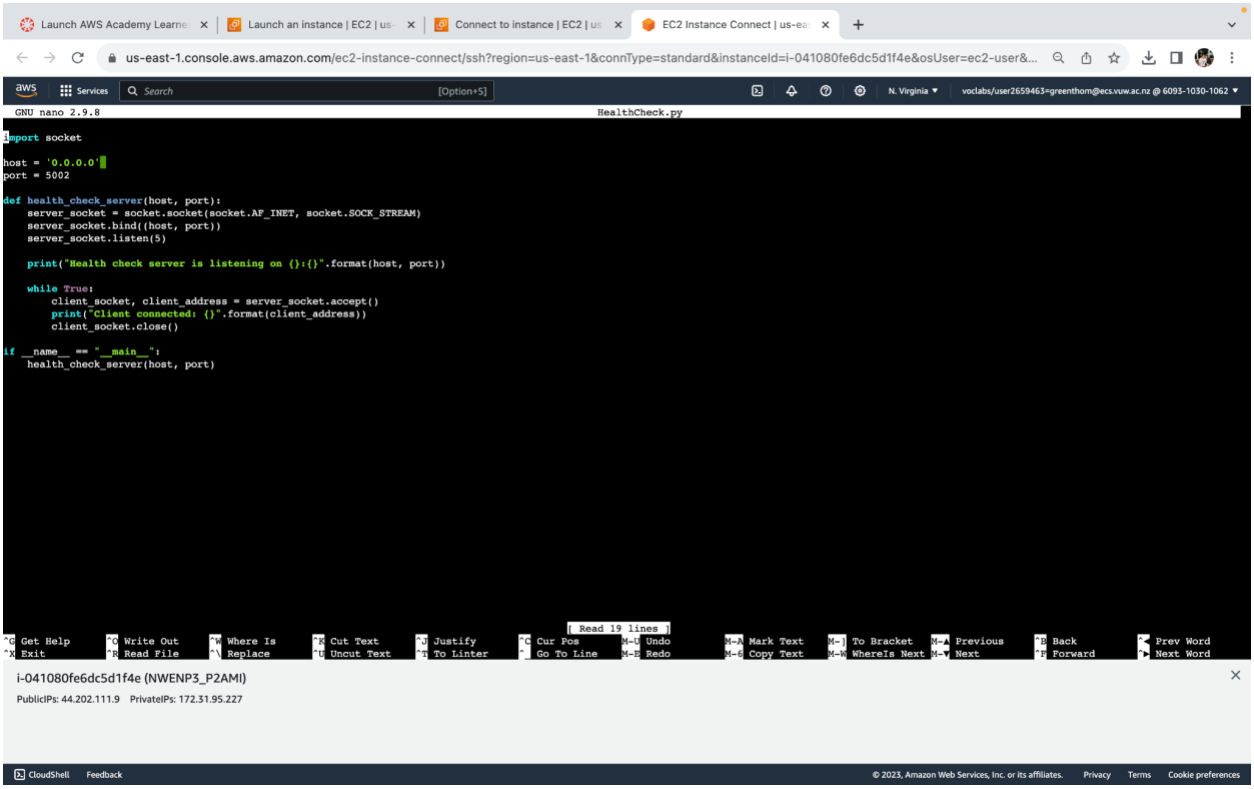
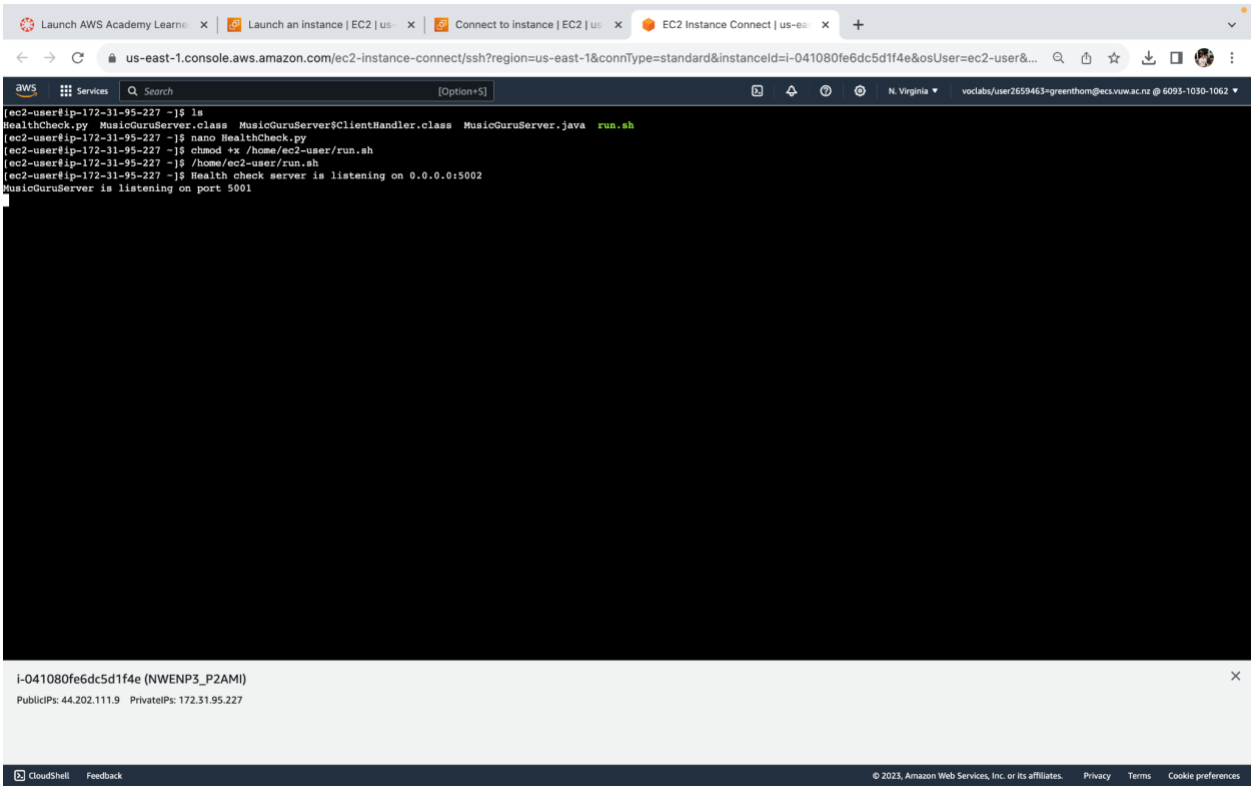
NWEN243 - Project 3
greenthom - 300536064

PART A

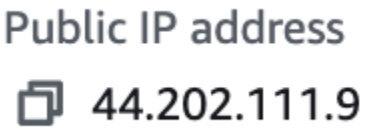
Made AMI from project 2 MusicGuruServer. And created HealthCheck.py in that ec2 instance.



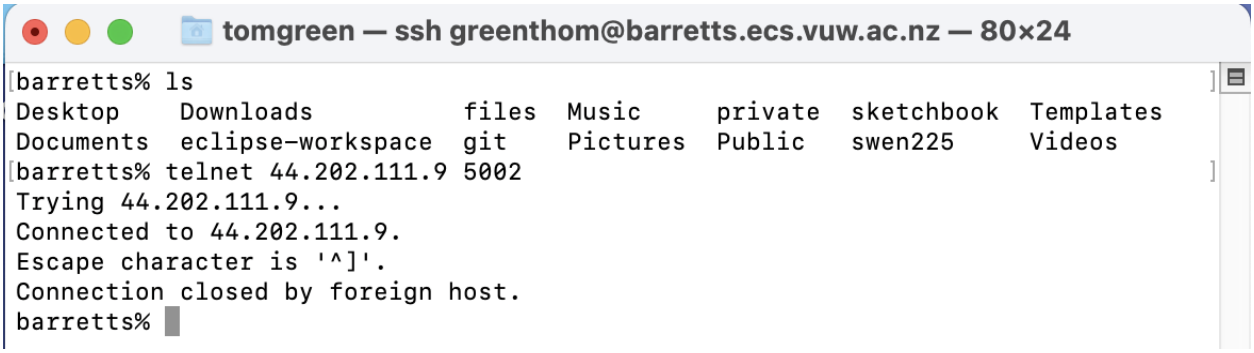
I added HealthCheck.py to my run.sh. MusicGuruServer is running at port 5001 (same as from project 2,)and HealthChecks is at port 5002. Rebooted the instance and connected back to it. File permissions are set to ensure that the script is executable using chmod +x /home/ec2-user/run.sh. Here, you can see in the instance that HealthCheck and MusicGuruServer are running properly from run.sh, using the command /home/ec2-user/run.sh



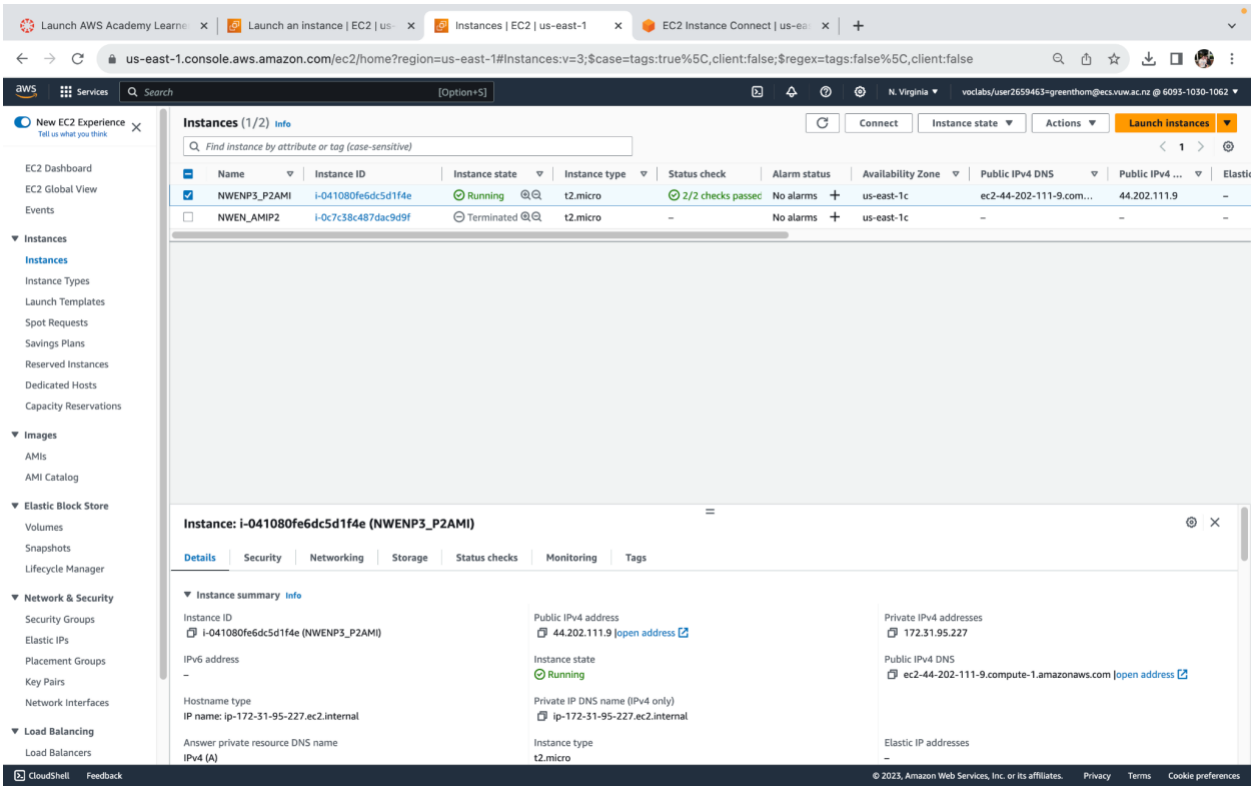
This is the public IP address of my instance running my HealthCheck and MusicGuruServer code



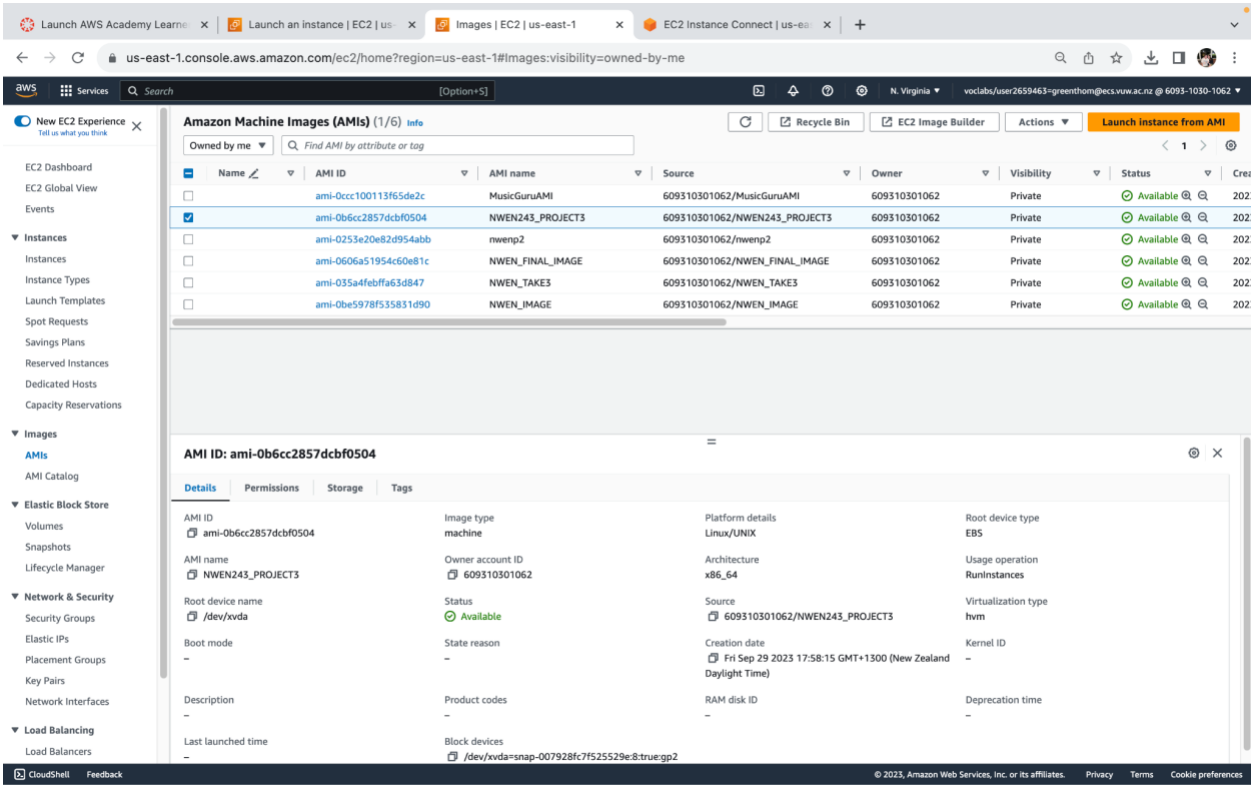
Here, we contact the instance using telnet, which contacts the HealthCheck server code.



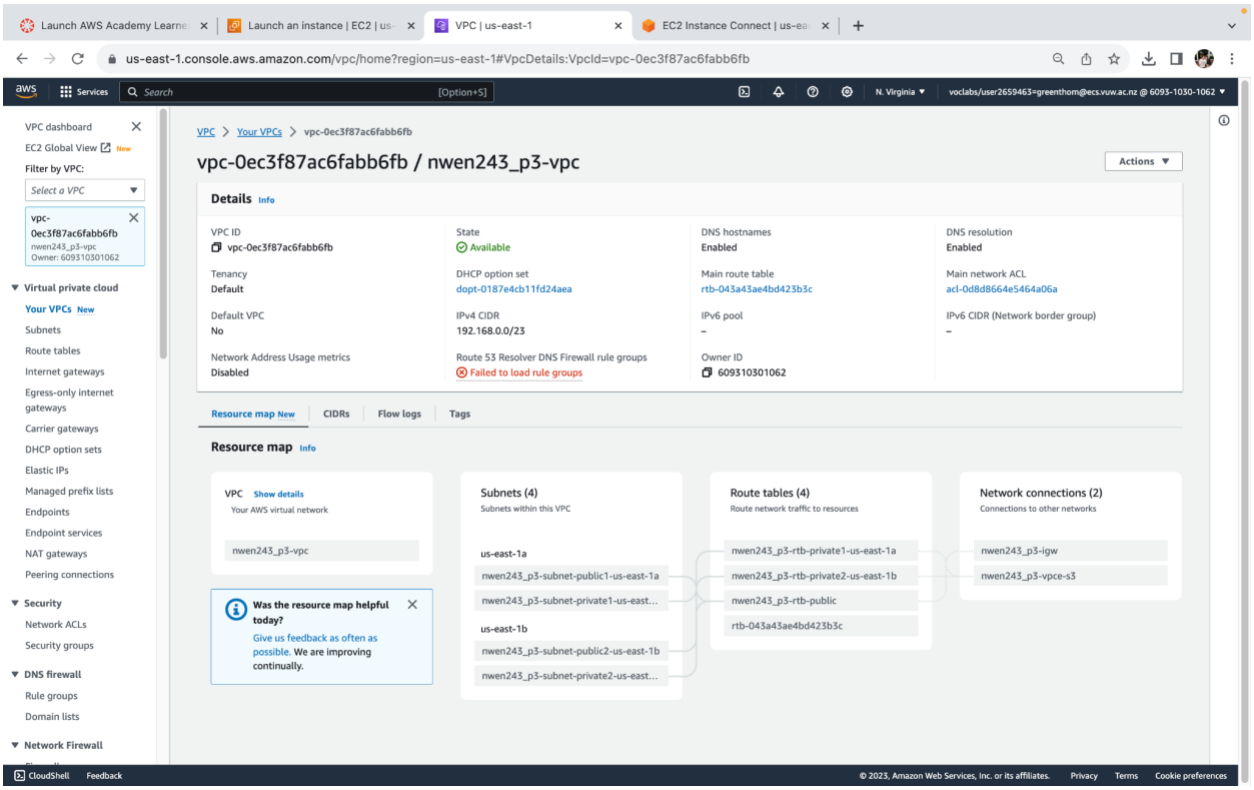
Now that we have checked that it works, we then create an AMI from the instance with the HealthCheck server code



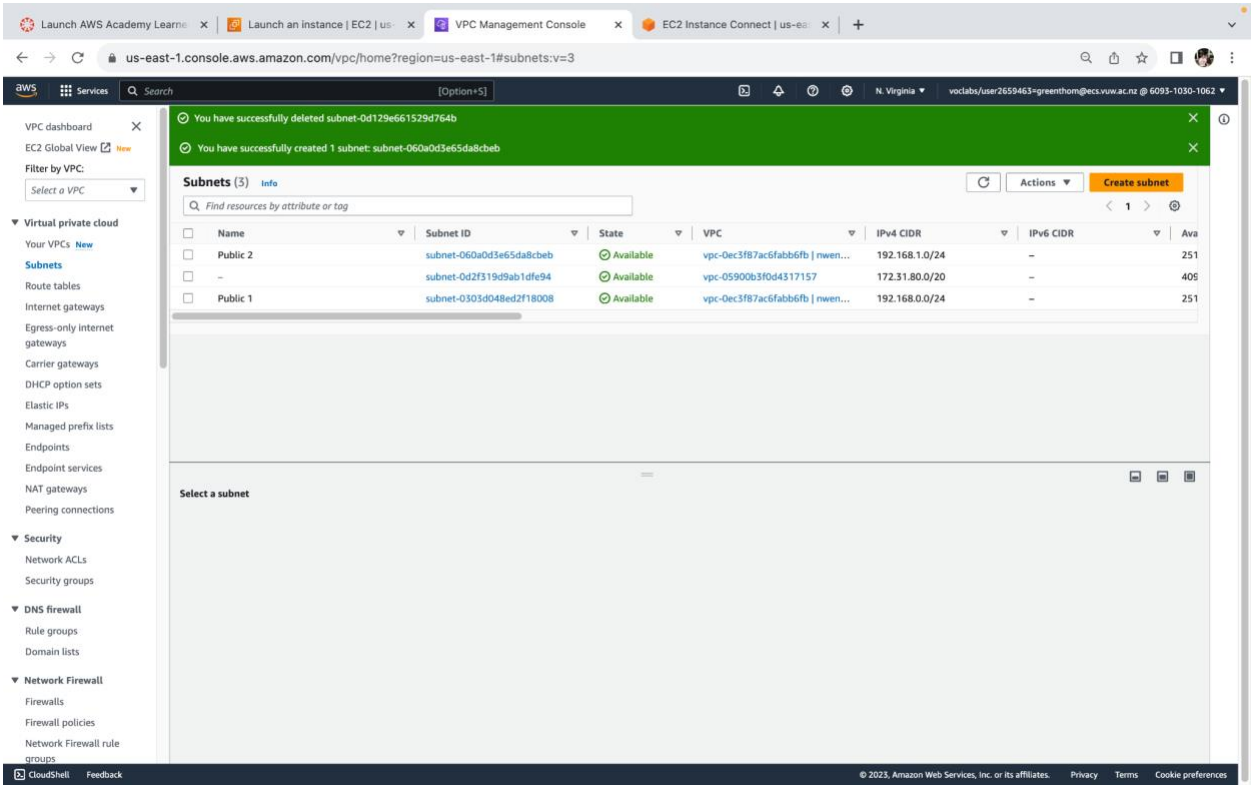
Here, we can see the AMI that we created for project three, which we will use throughout parts A and part b



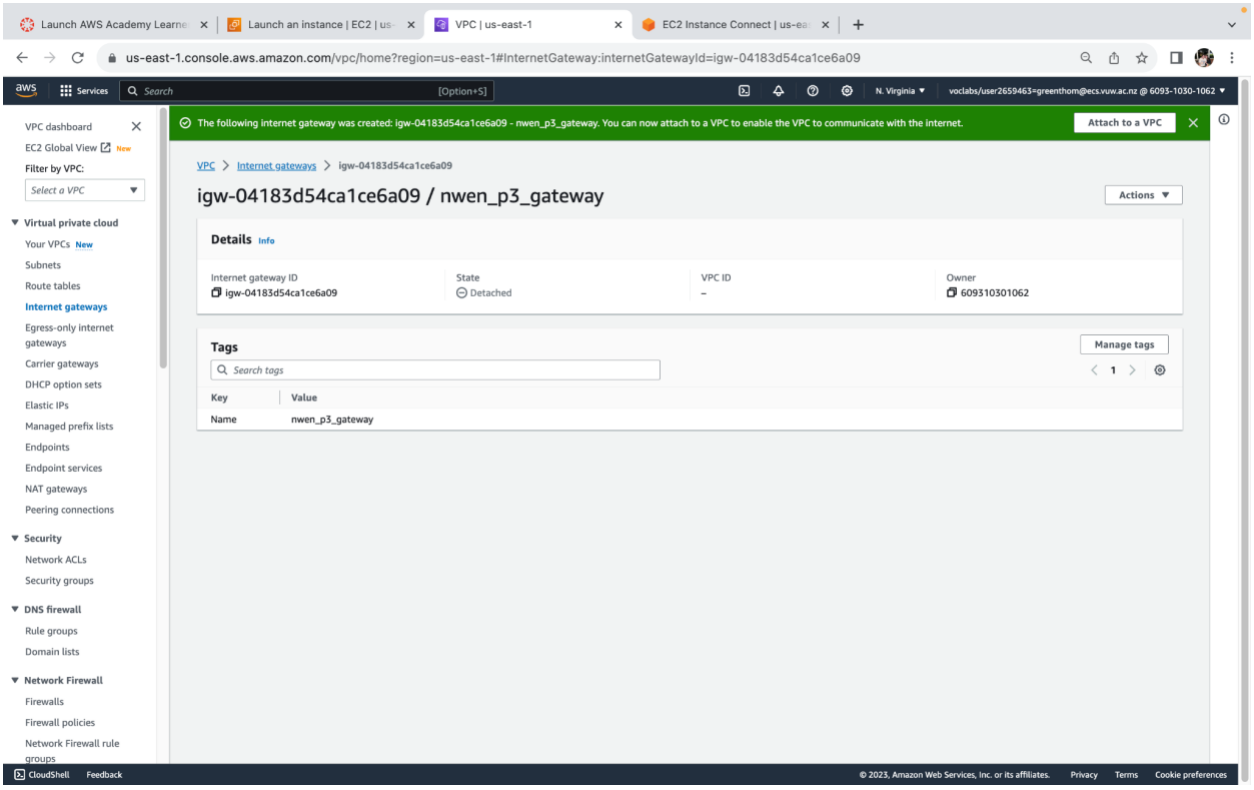
We created a new VPC from here, which I named 'nwen243_p3-vpc' at IPv4 CIDR '192.168.0.0/23.'



