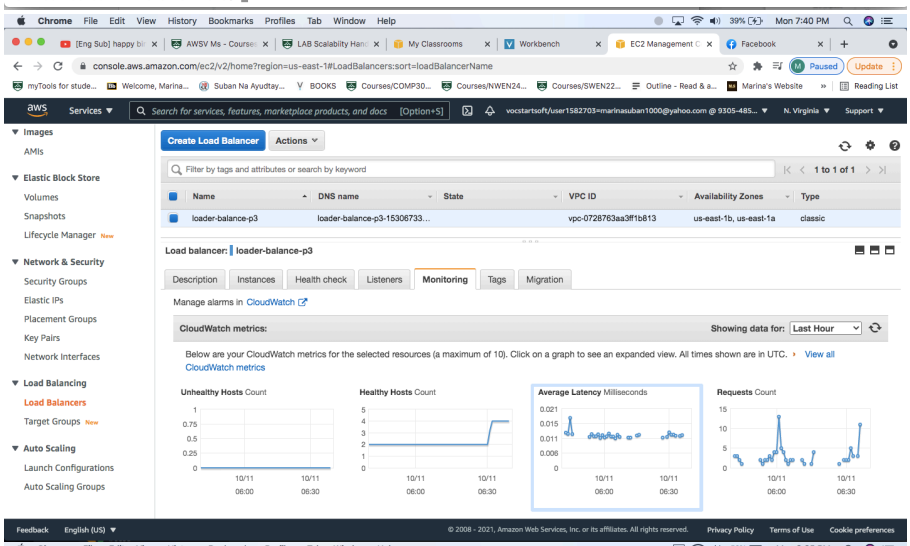
What each screen shot reveals:

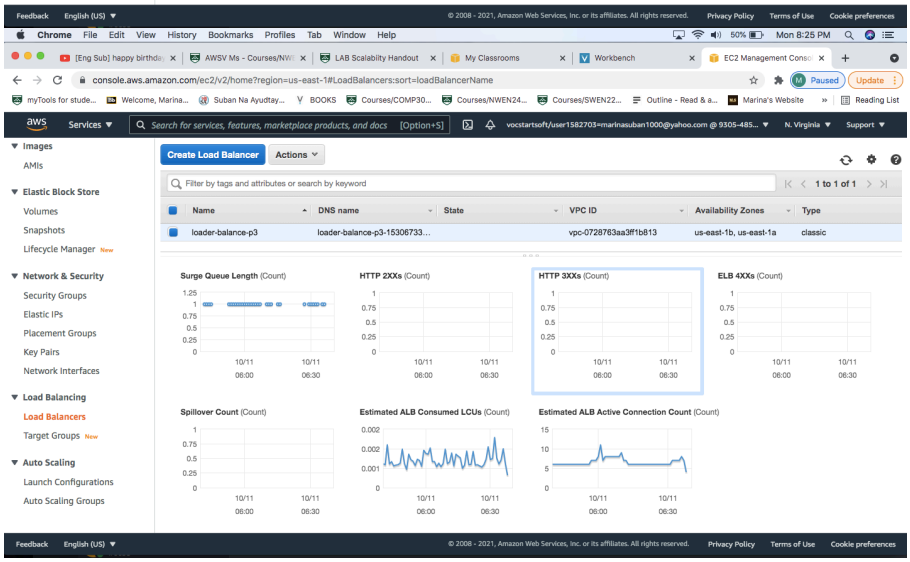
The screenshot demonstrates that the load balancer is working successfully; the IPs are alternating, which shows the load balancer is balancing the workload between the two instances. Unhealthy host is the number of instances failing (0), Healthy host is the number of operating subnets in the loader balance (2), Request are how many connections have been made to the Server within the last few moments (4->2) and average latency is the number of processed bytes.

How they line up to the sections in this Project:

This is the ‘Testing’ section for the loader balance with two subnets of Project 3 Part (b) - Scale







What is going on:

Here we are adding two new instances without static IP's to the load balancer. This should double the supply of connections we can handle.

What each screen shot reveals:

The screenshot demonstrates that the load balancer is working successfully; the IPs are alternating, which shows the load balancer is balancing the workload between the four instances.

Here we can see the number of requests has spiked however average latency in milliseconds remained relatively constant. We can also see the number of healthy hosts increase from two to four, signifying the two additional instances were indeed working.

How they line up to the sections in this Project:

This is the ‘Testing’ section for the loader balance with four subnets of Project 3 Part (b) - Scale

Questions

What policy does it look like the load balancer is using when the hosts are lightly loaded?

The loader balance appears to alternate between the two instances. When testing with a light load using 2 subnets, there were no sequential messages from the same IP. This may suggest that the loader balance was using a Round robin policy (This policy distributes incoming traffic sequentially to each server in a backend set list).

Might it change under heavy loads?

When testing under a heavy load, the response was similar. However, we were able to observe sequential IPs. This never occurred under a light load.

This is likely due to one of the instances having a much quicker response time; potentially due to its location. This would mean it can process responses more quickly. Additionally, the other instances could just be already handling client requests.

This may suggest that the loader balance was using a weight to distribute incoming traffic to servers.