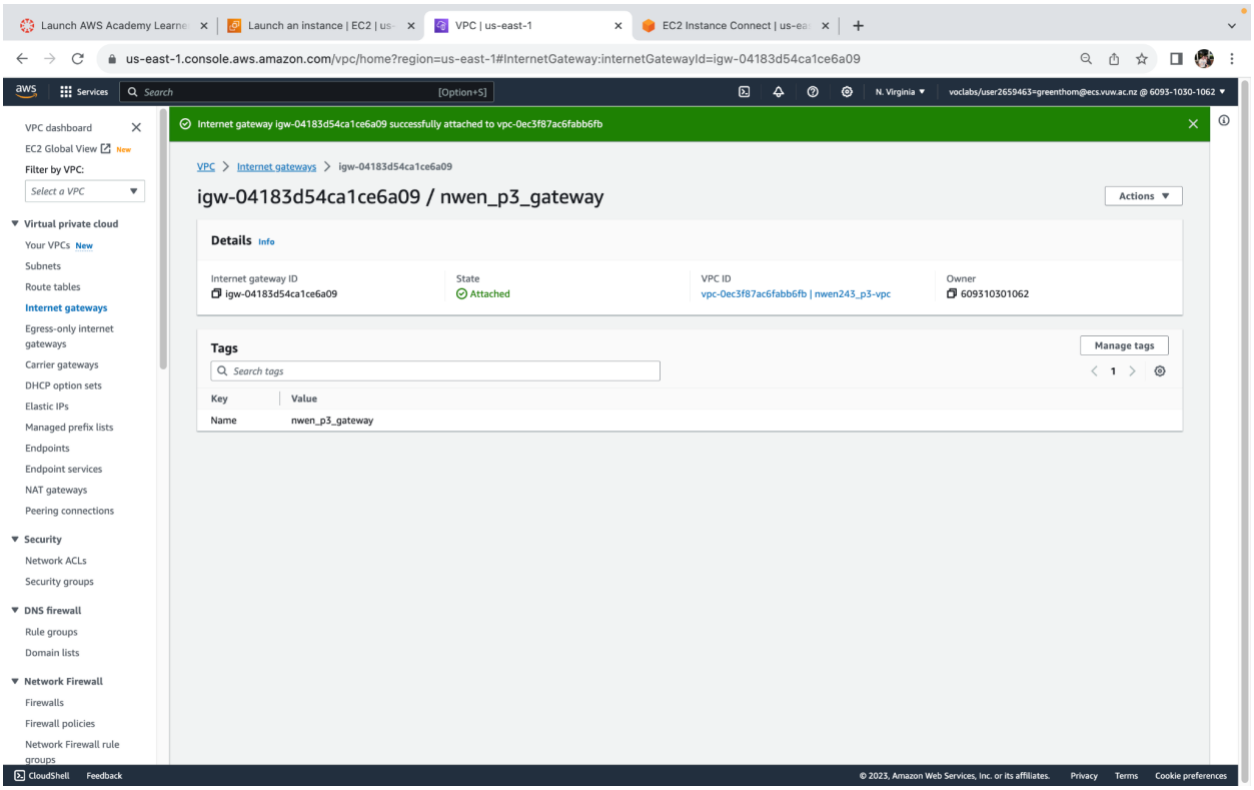
From here, we create two public subsets (public one and public 2) with a divided CIDR - Public One: 192.168.0.0/24 and Public Two: 192.168.1.0/24. Also, both have different zones at us-east-1a and us-east-1b, respectively.



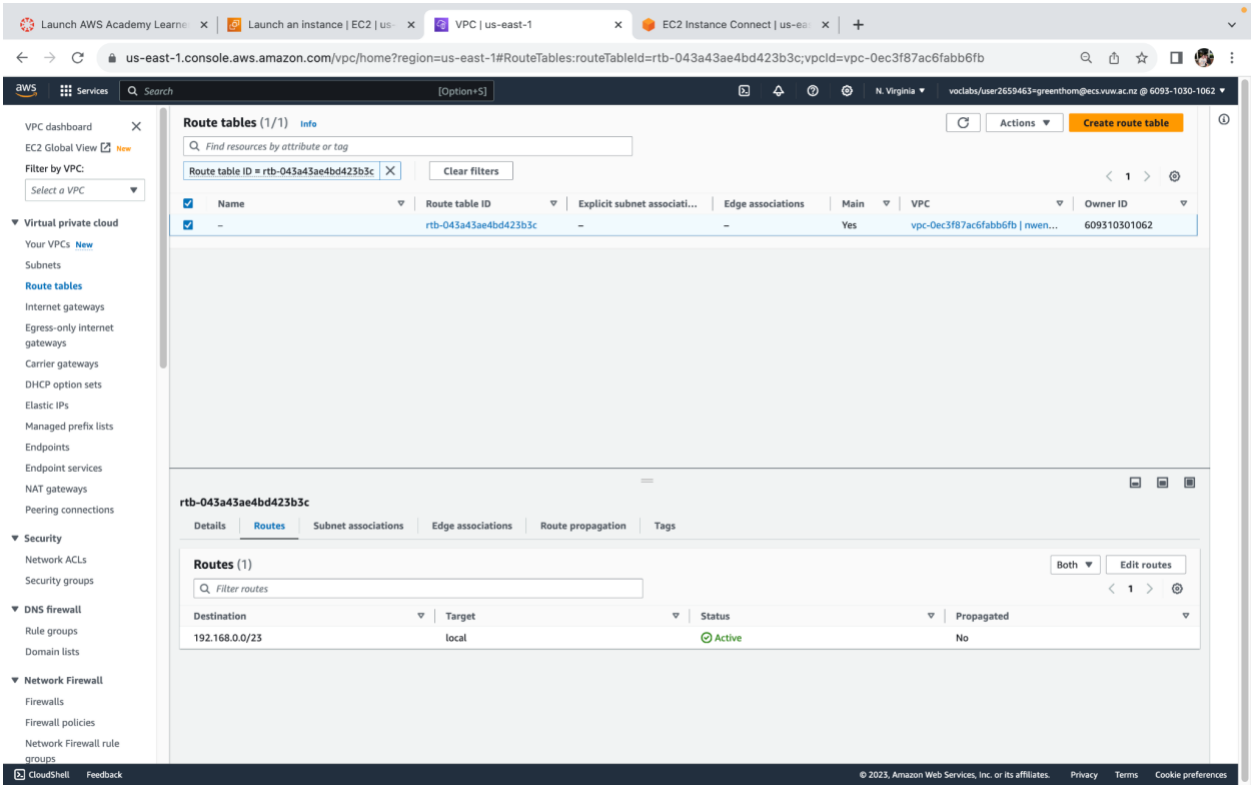
I made an internet gateway and have named it 'nwen_p3_gateway.'



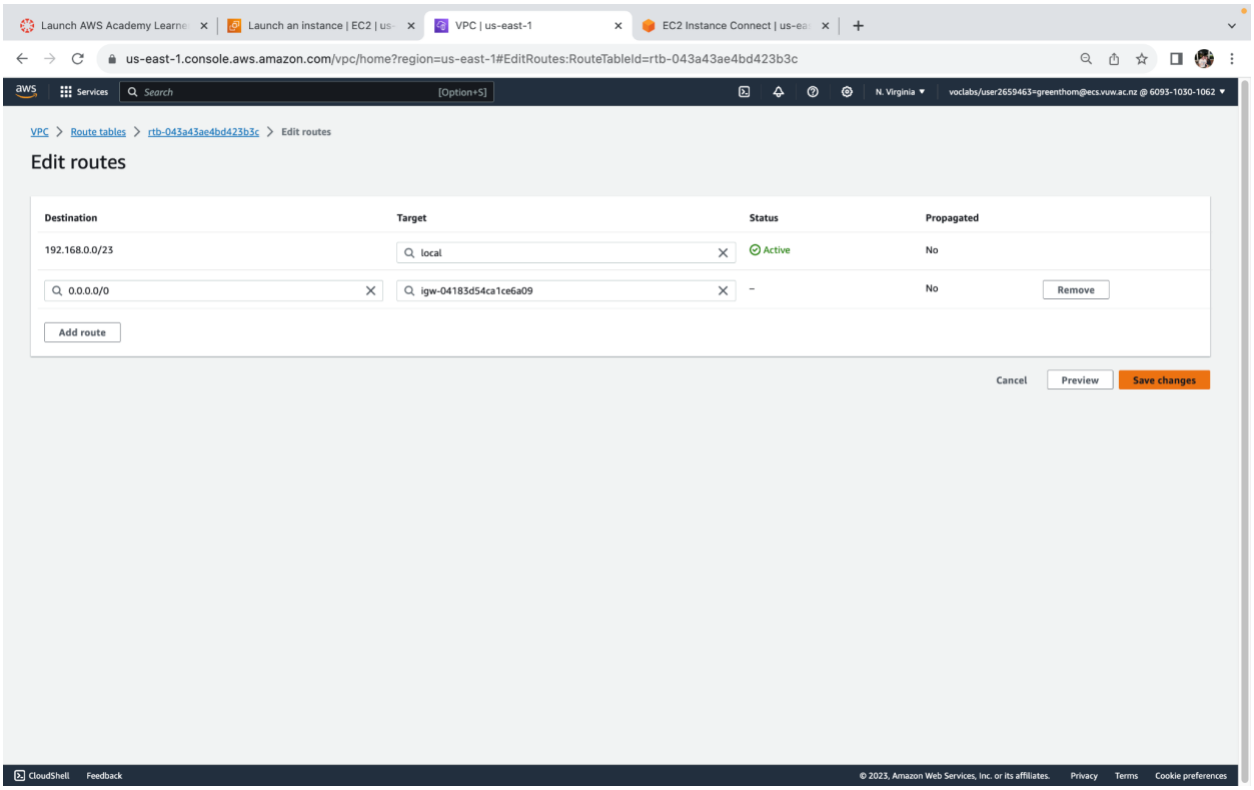
Here, we have attached our gateway to the VPC we just created before. Here, we can see the state of the gateway has gone from detached to attached, showing it is successful



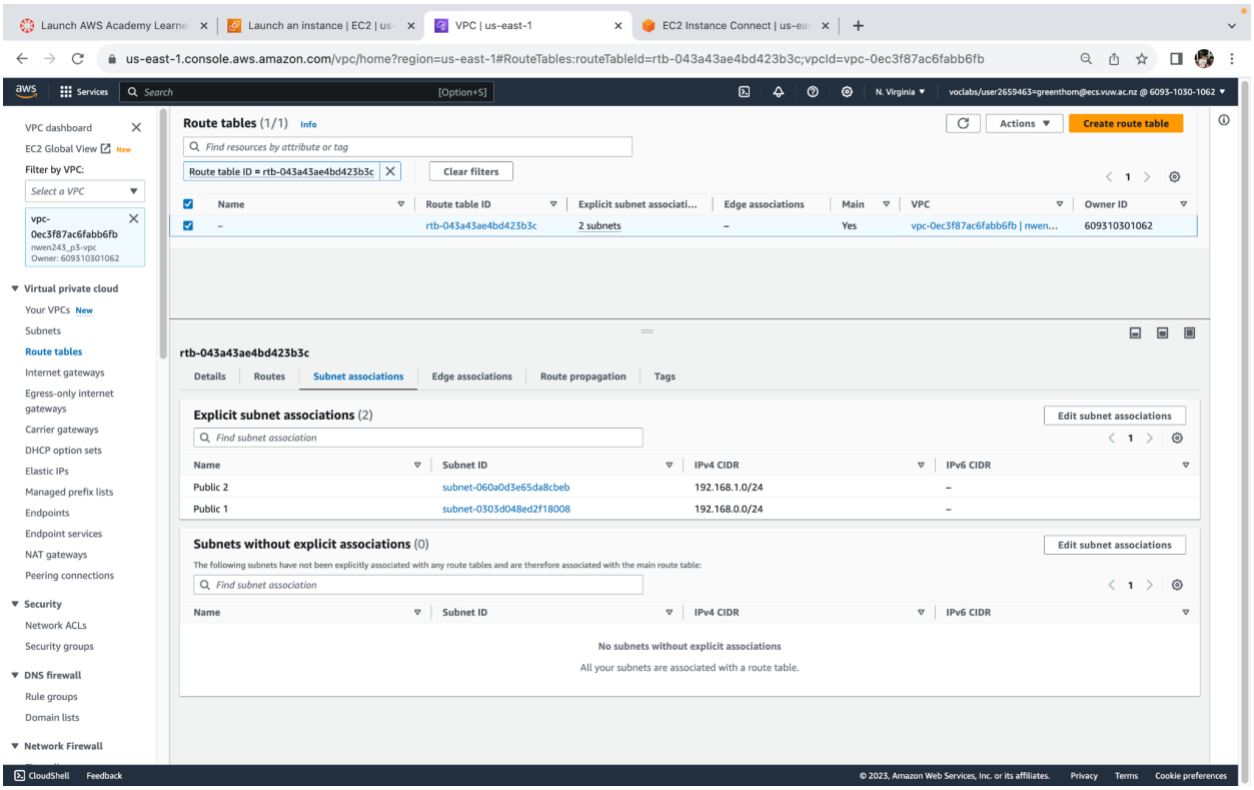
Here, we can see the default route



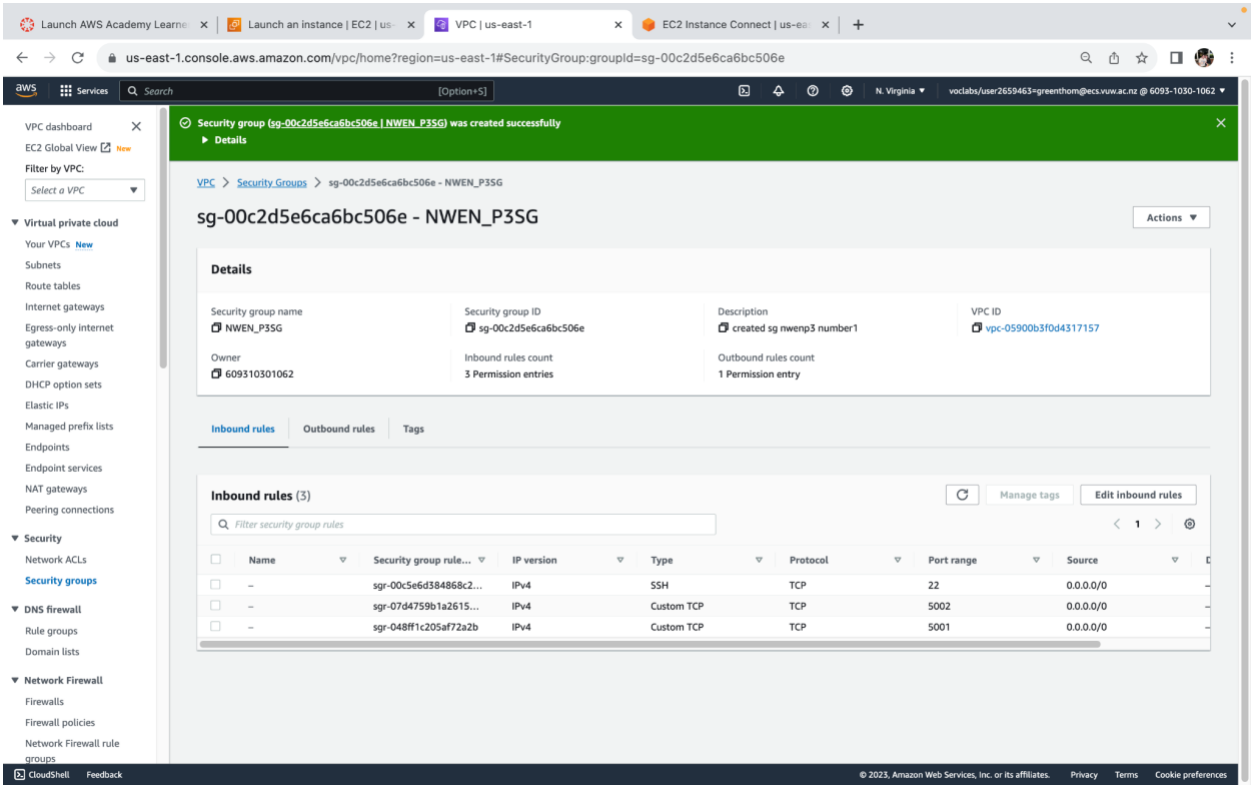
From here, we edit this to include the internet gateway we added before and add the destination to 0.0.0.0/0



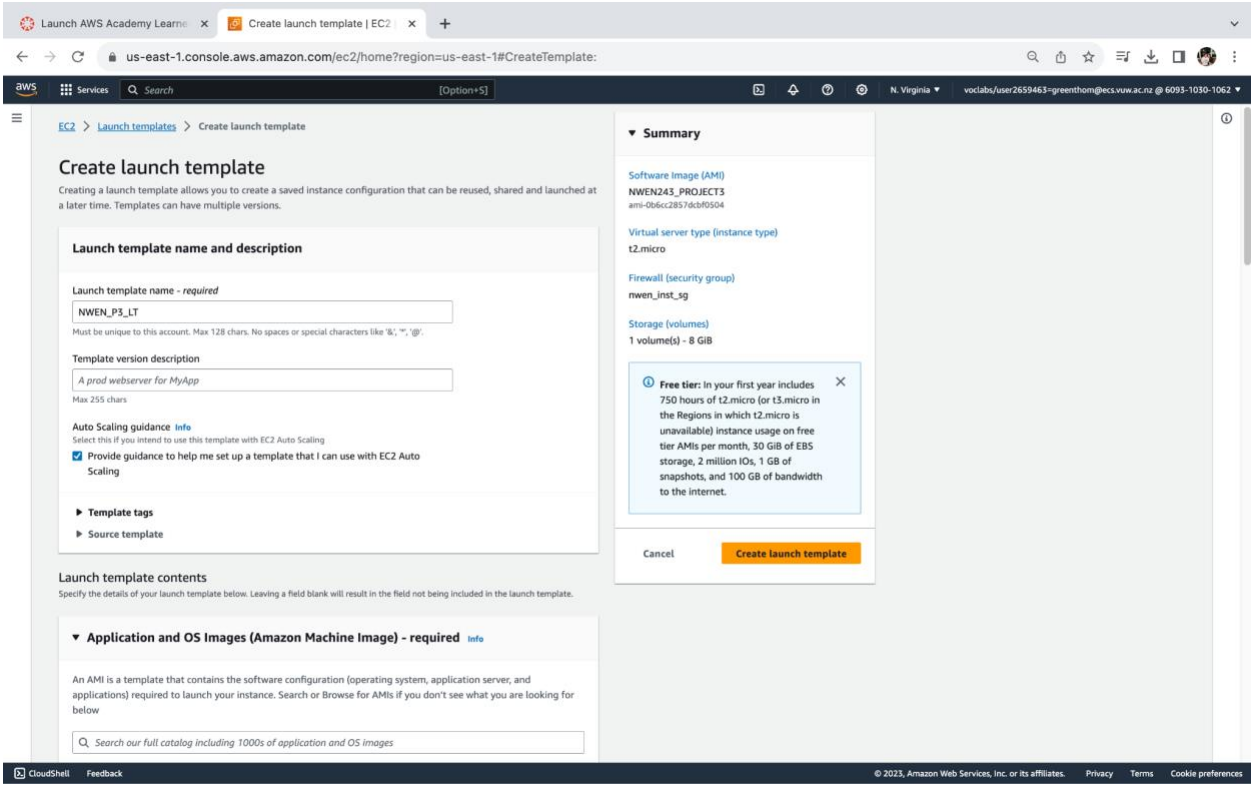
The next thing we have to do is associate the new routing table with the public subnets. Here, we can see that we have added the Public 1 and 2 subsets we created before.



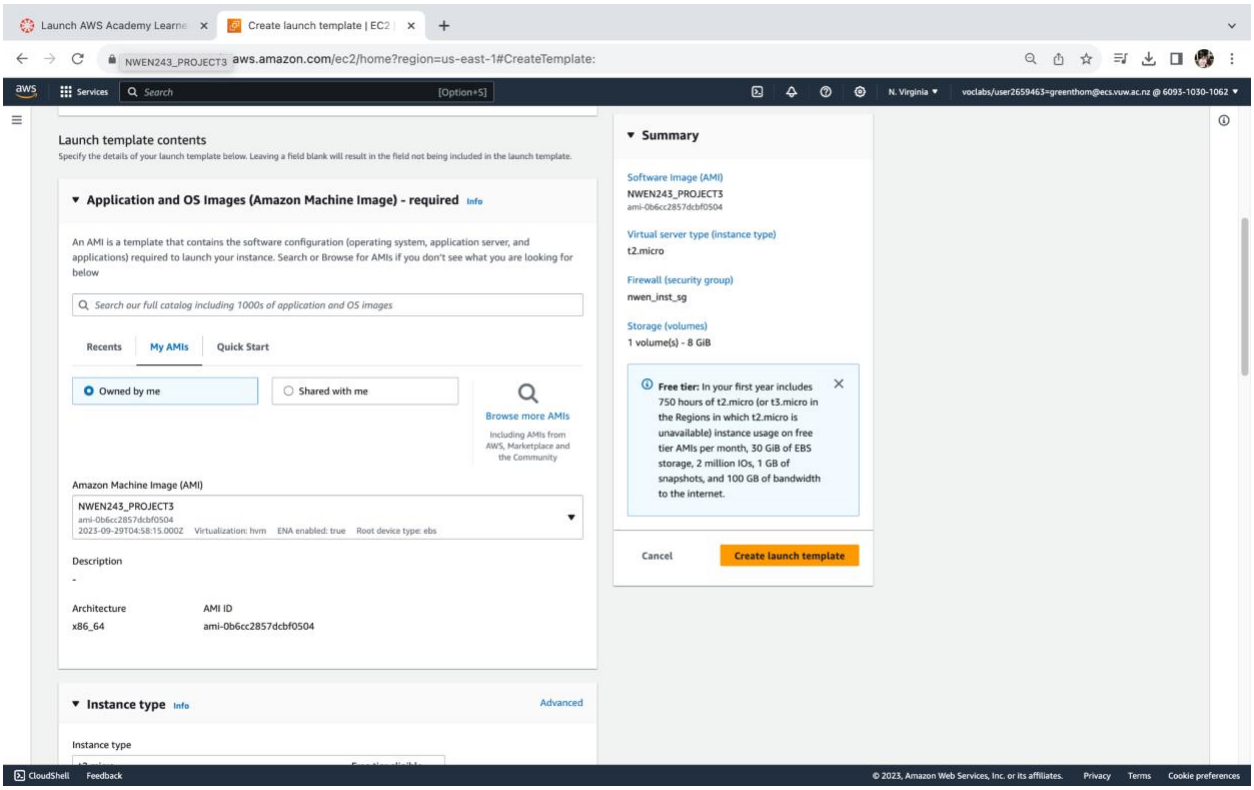
From here, we then go and make a security group and have named it 'NWEN_P3SG'. We replaced the default VPC with the one we created before. We then add inbound rules: one for SSH and two for our Server and HealthCheck code at ports 5001 and 5002, respectively.



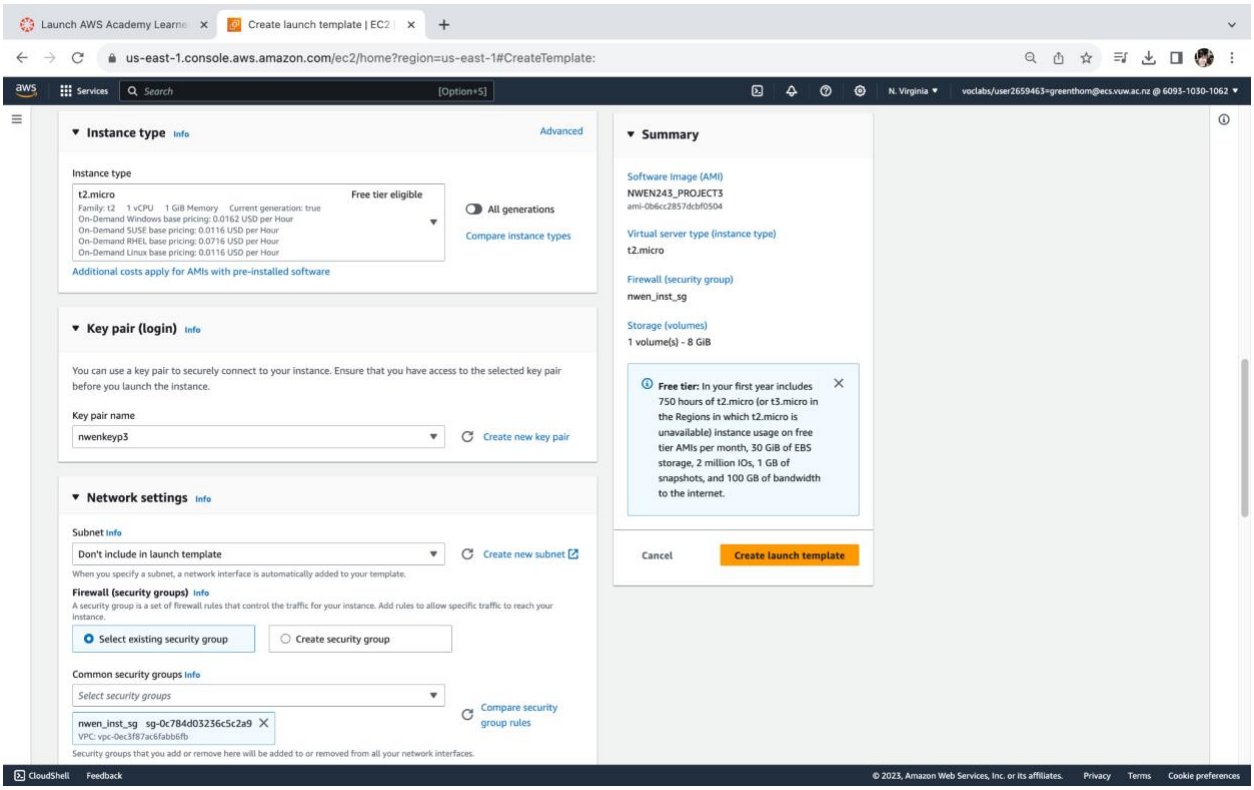
Now, we go and create a launch template for the P3 AMI. We named it 'NWEN_P3_LT' and checked the box to provide EC2 autoscaling.



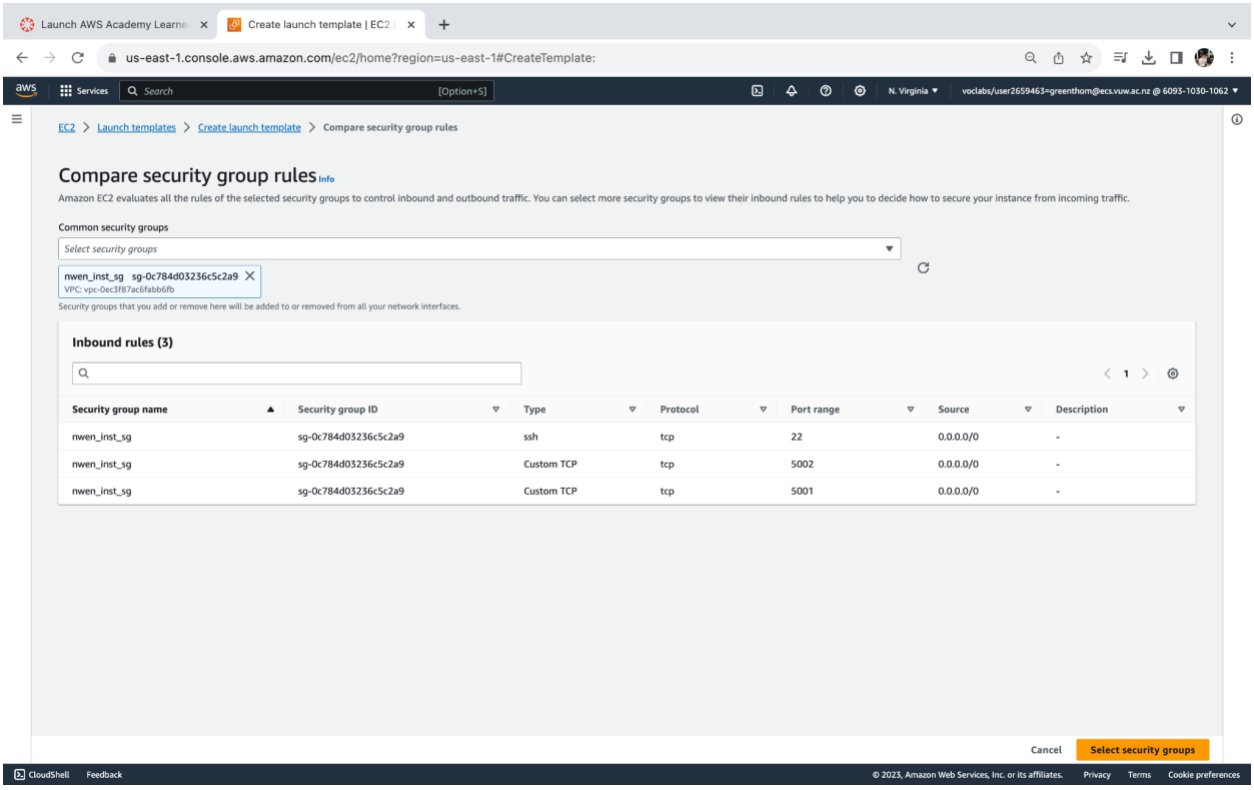
From here, we go down to Application and OS Images, where we select the AMI, we created at the start called 'NWEN243_PROJECT3.'



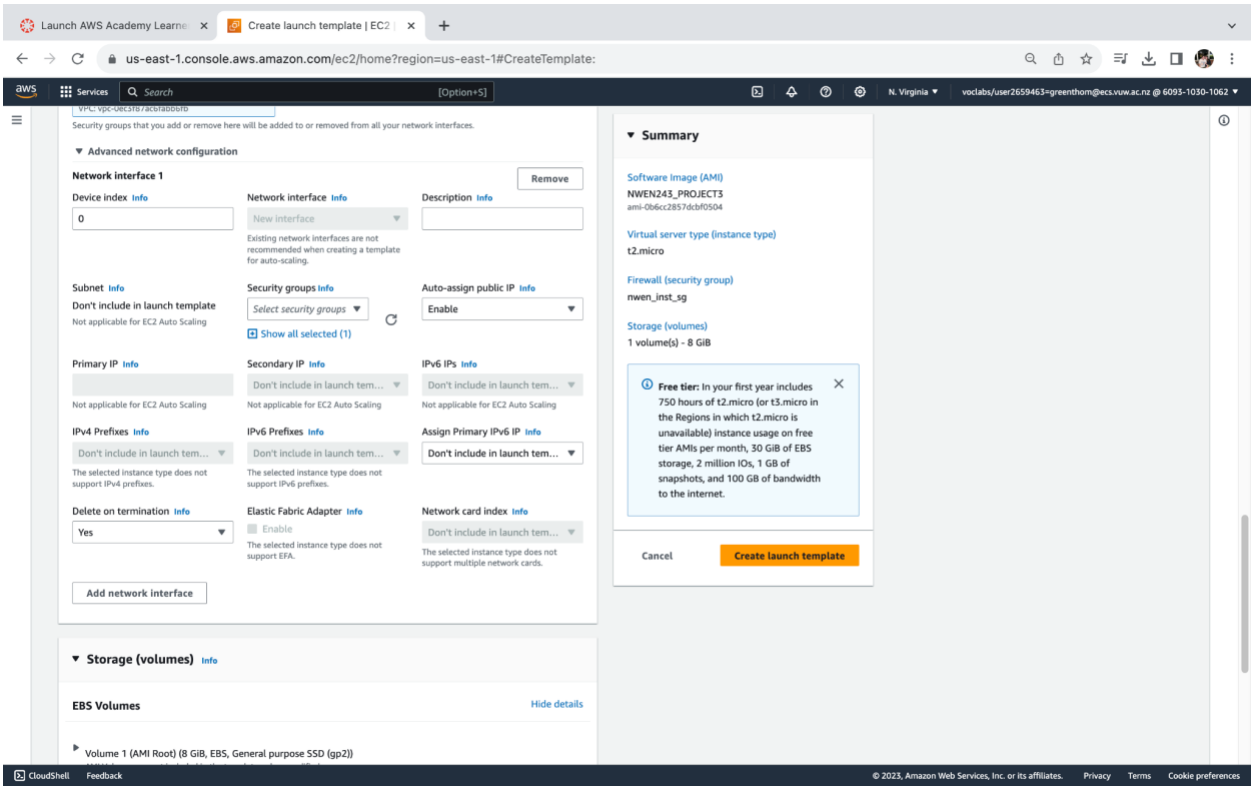
From here, we make sure our instance type is using t2.micro. We then create a new key pair for ssh login called 'nwenkeyp3'. As well as selecting our VPC, we made and added an SSH and two TCP rules, one for MusicGuruServer and one for HealthCheck. Since this was my second time doing all of this, I already had a security group called 'nwen_inst_sg.'



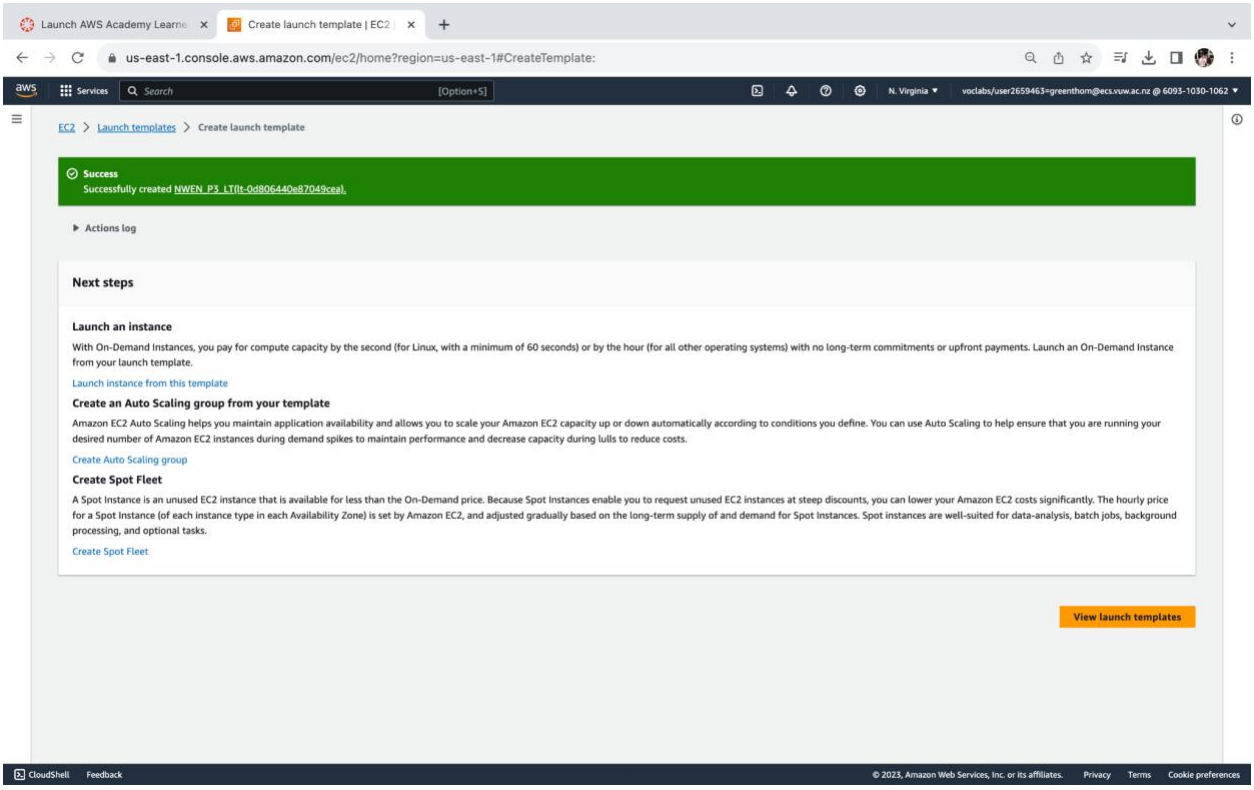
Here, we can see the security group rules showing SSH and TCP.



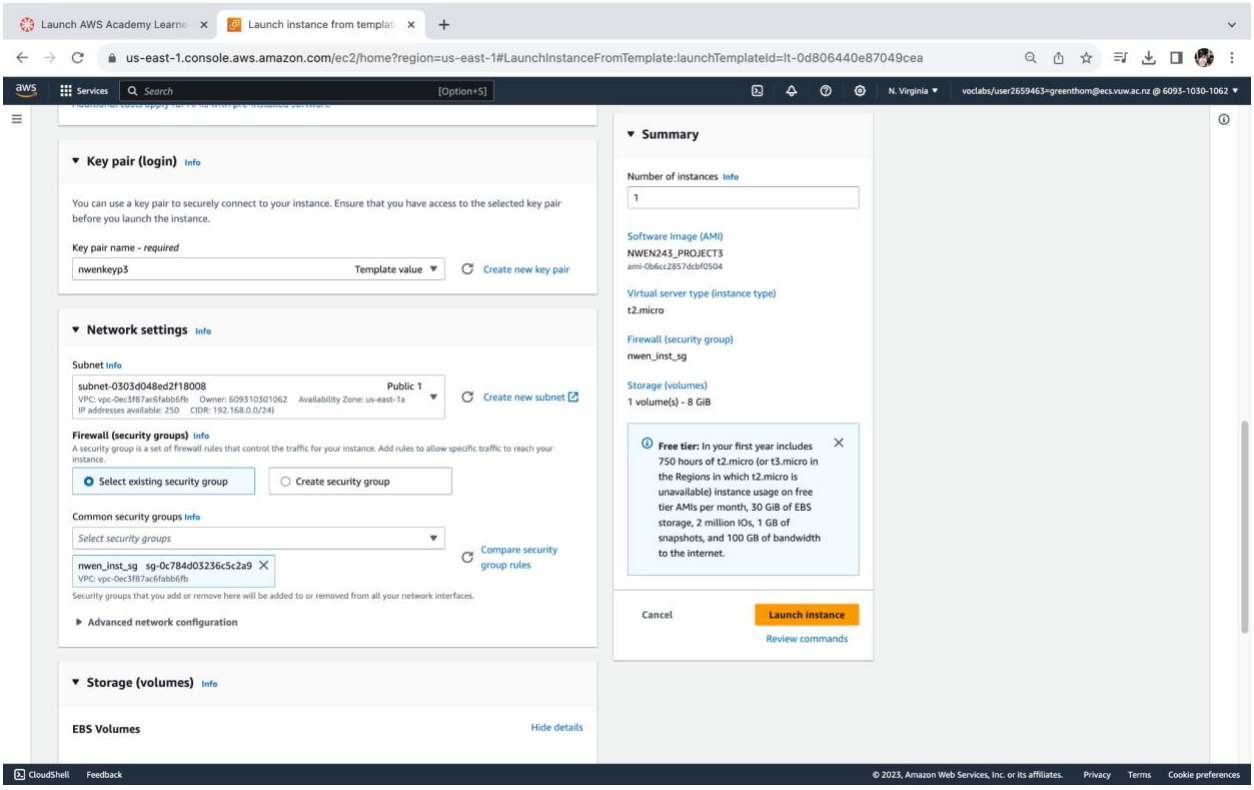
We then go to advanced network configuration and get the drop-down. We then add a network interface and enable auto-assign public IP and delete on termination.



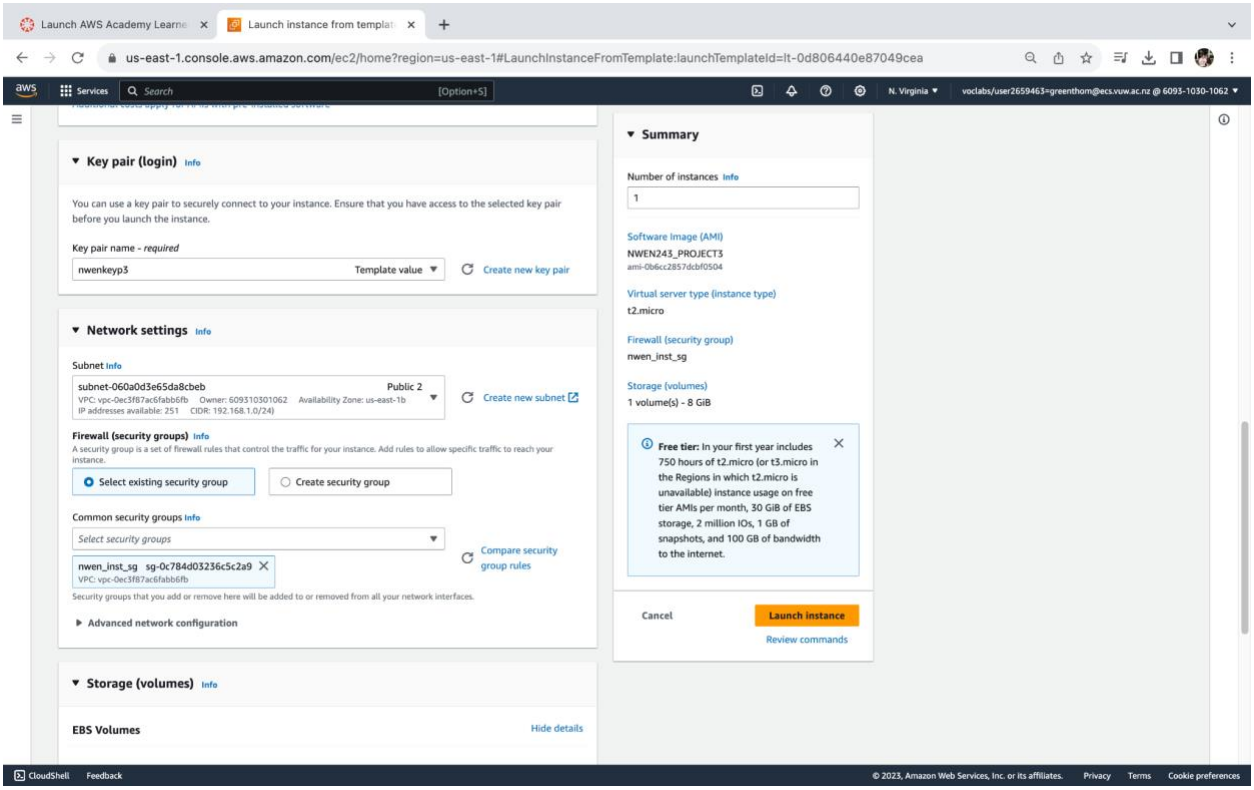
Here, we can see our Launch Template created successfully



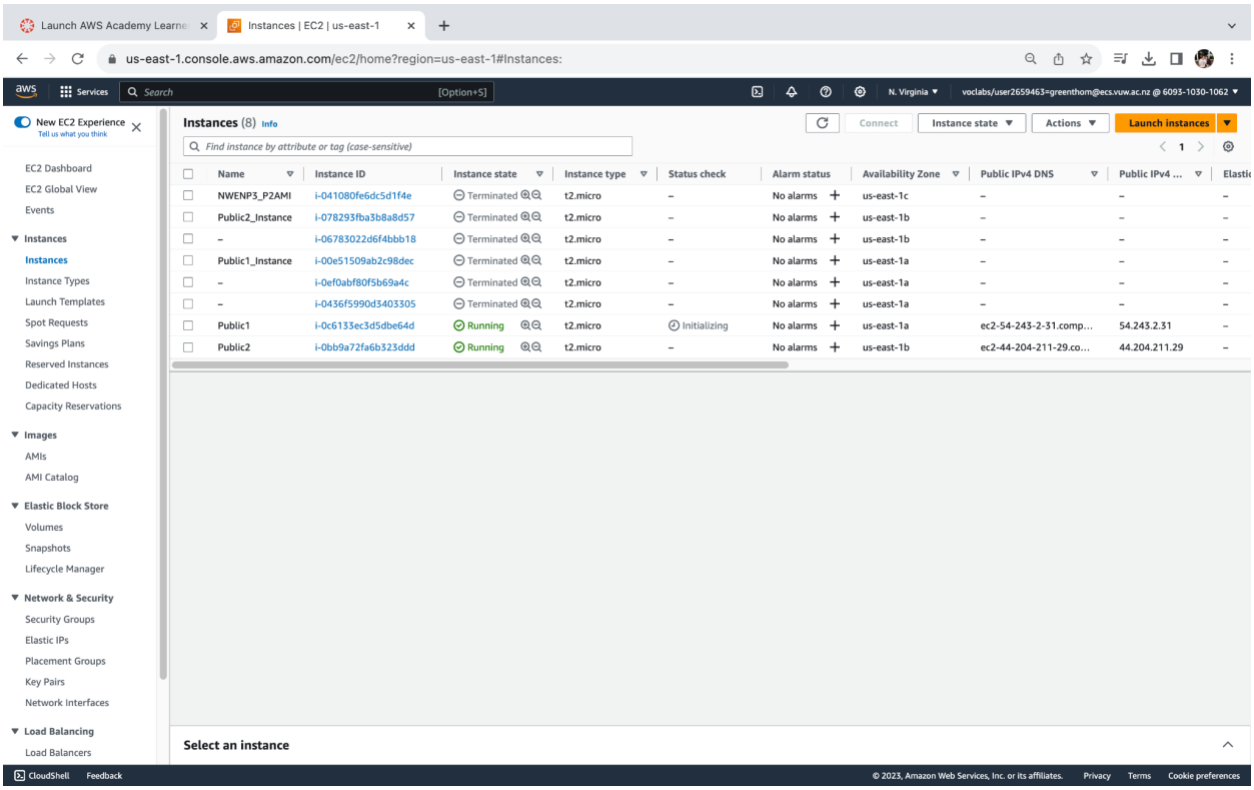
We can now test if the launch template is working. We use the public one subnet.



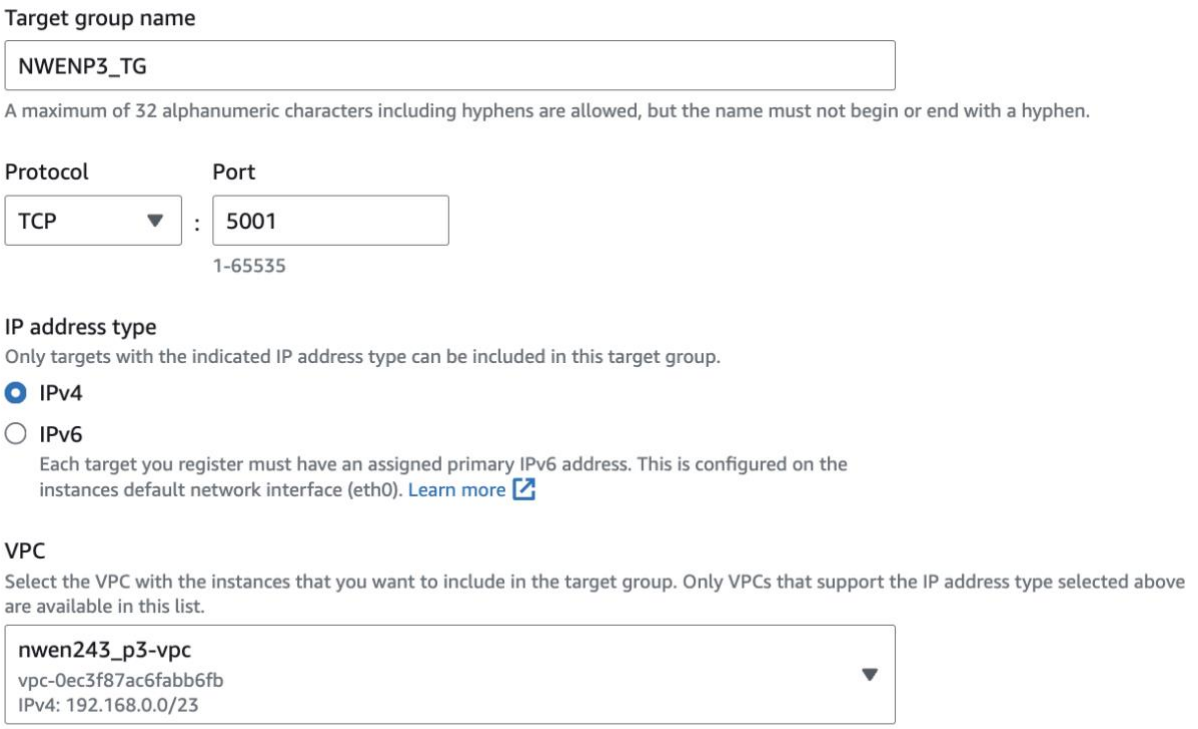
The public two subnet.



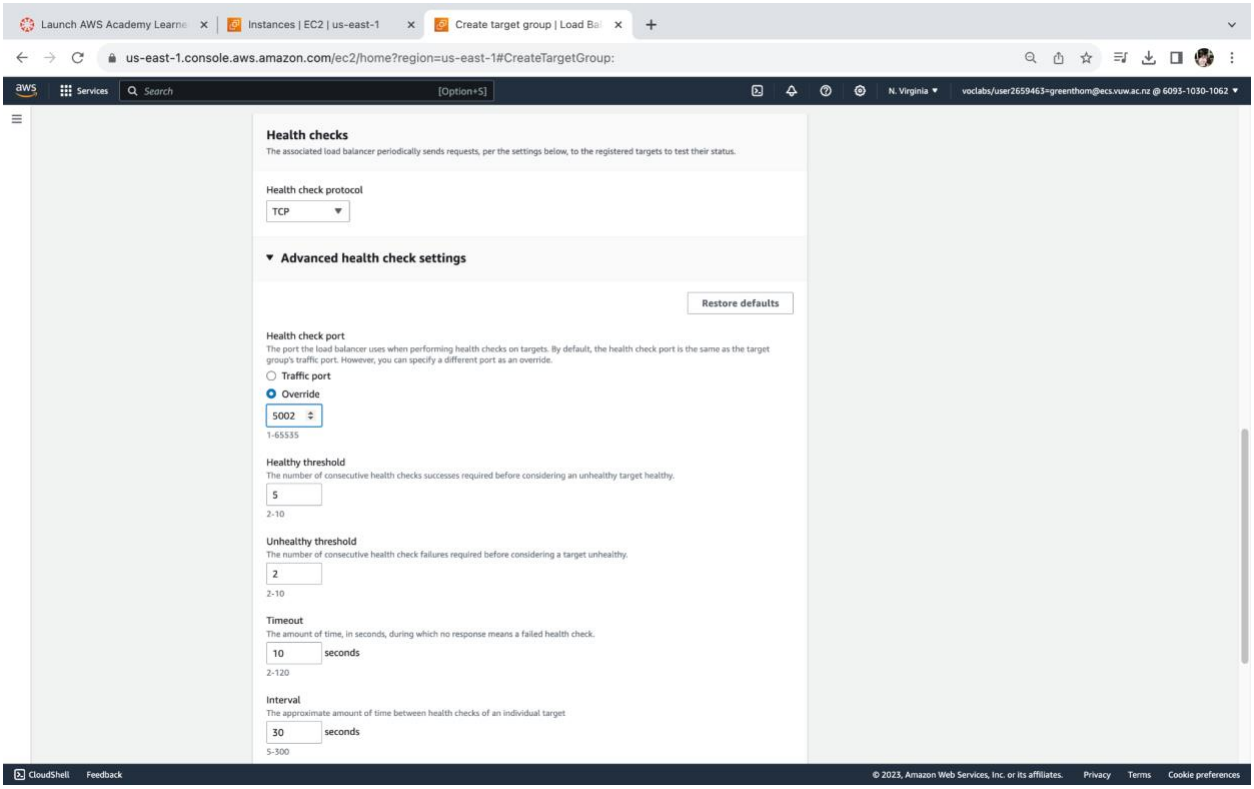
From here, we can see that Public 1 and Public 2 were created successfully at the same availability zone that our subsets are working at.



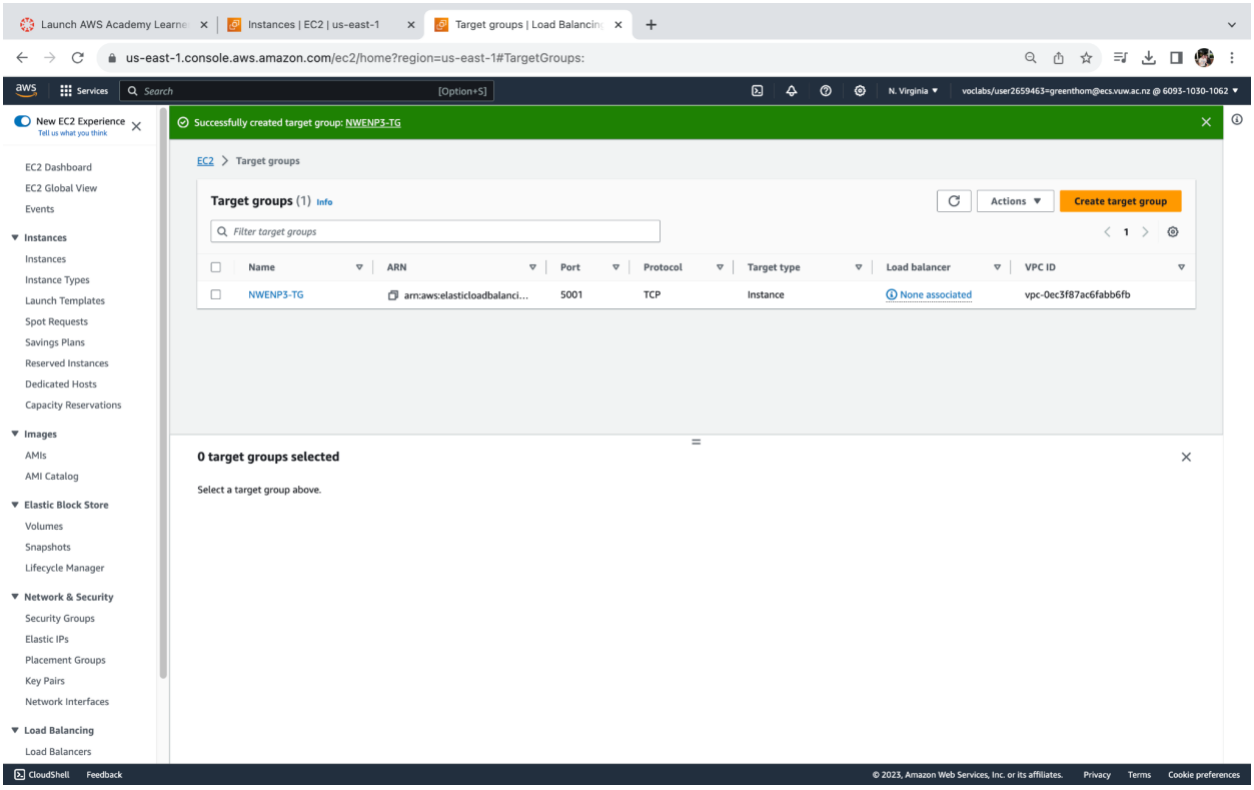
Next, we make a load balancer and a target group. Here, we are making a target group. We set the target group name to 'NWENP3_TG' and selected the protocol as TCP with the respective MusicGuruServer port number. We leave the IP address type as IPv4 and select the VPC that we made.



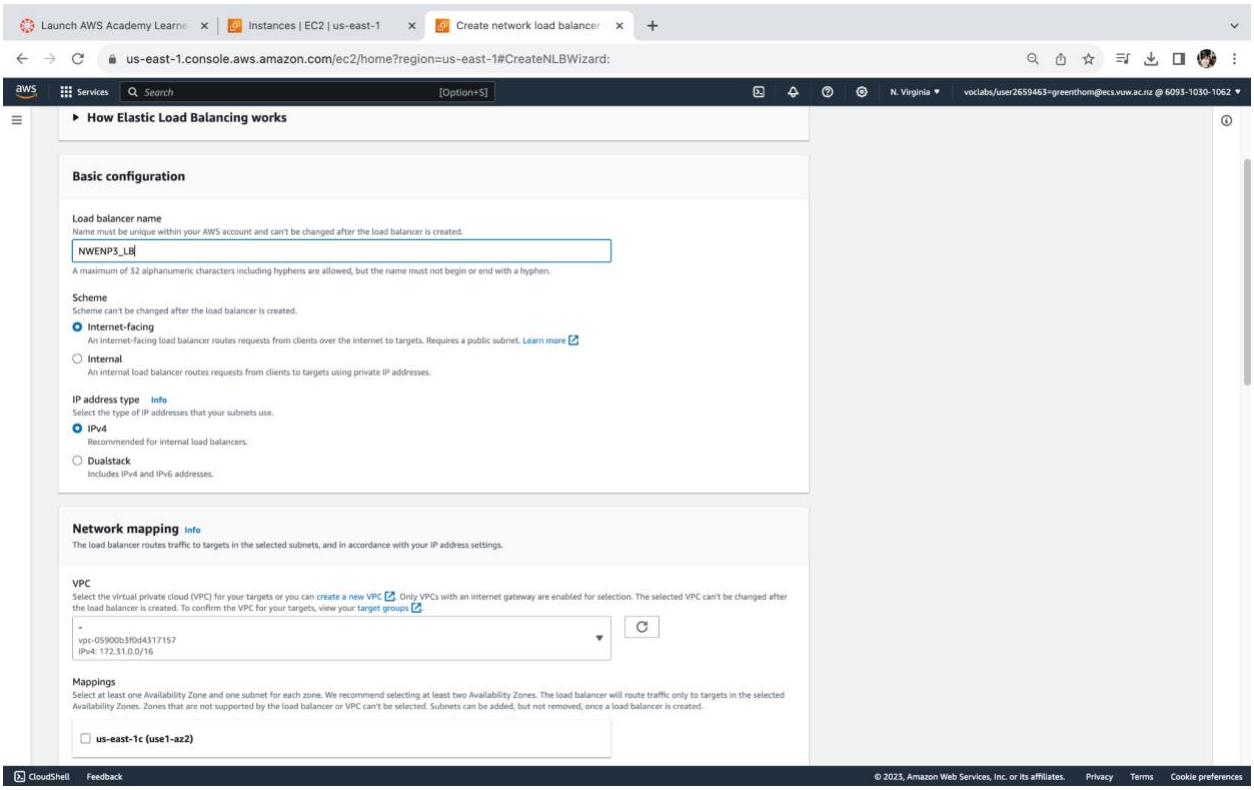
We next go and edit the health check settings. We make the HealthCheck protocol TCP. We also changed the advanced health check settings. We leave all of them to default, apart from the health check port being set to 5002; that is what we are running the TCP on.



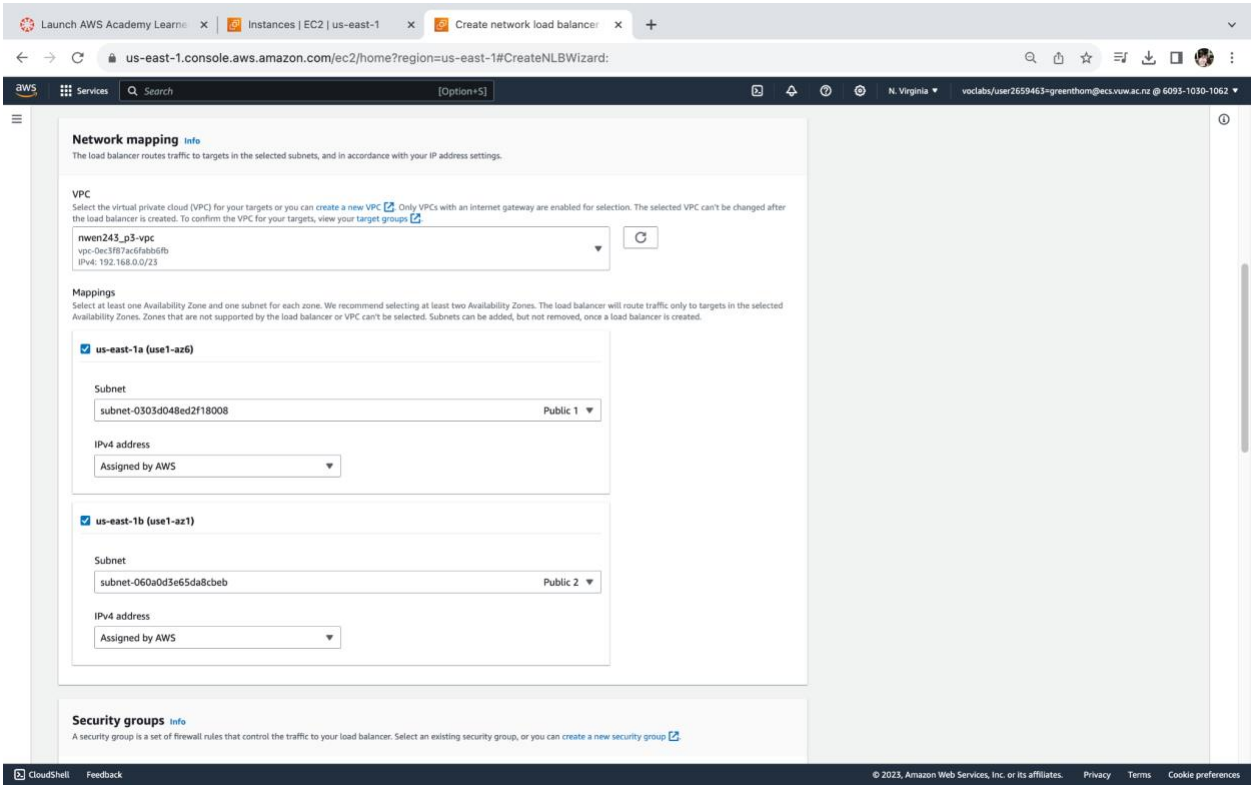
Here is the target group we created, with the corresponding port, protocol, and VPC ID selected



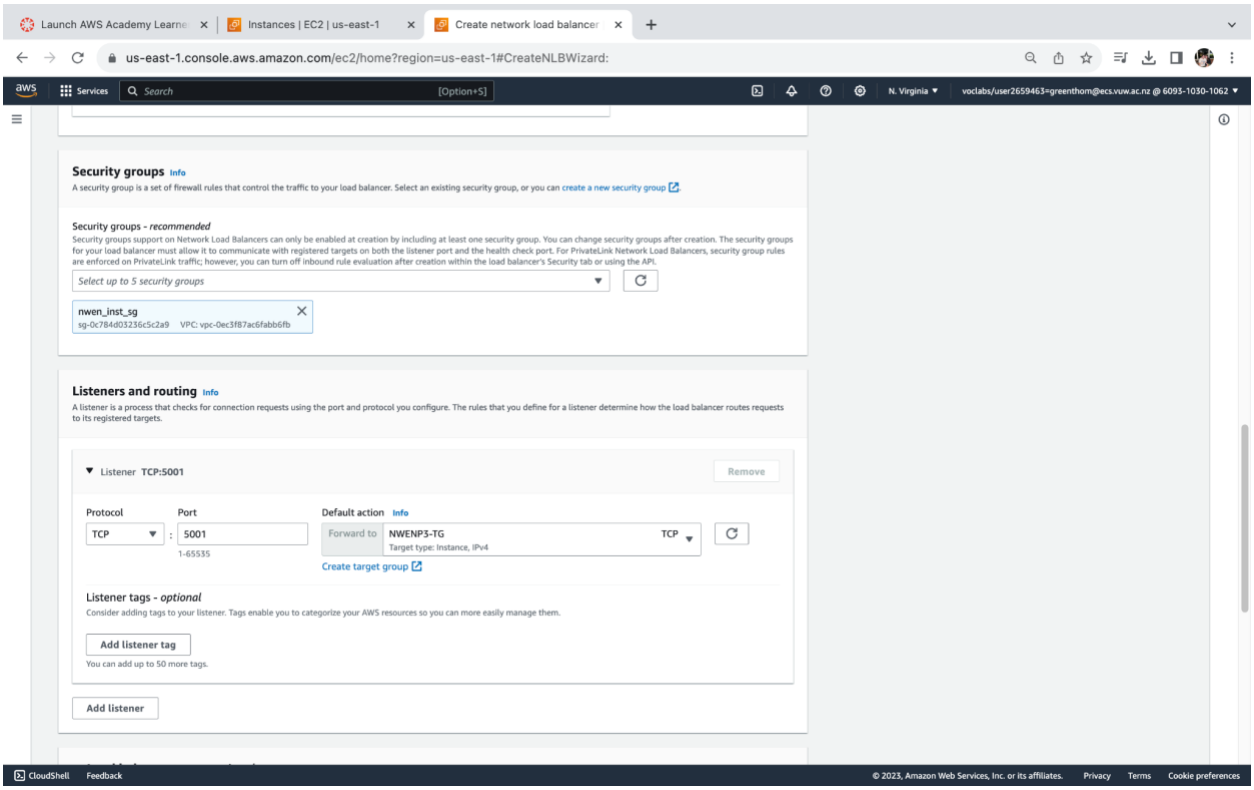
Next, we create a load balancer. We set the load balancer name as 'NWENP3_LB' and left the defaults as internet facing and IPv4.



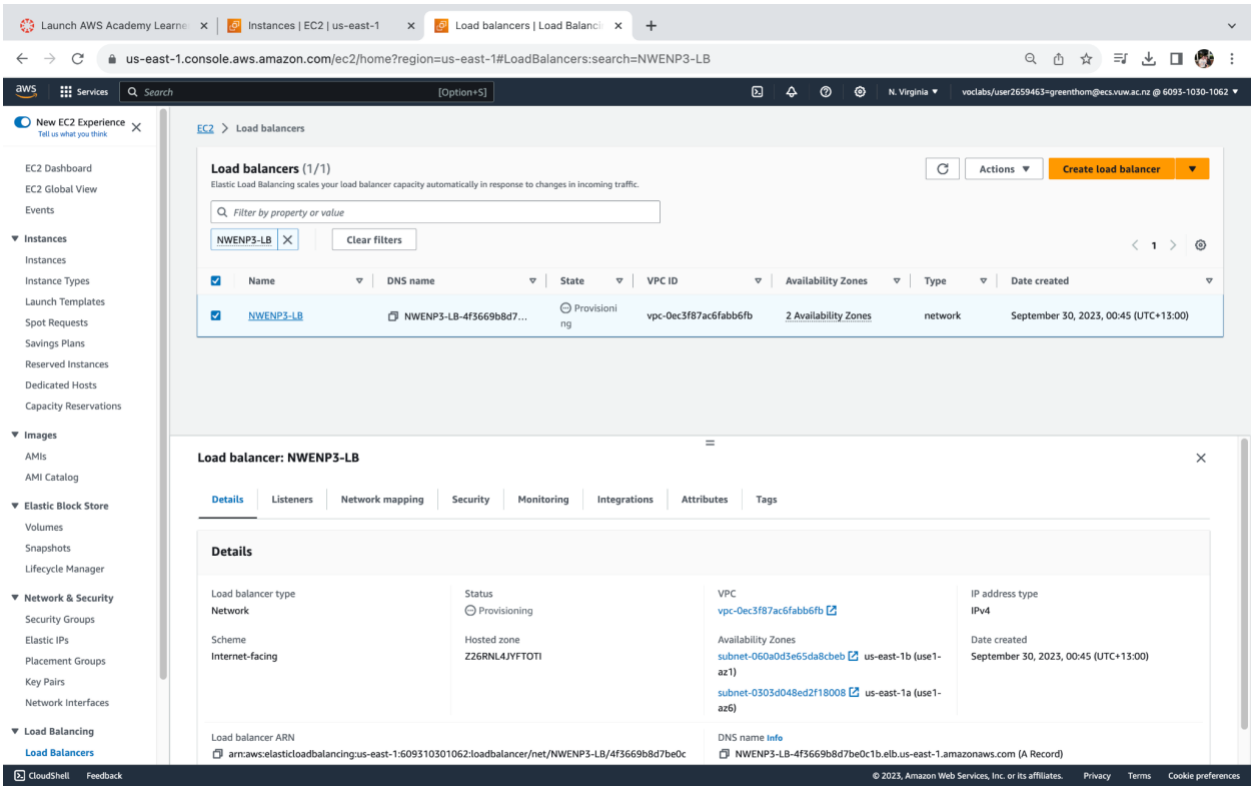
Next, we sort the network mapping. We select the VPC we created. We sort the mappings by selecting the Public 1 or Public 2 that we also created earlier. We also leave the IPv4 addresses assigned by AWS.



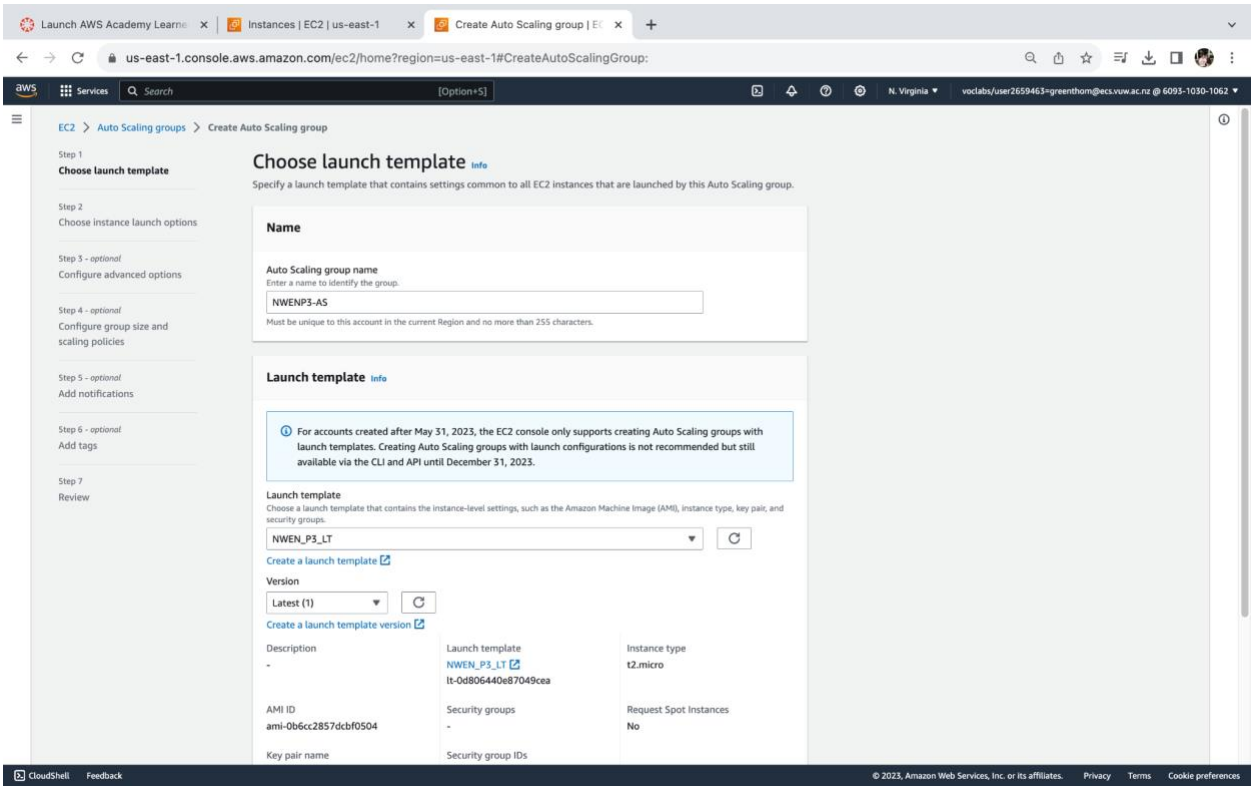
Next, we add the security group we made (We did this before updates were made to the script, so did not add the default one as well). Next, we sort listeners and routing; for the protocol, leave it at TCP and port at 5001 (MusicGuruServer) and choose the target group we made (NWENP3-TG)



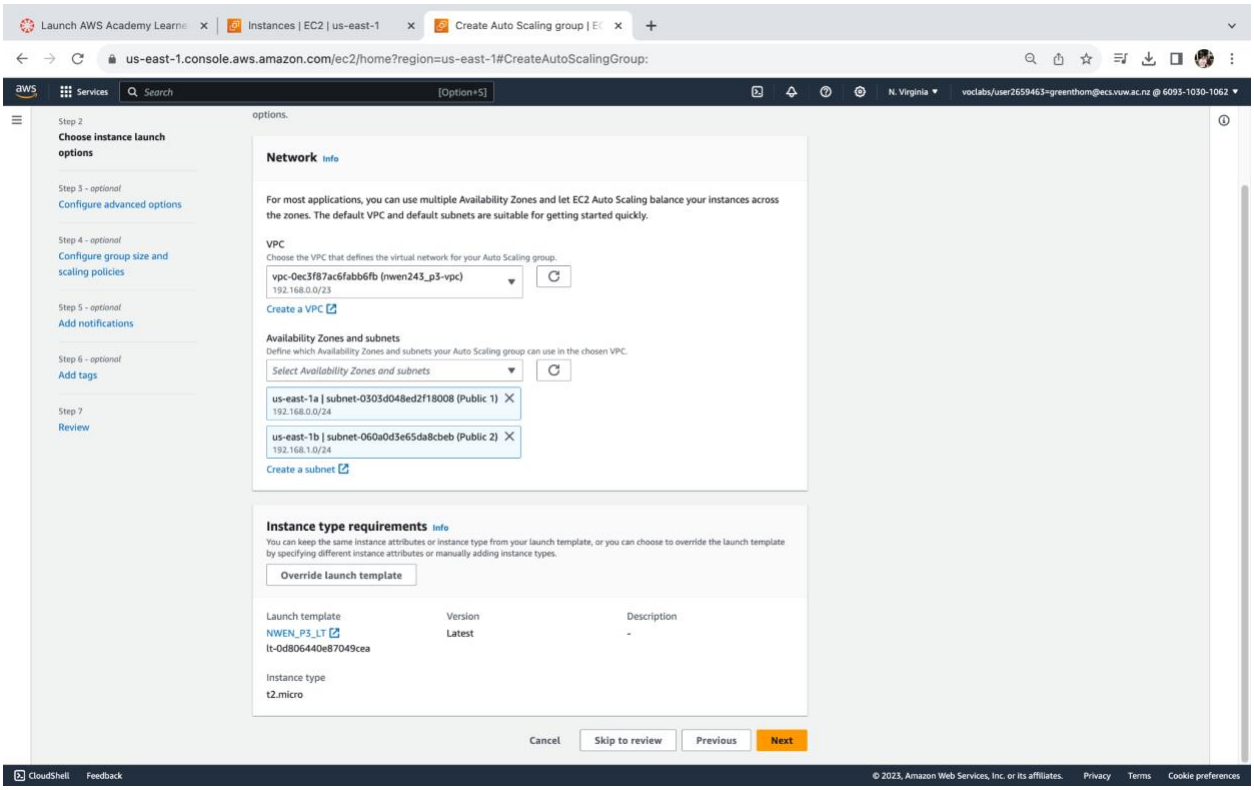
Here, our created load balancer (NWENP3-LB) shows a provisioning state.



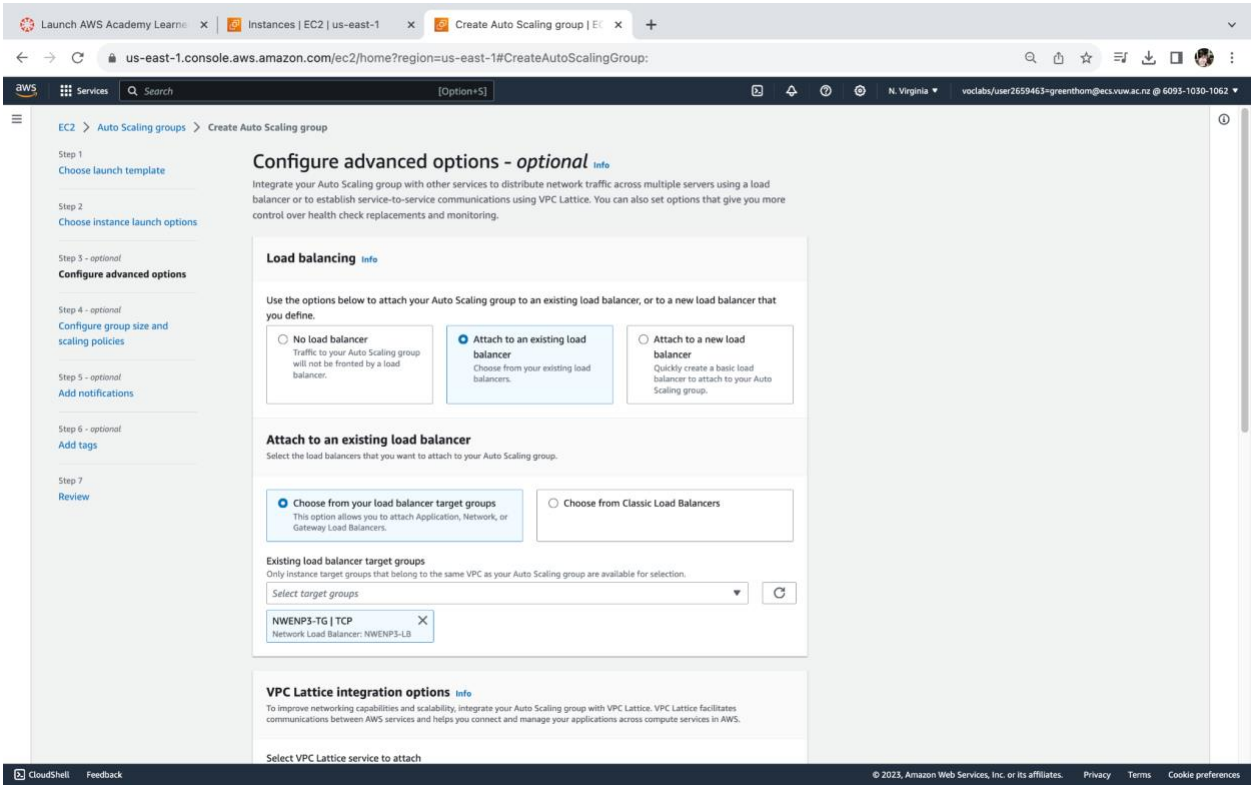
Next, we create an auto-scaling group. We named the auto-scaling group 'NWENP3-AS' and selected the launch template we had just made. We changed the version to Latest(1). We then click next.



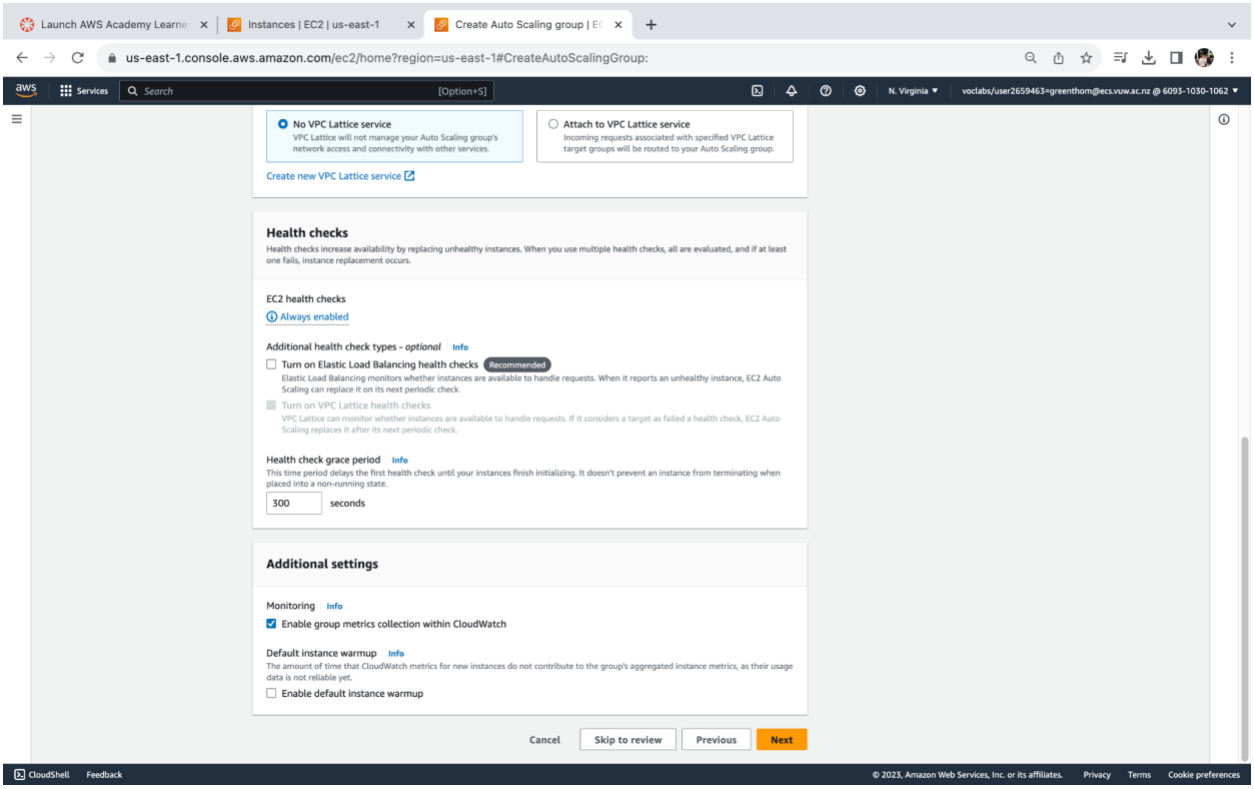
We then assign the VPC we created and the Public 1 and 2 subsets.



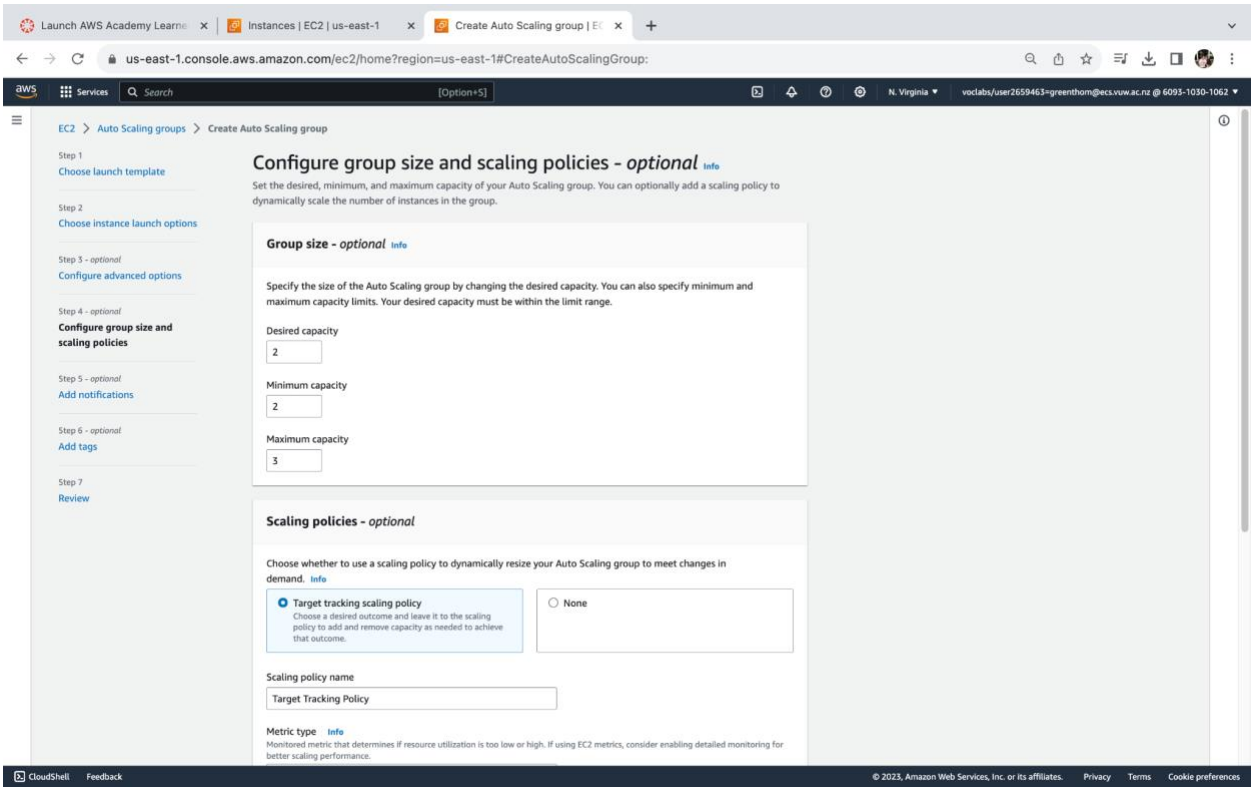
Now we select to attach to an existing load balancer, where we then do and attack it to the target group we made earlier (NWENP3-TG)



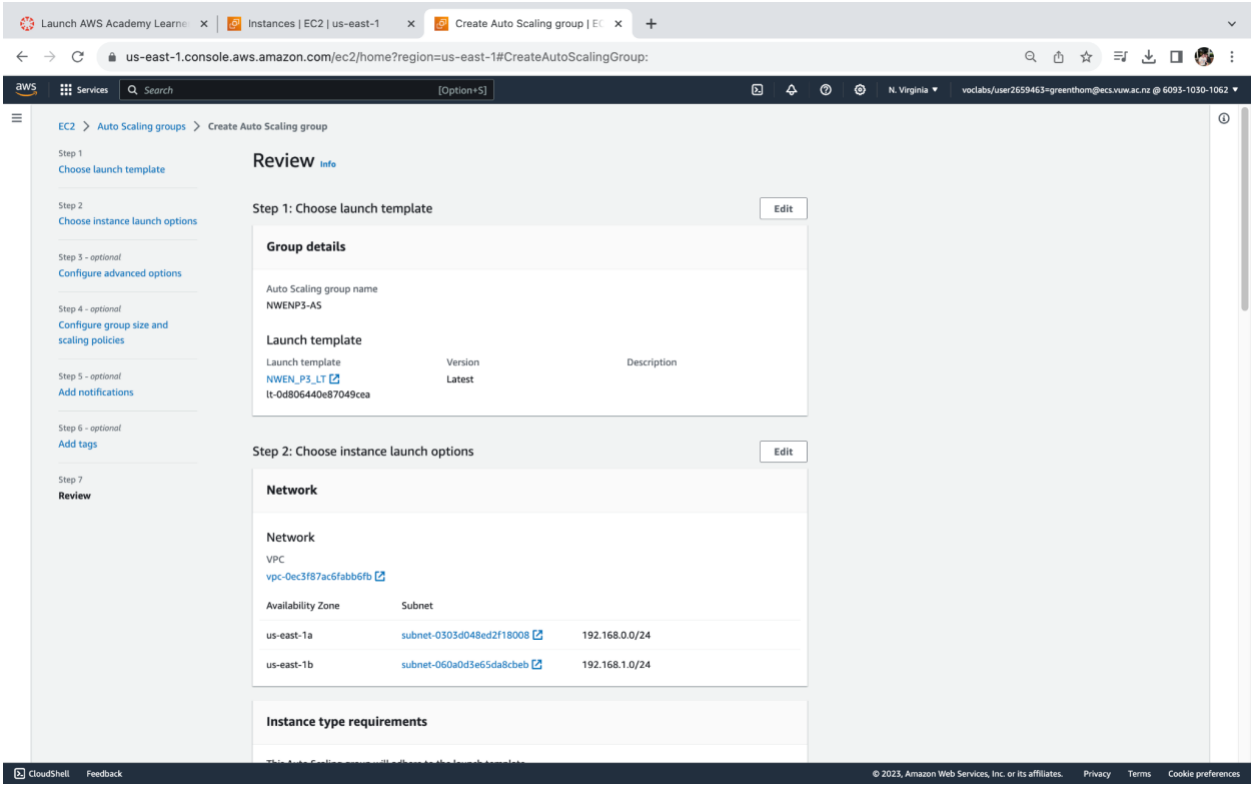
We then scroll down and do not turn on health checks, and we enable group metrics collection within Cloudwatch



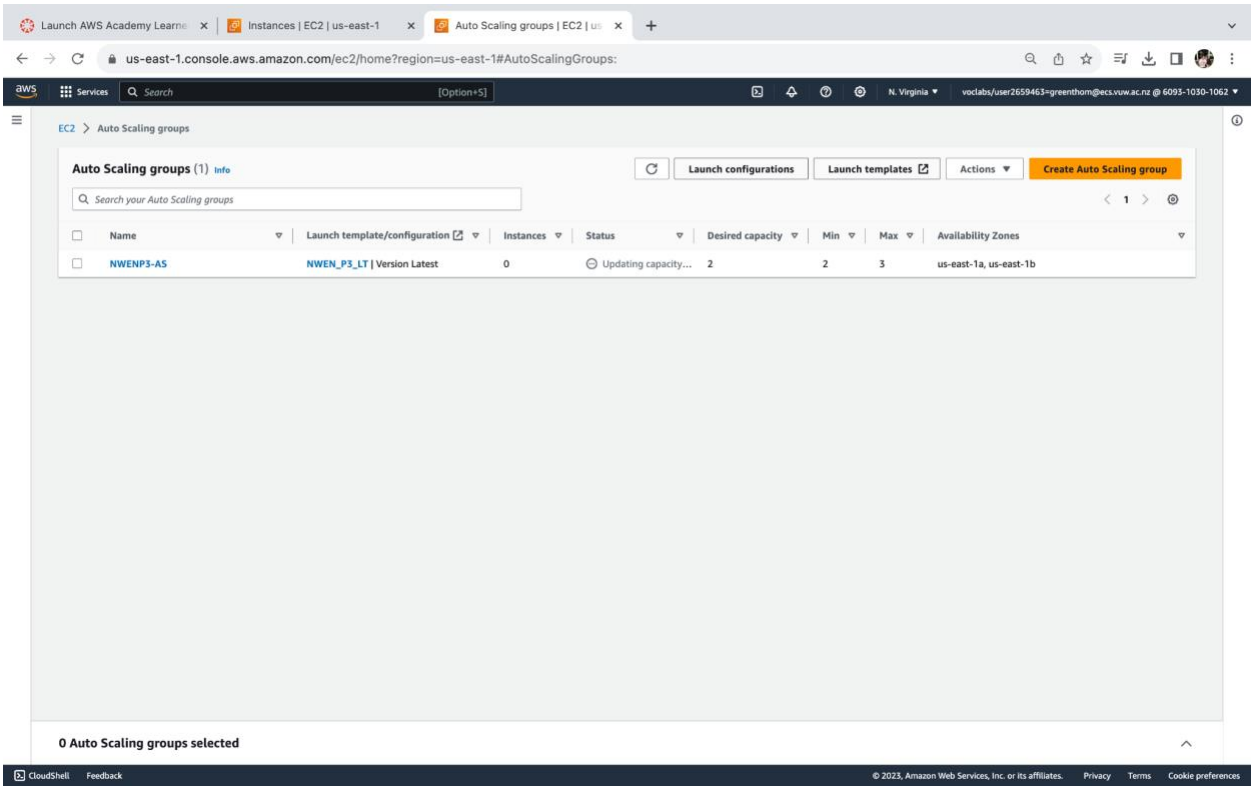
Next, we configure the group size and scaling policies. We set our desired capacity at 2, our minimum capacity at two, and our maximum capacity at 3. This provides the number of instances that will be made. Next, we turn it on for our scaling policies by selecting the target tracking scaling policy. We leave it as default.



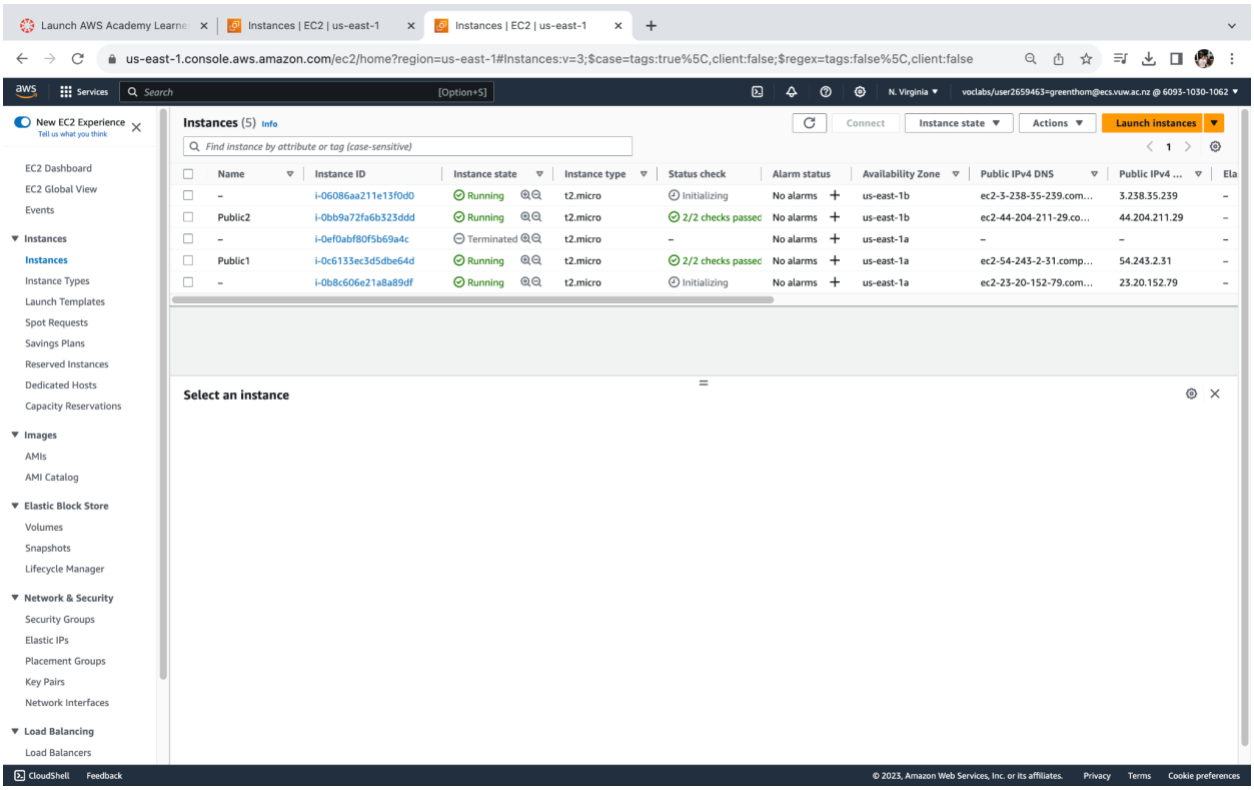
We then skip through notifications and tags and look at the review.



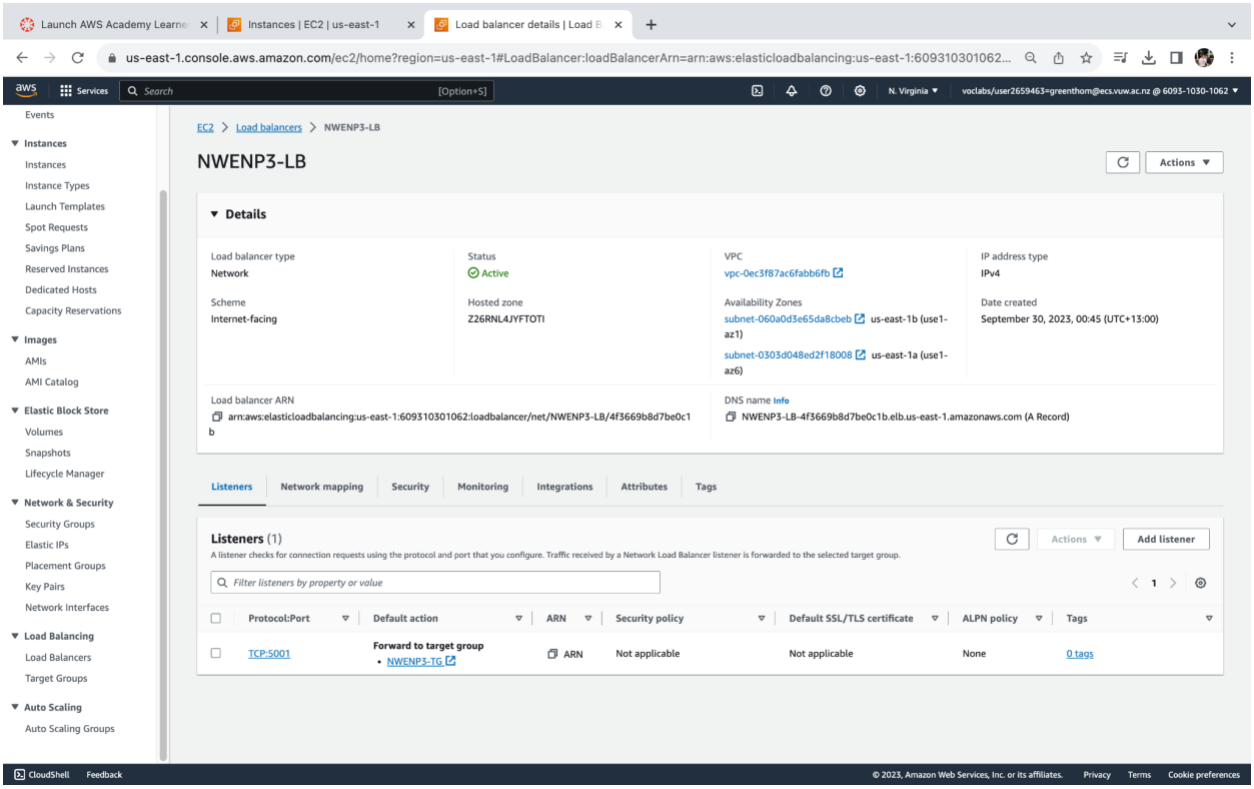
Here, we can see the autoscaling group we just created



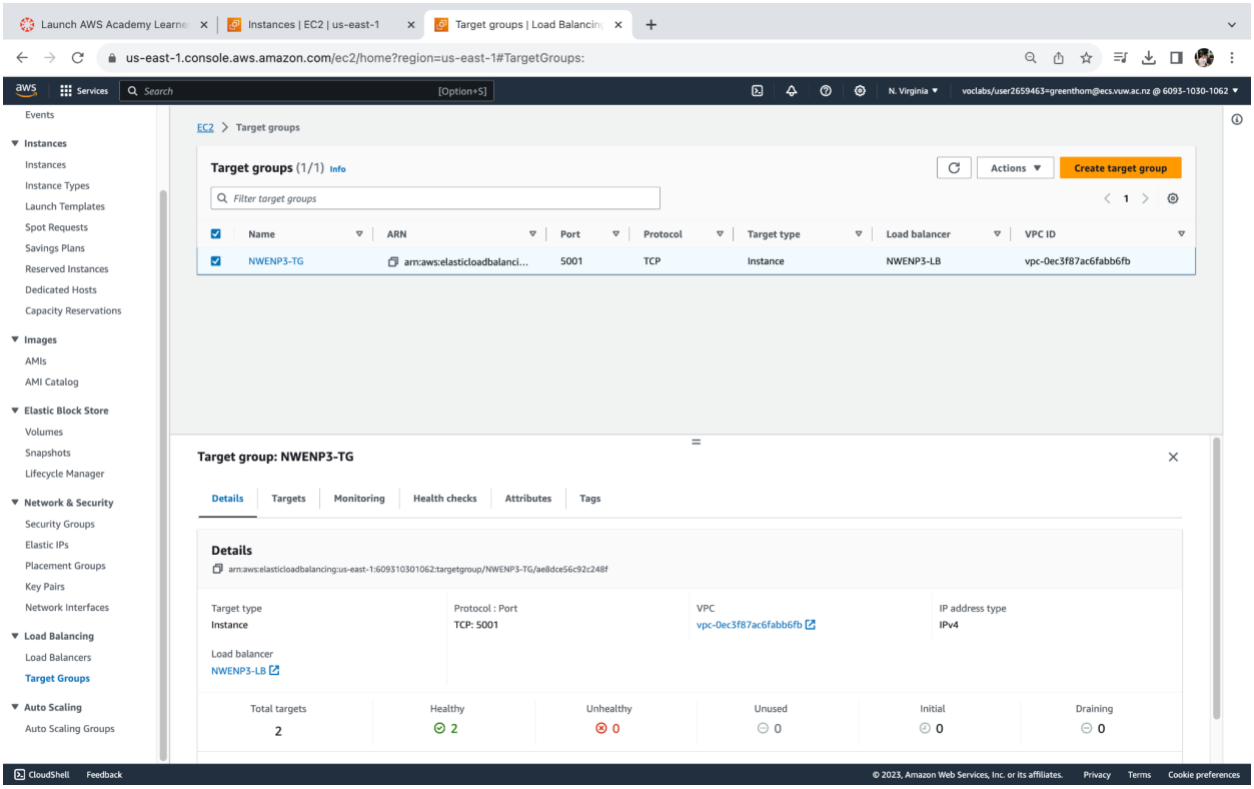
Here, we can see the newly created instances that match my policies.



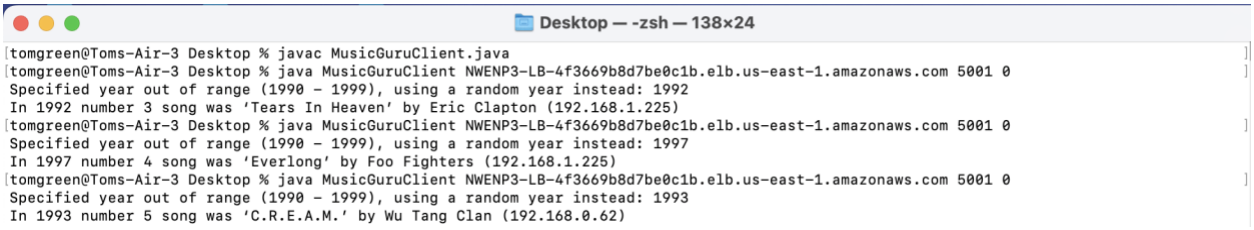
Next, we go onto the load balancer we created (NWENP3-LB) and copy the DNS name. We use this later when testing that our code works.



We then go to our target groups to find the health status of the instances created. As we can see, they are healthy.



Next, we use the DNS name info we got before to check our MusicGuruClient runs with these instances created, as we can see the IP address of the responder changes.

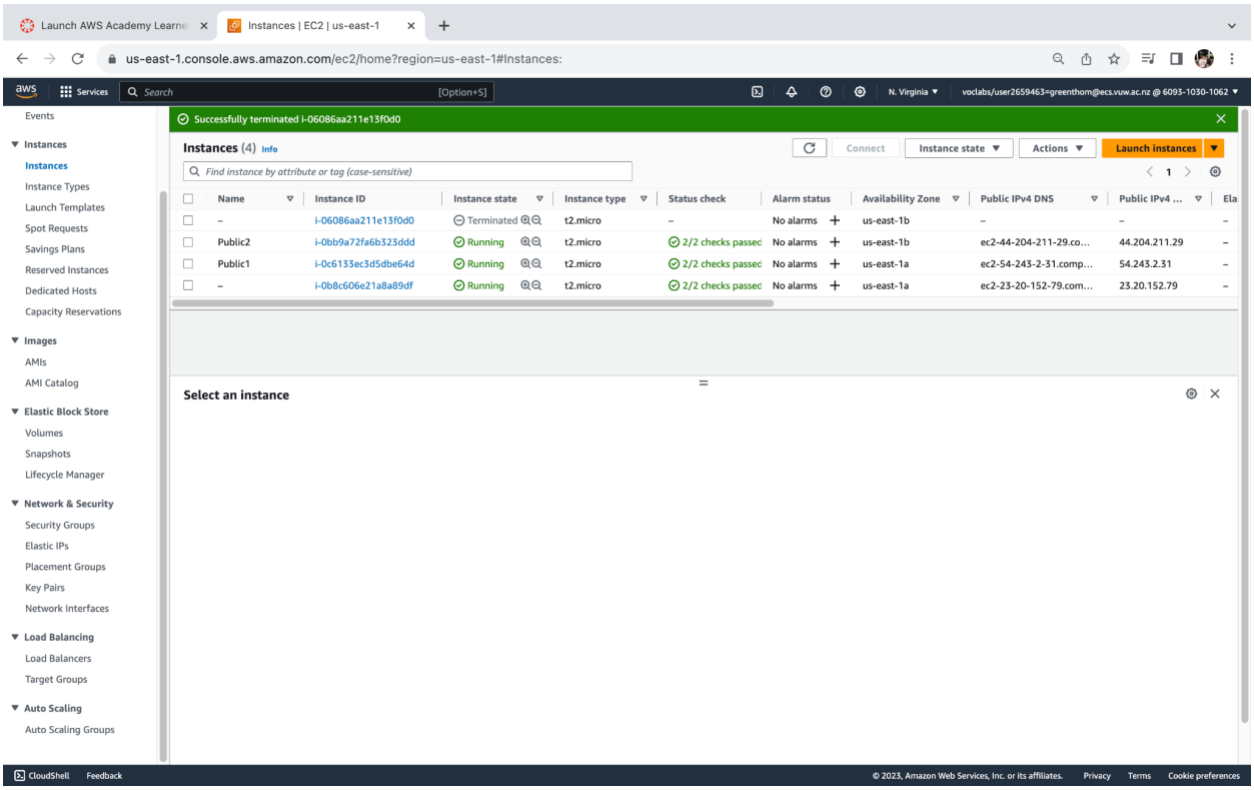


QUESTIONS

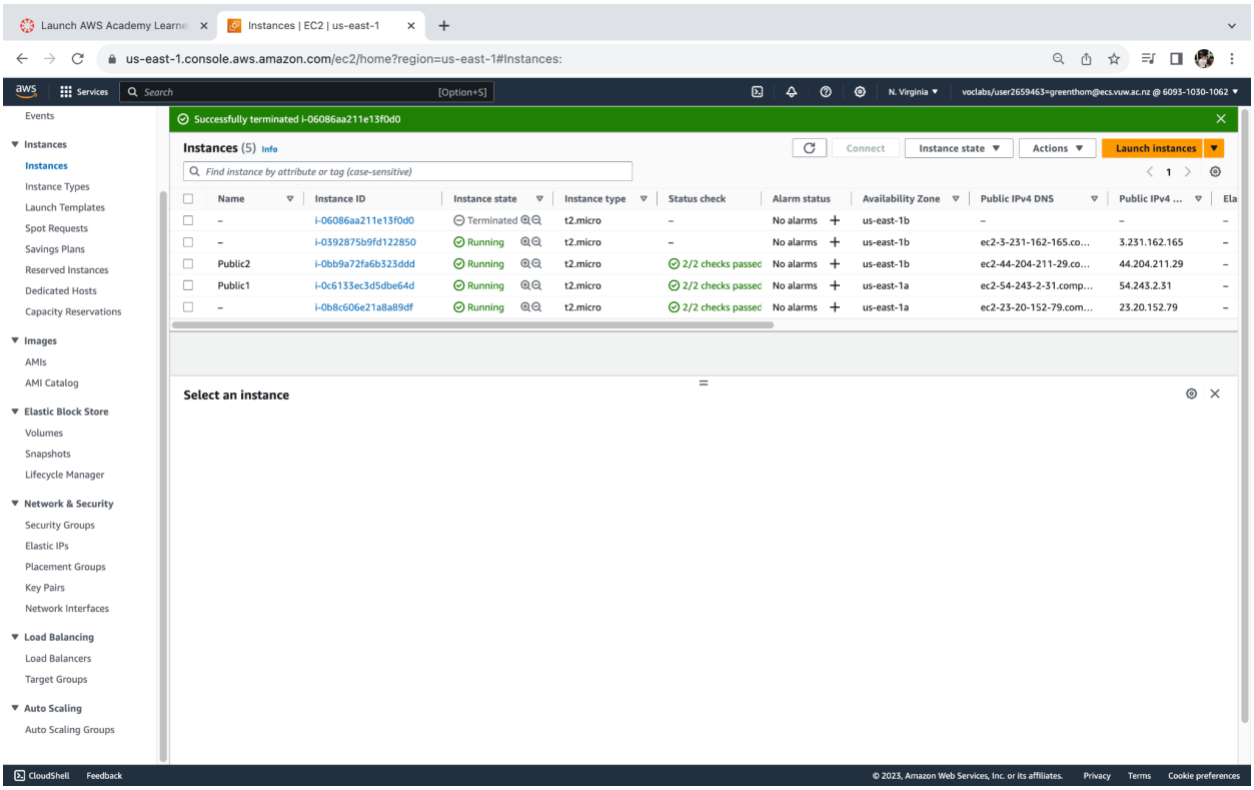
1. **Is there any pattern? Experiment a little to find out.**

Based on the output, we can see that the requests made to the MusicGuruService through the load balancer are distributed between two different IP addresses (182.168.1.225 and 192.168.0.62). This is due to two instances serving requests, and as we can see, these instances are randomly providing us an output when connecting to the server. The load balancer seems to evenly distribute requests across instances, as there isn't a consistent pattern of requests going to a specific instance. Instances with IP addresses 192.168.1.225 and 192.168.0.62 are both handling requests, indicating that the load balancer is appropriately directing traffic to these instances.

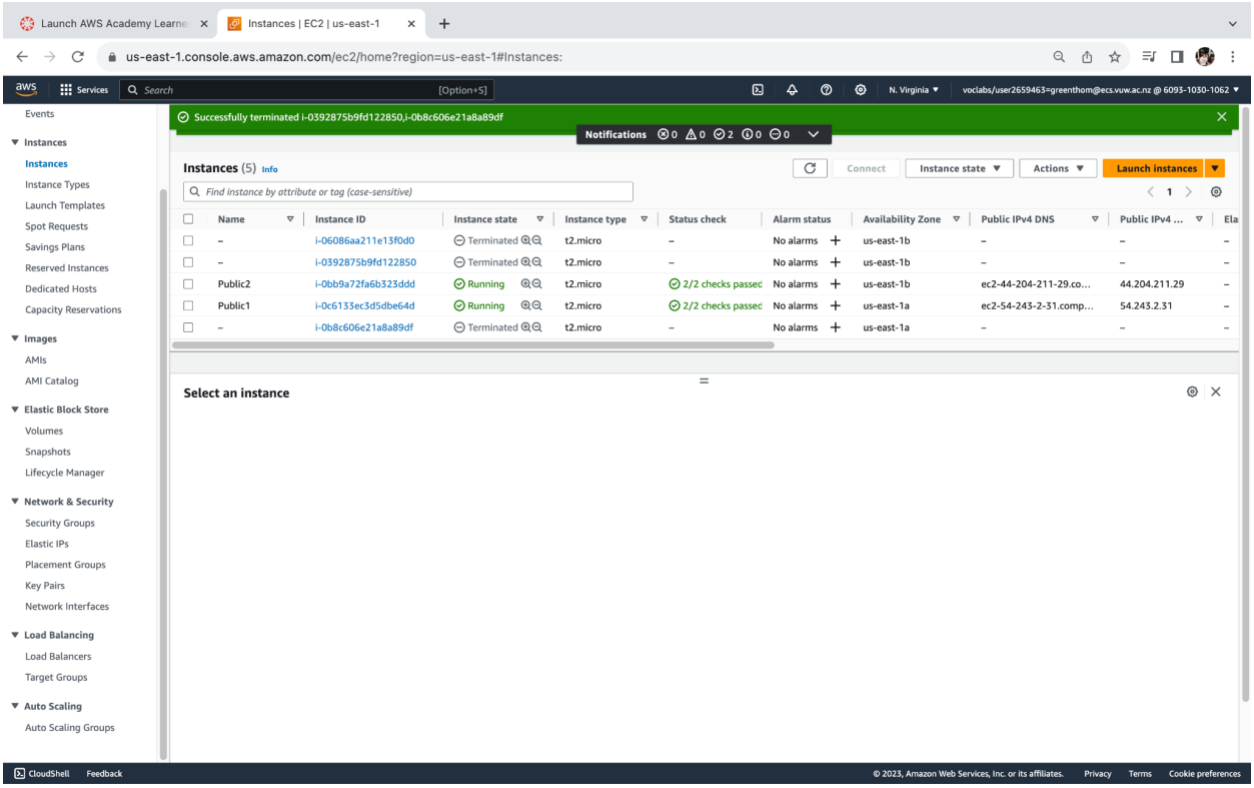
Here, we can see that we have terminated one instance operating our 'Public 2' subnet indicated as running in the availability zone us-east-1b. This provided me only the ability to run the 'Public 1' subnet, giving me the same IP each time due to only one instance running my MusicGuruServer.



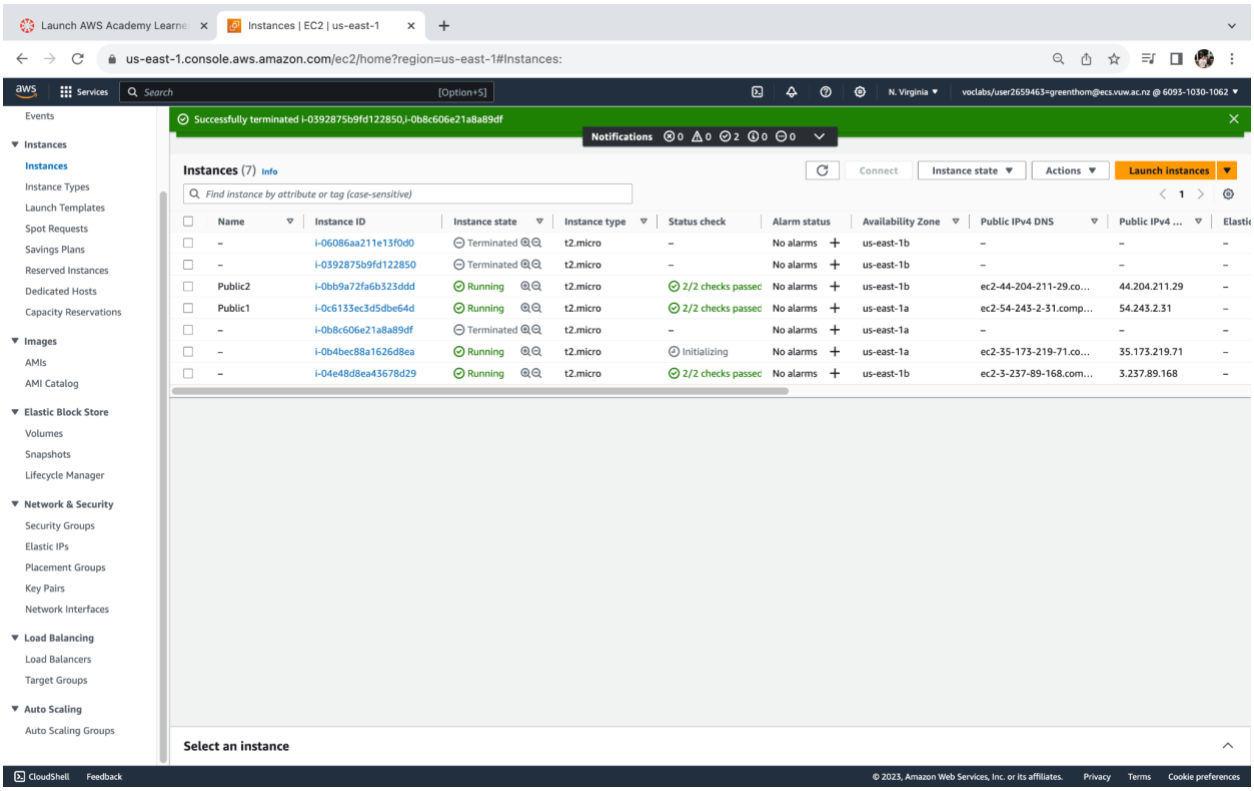
However, when I terminate an instance, it replaces it with a new one corresponding to the terminated instance details (Public1 or Public2).



Here, we terminated all the instances providing me with no output on my MusicGuruServer when attempting to run at this time. We terminated all the instances to see if the load balancer would create the instances again.

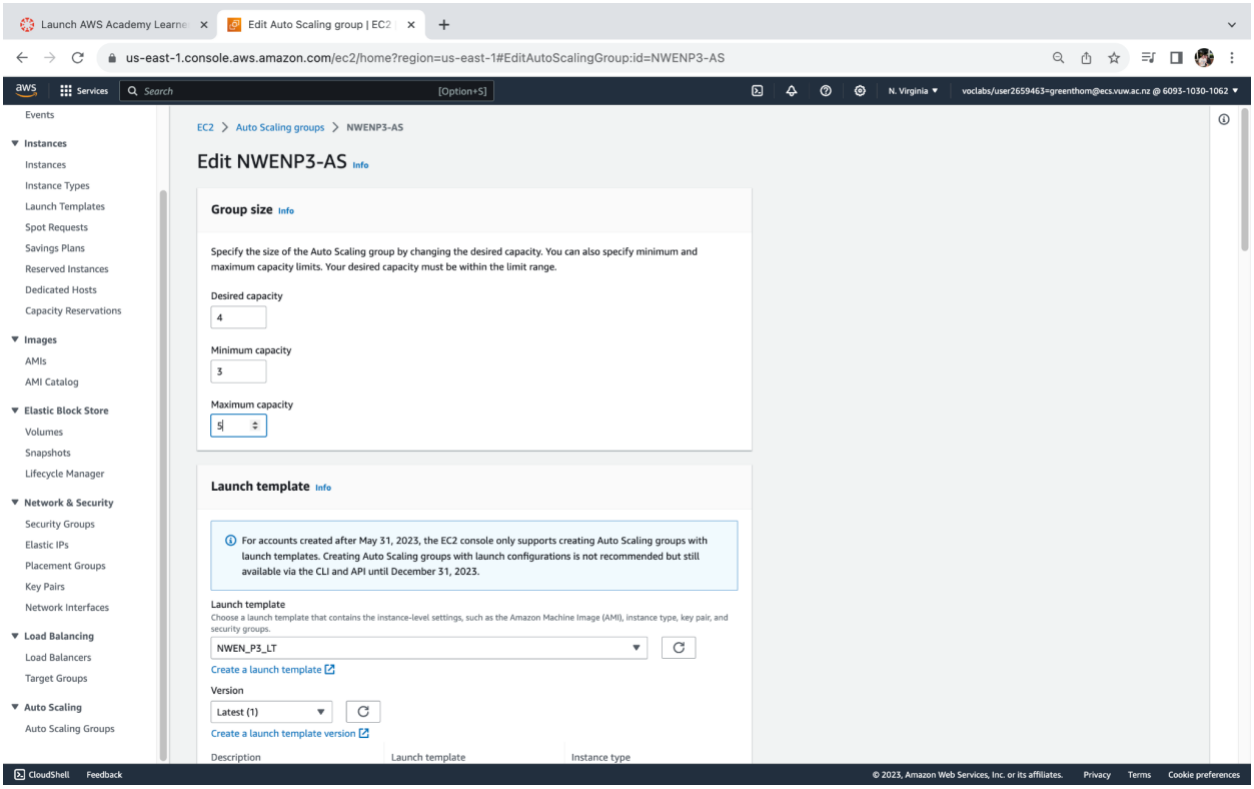


As we can see here, our load balancer is working correctly, as it made two more instances due to the originals being terminated. The auto scaling group recognized that the number of instances had fallen below the desired capacity of 2. Therefore, to maintain the desired capacity it initiated 2 new instances to replace the terminated ones reflecting the Public one and two subsets.

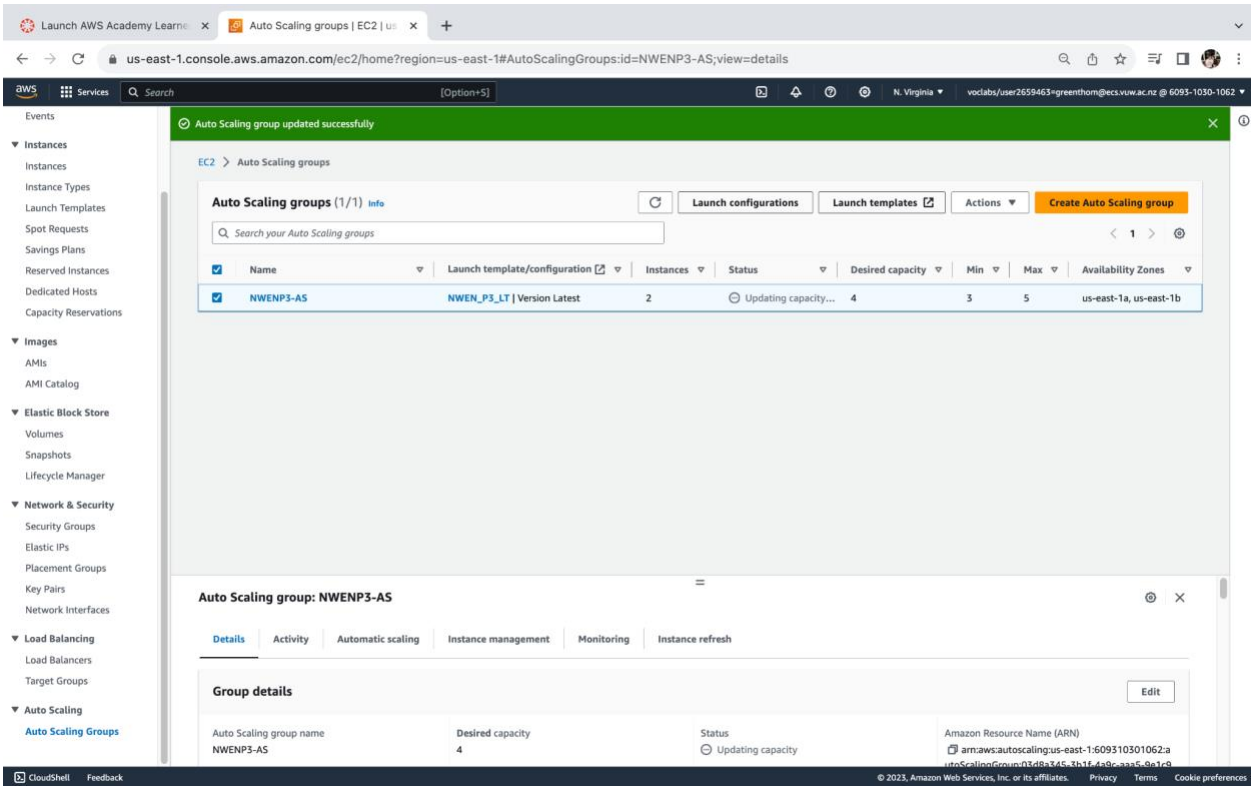


Question 2: What happens? It can take a long time - so be patient.

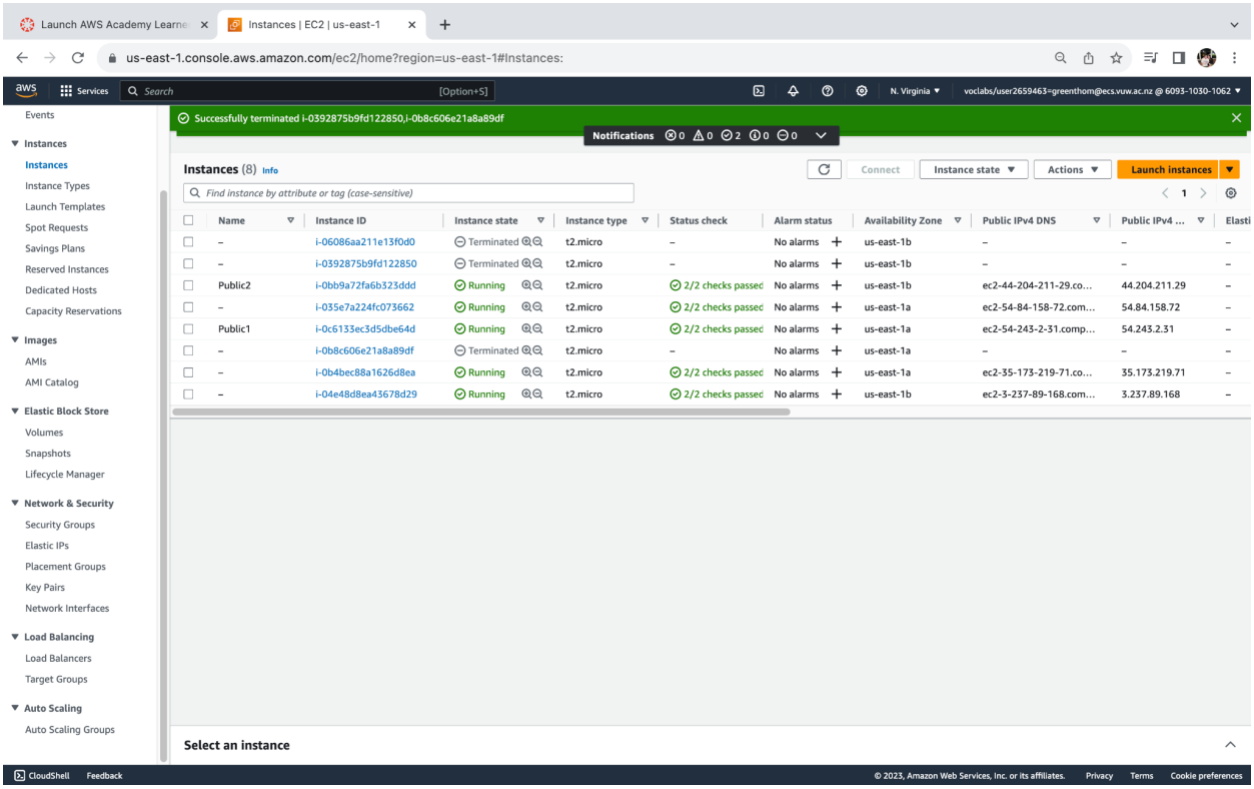
Here, we edit our auto-scaling groups to determine the desired capacity at 4. This will, therefore, increase the amount of our instances.



As you can see, our desired capacity has increased to the amount we set. Now, we can observe our instances to see if these updates.

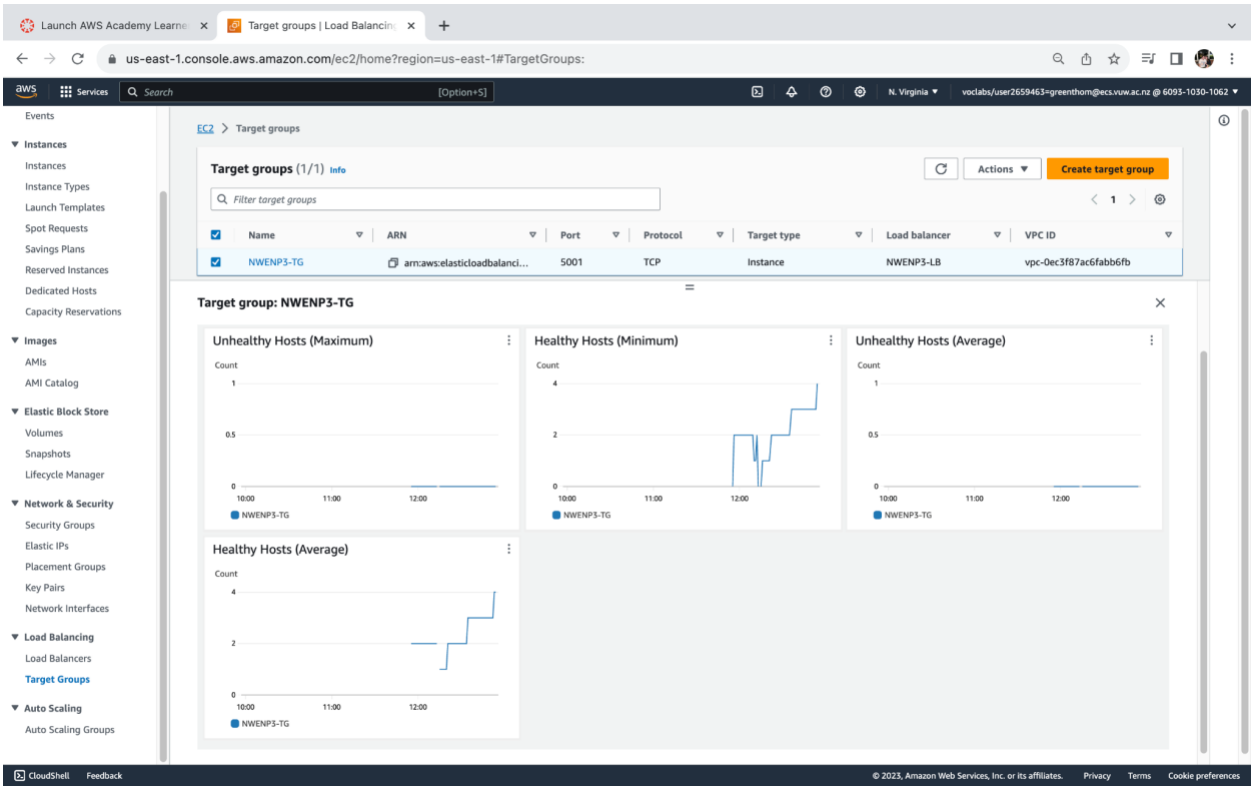


When auto scaling needs to launch a new instance, it has to create these instances based on the launch configuration I have defined. This process includes setting up the instance, installing the necessary software, and configuring it as specified. It can take some time, especially if my instances have complex setups or require specific configurations. Auto Scaling uses AMIs to create new instances. If my chosen AMI is large or not readily available, it can contribute to longer launch times. Auto scaling might introduce a delay between scaling activities.

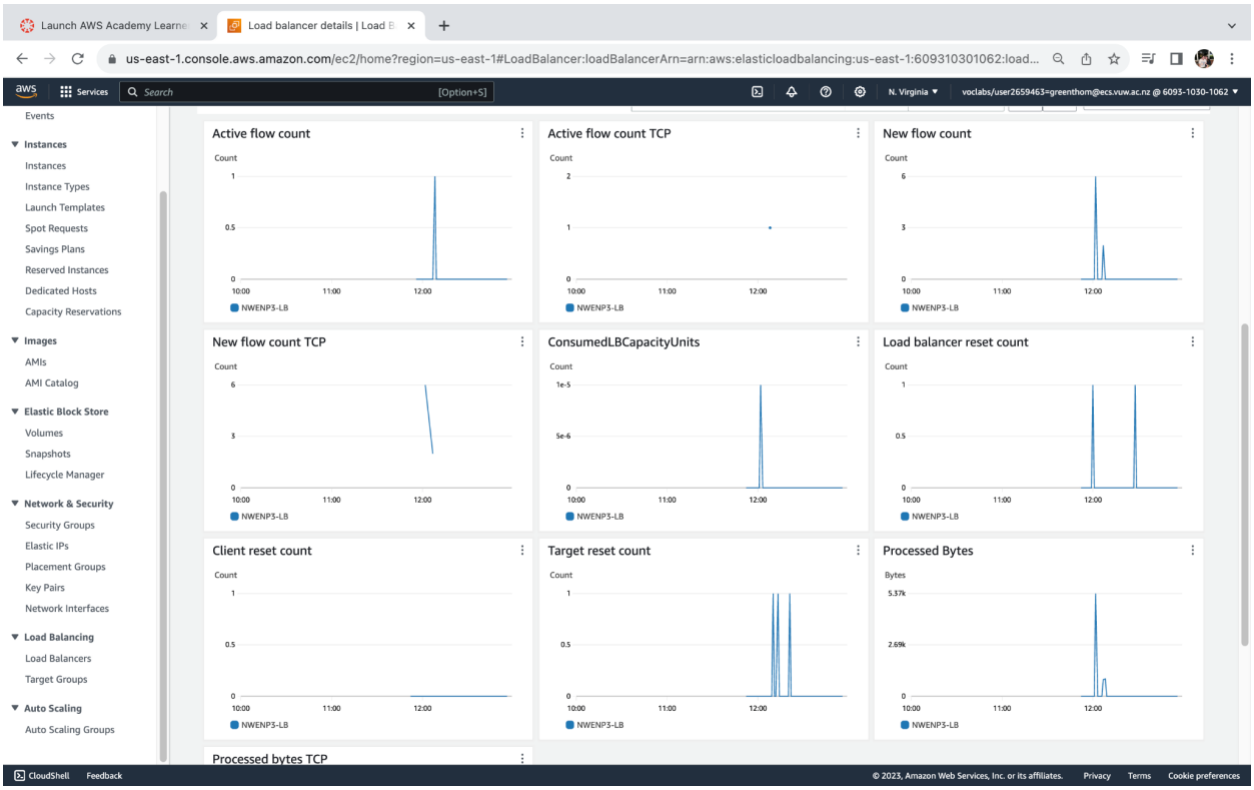


Question 3: Why is it so slow?

The termination of instances serves as the trigger for the auto-scaling service to work in response to an unmet desired capacity. The service initiates an analysis of the configured policies to determine the appropriate course of action. However, this decision-making process causes a notable slowdown as it waits for the health-checker to ascertain the health status of instances where these health checks occur at regular intervals (30 seconds). The auto-scaler operates with a health check grace period of 300 seconds, this is implemented to cause scaling cooldowns to stop rapid and potentially unnecessary scaling. These cooldown periods create time delays between scaling actions helping the environment to stabilize due to the demand of new instances. Also have using t2.micro which has limited CPU power.



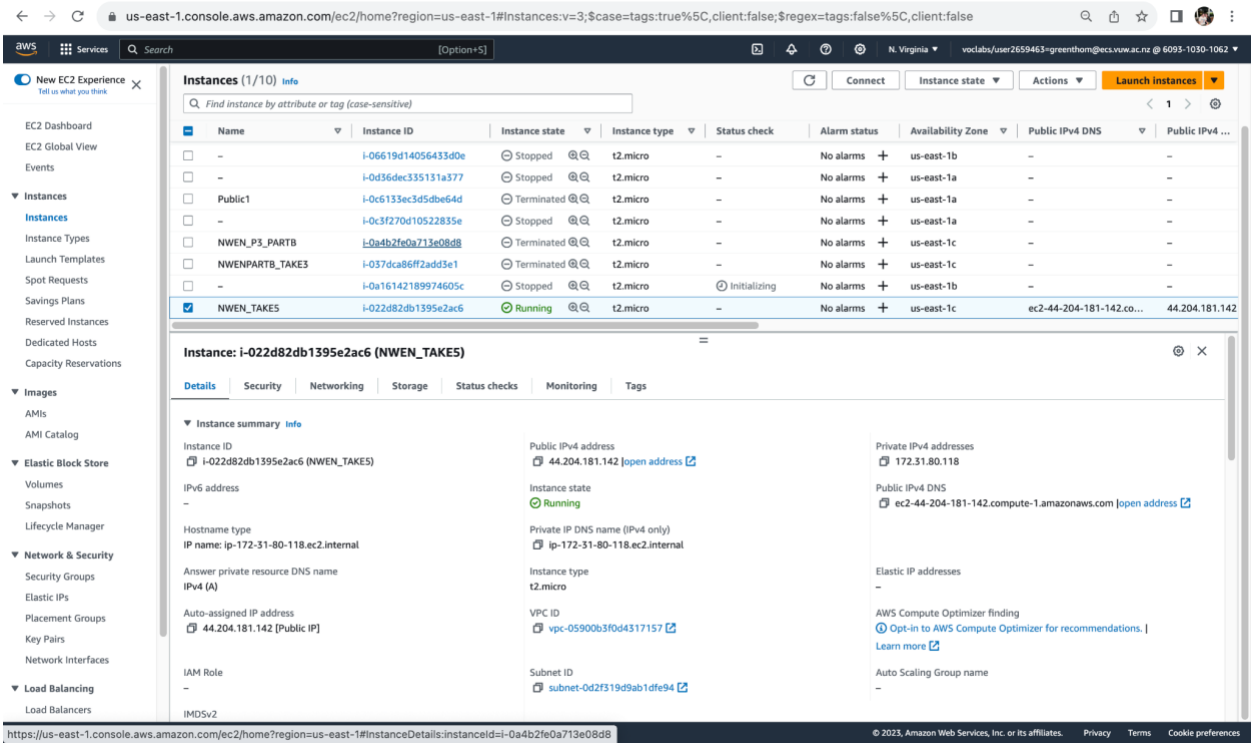
As we can see, this is the target group we set up for our instances. We can see that we have no unhealthy hosts based on our HealthCheck server. We can also know since we upped our autoscaling group's minimum value, we can see that we have four healthy running instances. Since we also upped our autoscaling group's average value, we can also see the counter of that increase showing we have healthy hosts. We can also see we have no unhealthy hosts (average). Load balancer metrics in CloudWatch show the request count and healthy host count, the load on the instances, and how it correlates with scaling actions.



Our CloudWatch metrics for the autoscaling group show the total number of instances. We can see how quickly the instances are added or terminated. The new flow count shows where I have been terminating and autoscaling, making more instances. We can see that the healthy hosts drop briefly before the AWS makes up a new instance. You can see little dips making and removing instances.

PART B

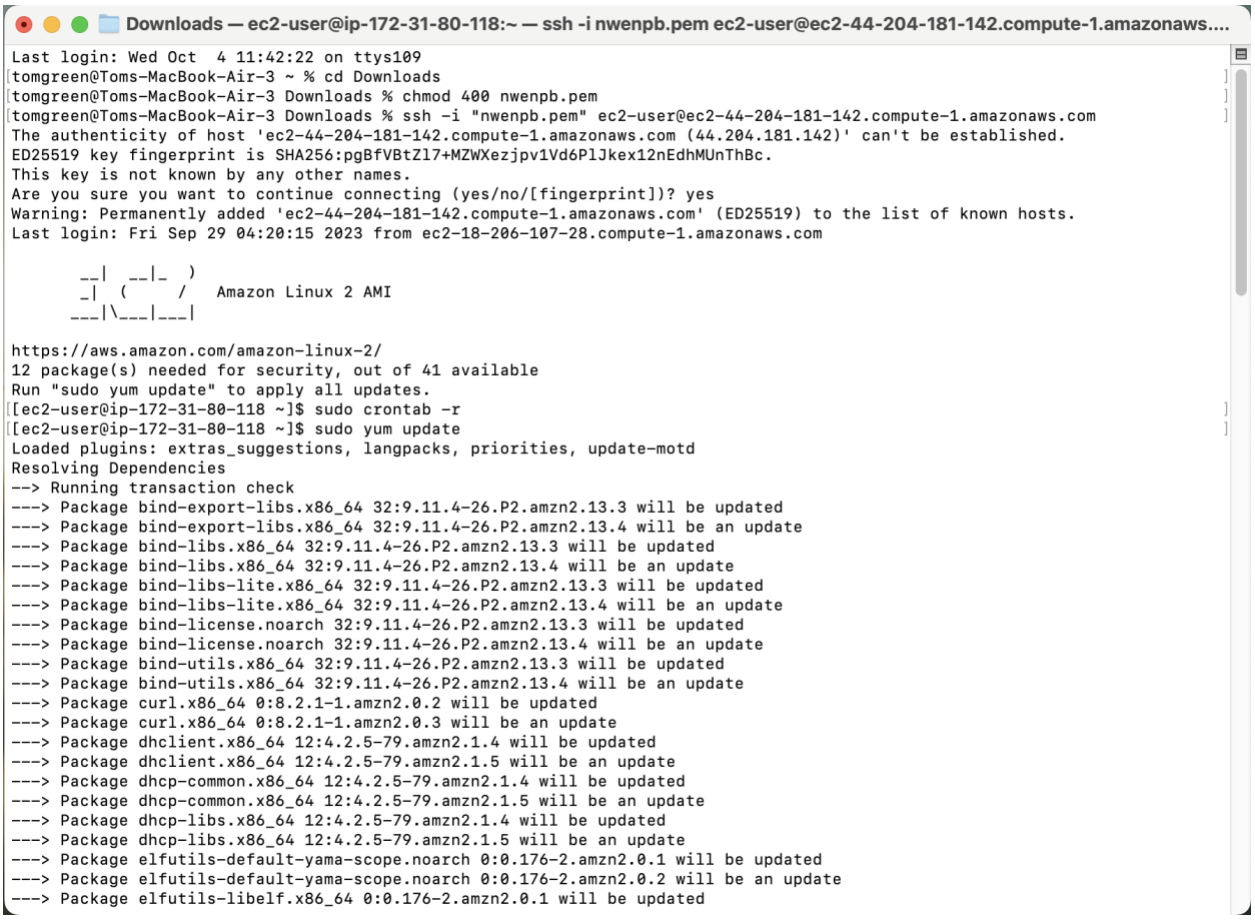
Here, we create an instance from our Project3AMI that was made at the start of project 3a)



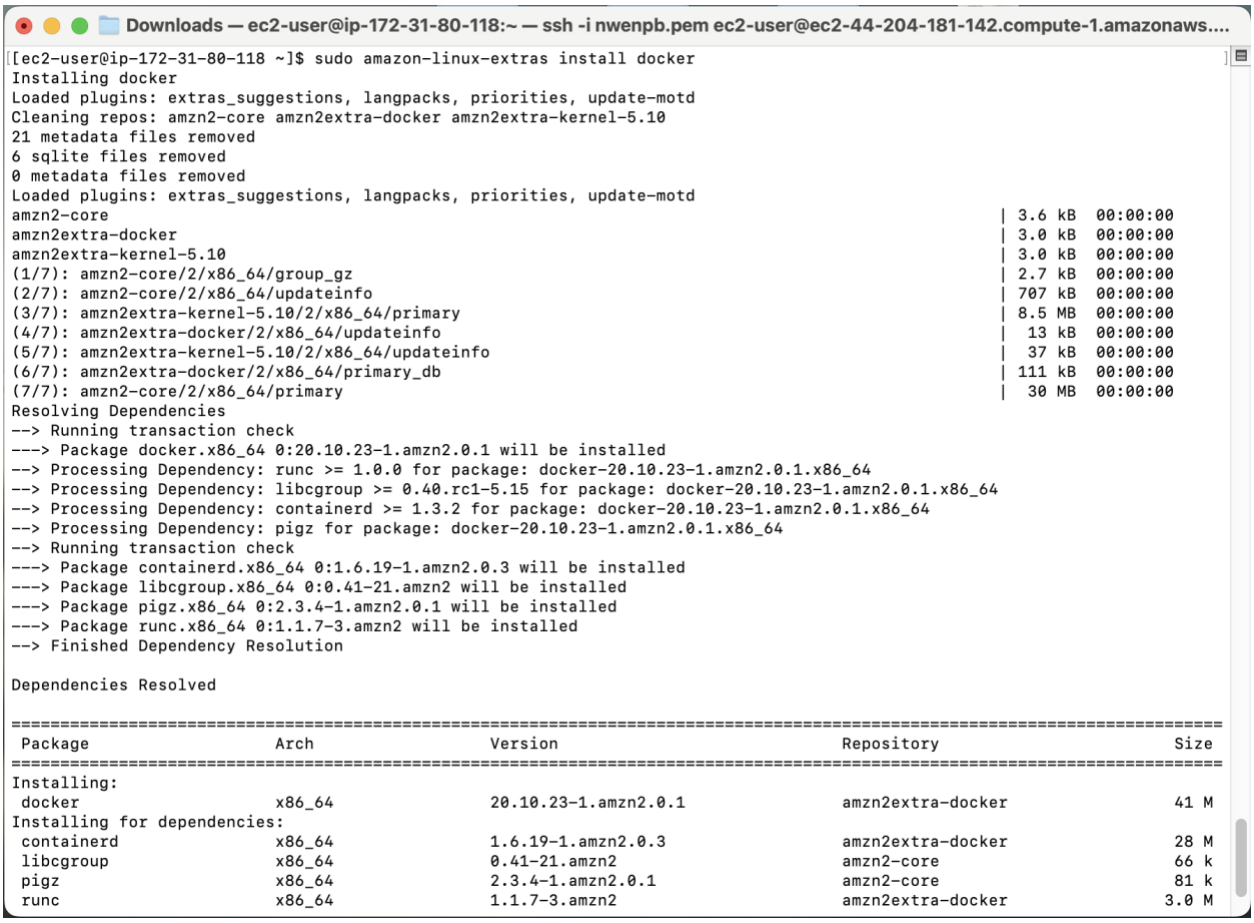
From here, we ssh into our instance

We must remove the crontab created during project two so the port is not being used.

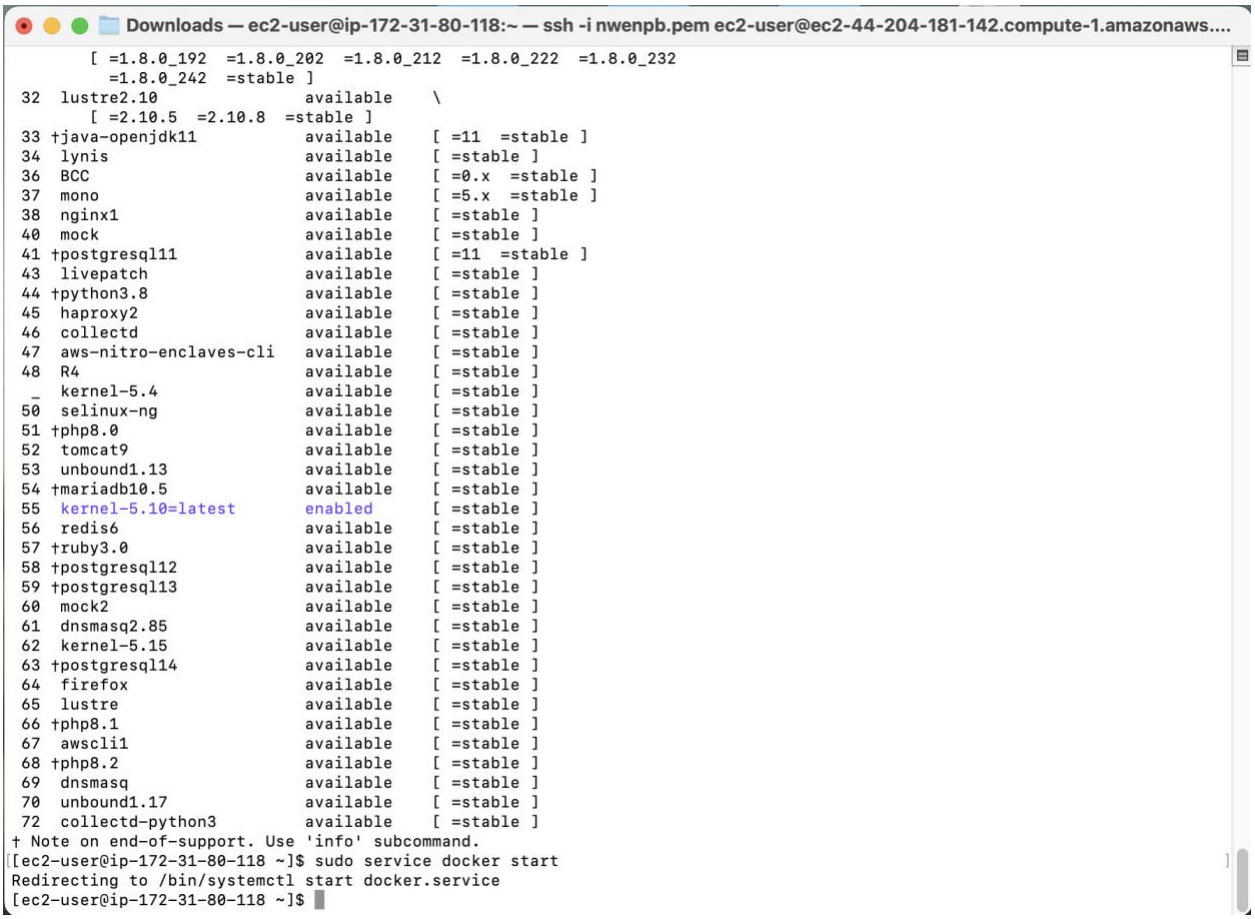
Next, we run sudo yum update to run a general update on the instance



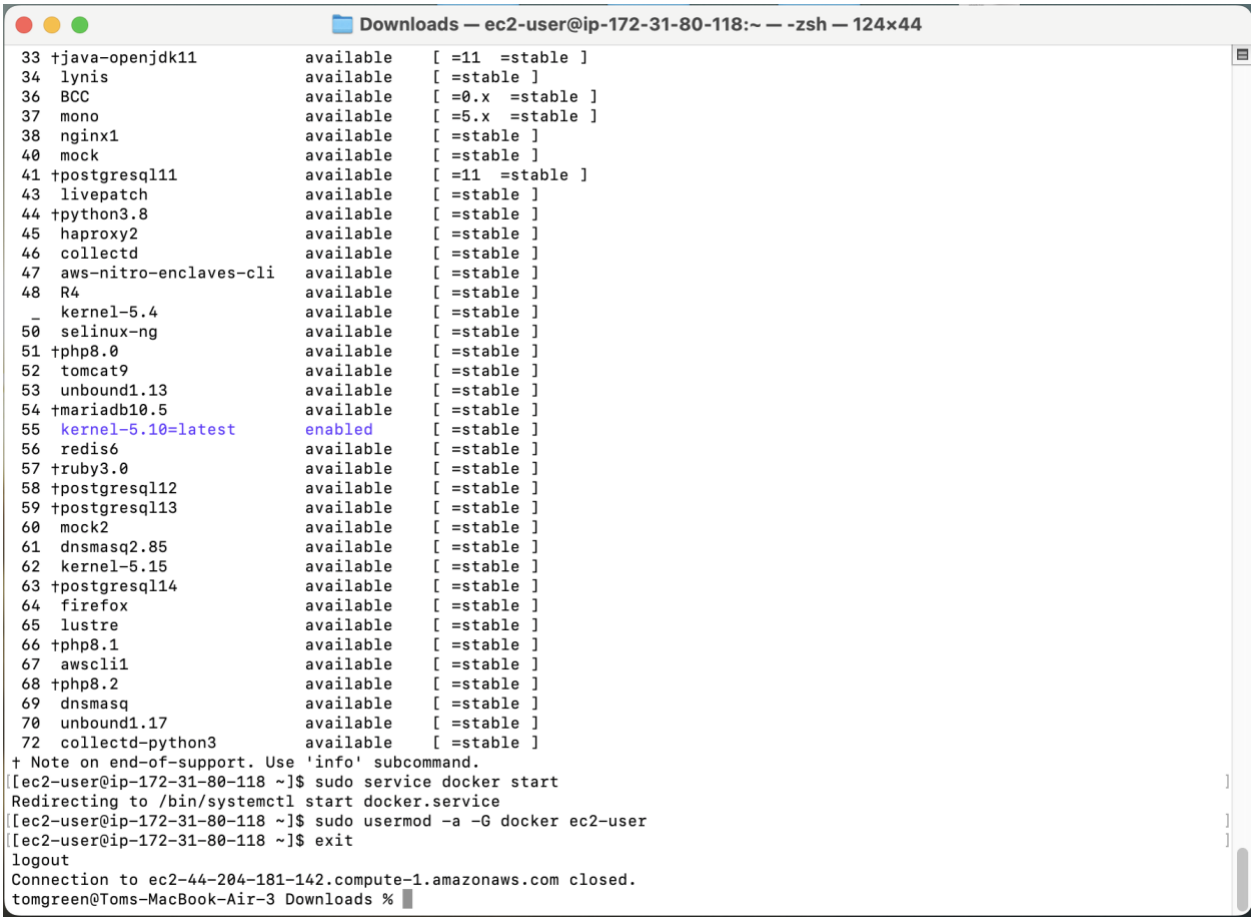
Next, we run ‘sudo amazon-linux-extras install docker’



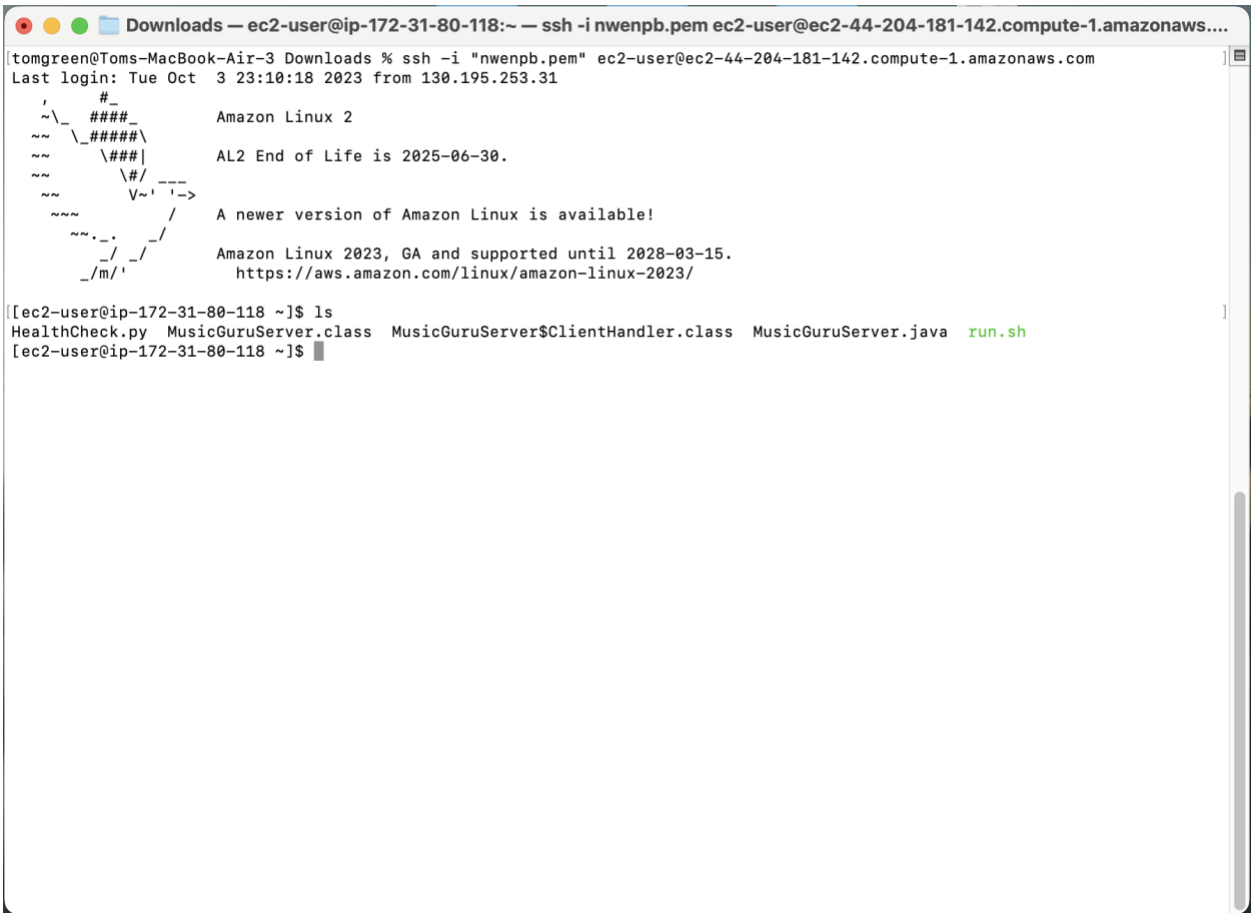
Next, we start the docker by running sudo service docker start



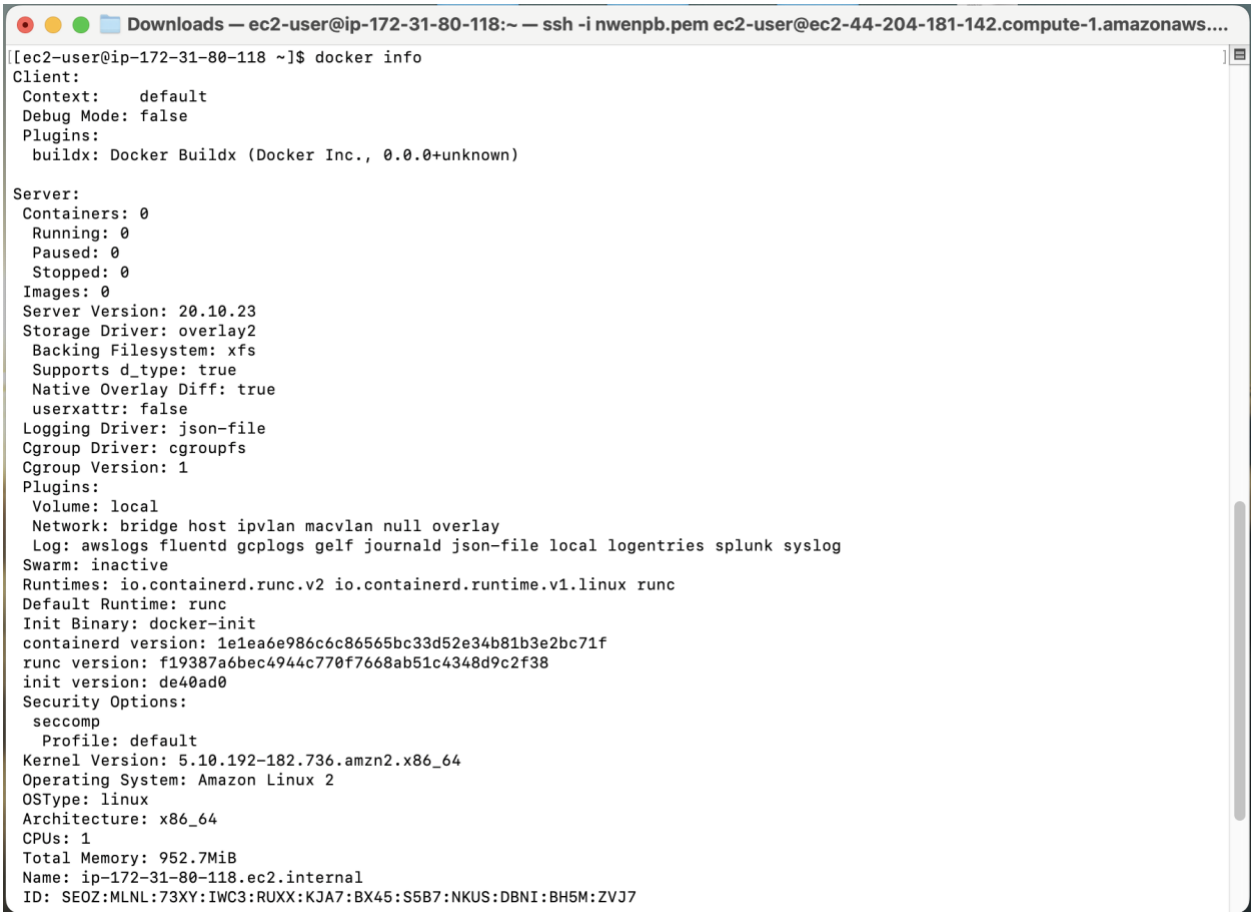
Next, we run ‘sudo usermod -a -G docker ec2-user’ to add the default user. From here, we then exit so it refreshes the permissions we just set



We then ssh back into the instance

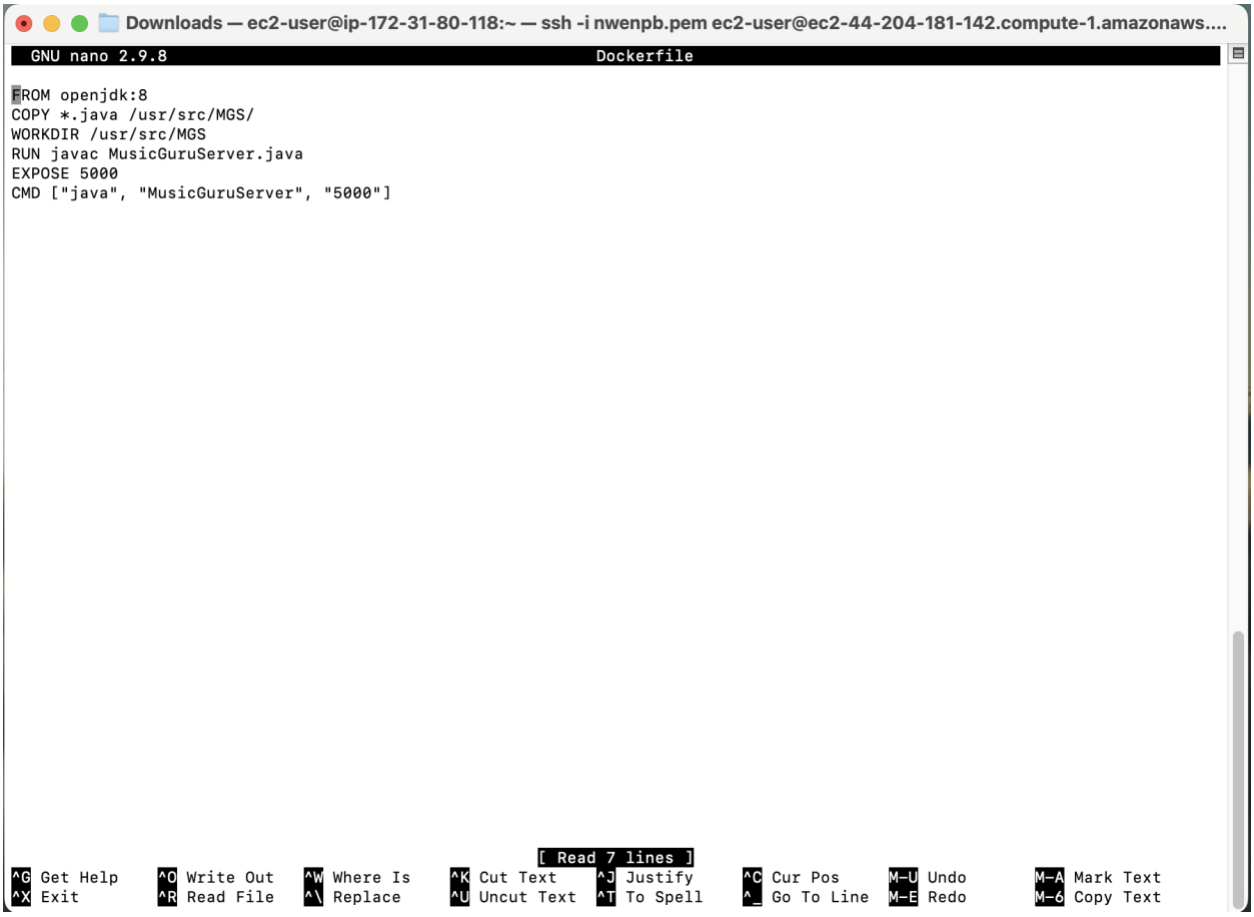


From here, we use the command docker info, where we can see a status dump where we can see that no containers are hosted

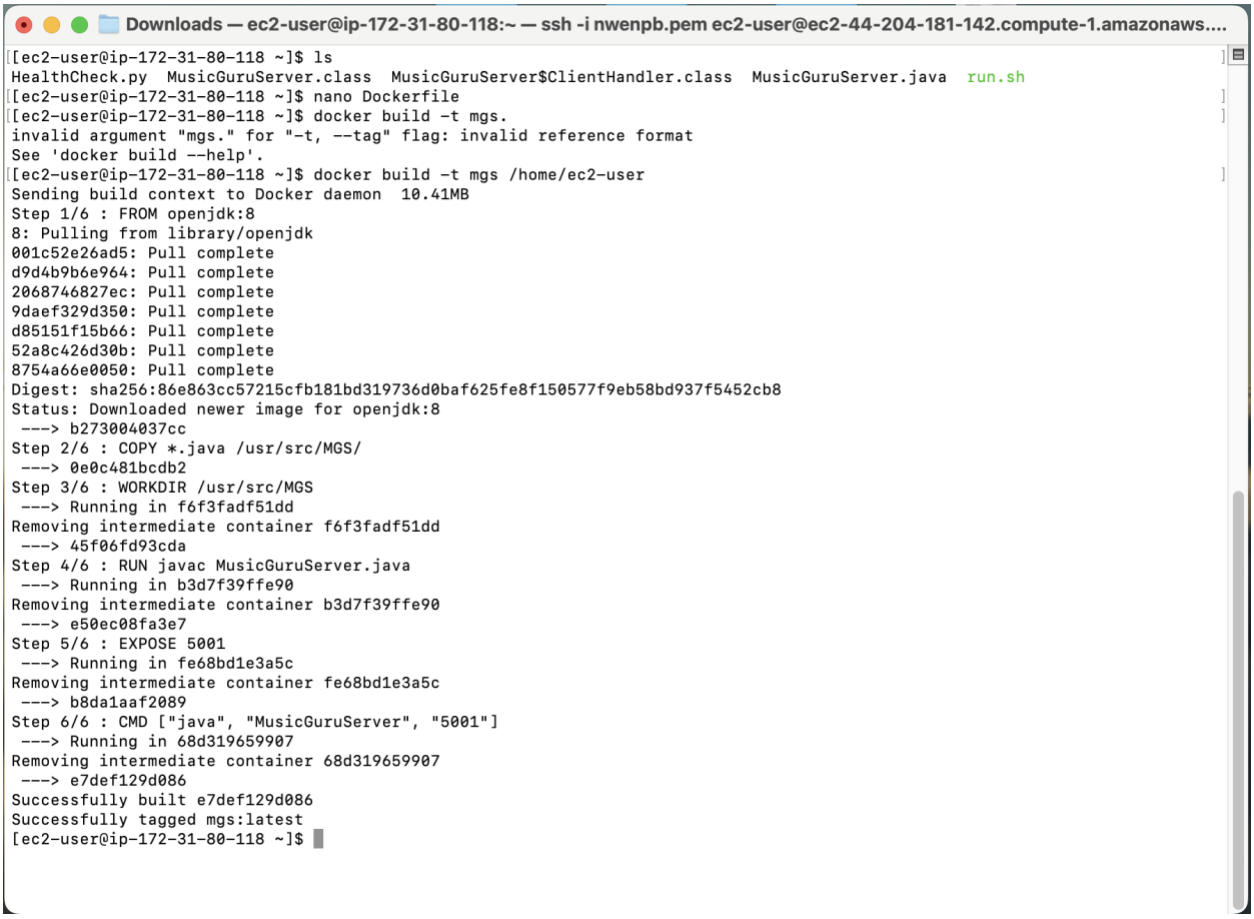


Now, what we are doing is creating a 'Dockerfile.'

From here, we use the Java Dockerfile. Since we did not use a text/JSON file to hold our songs, we do not need to add the second COPY line.

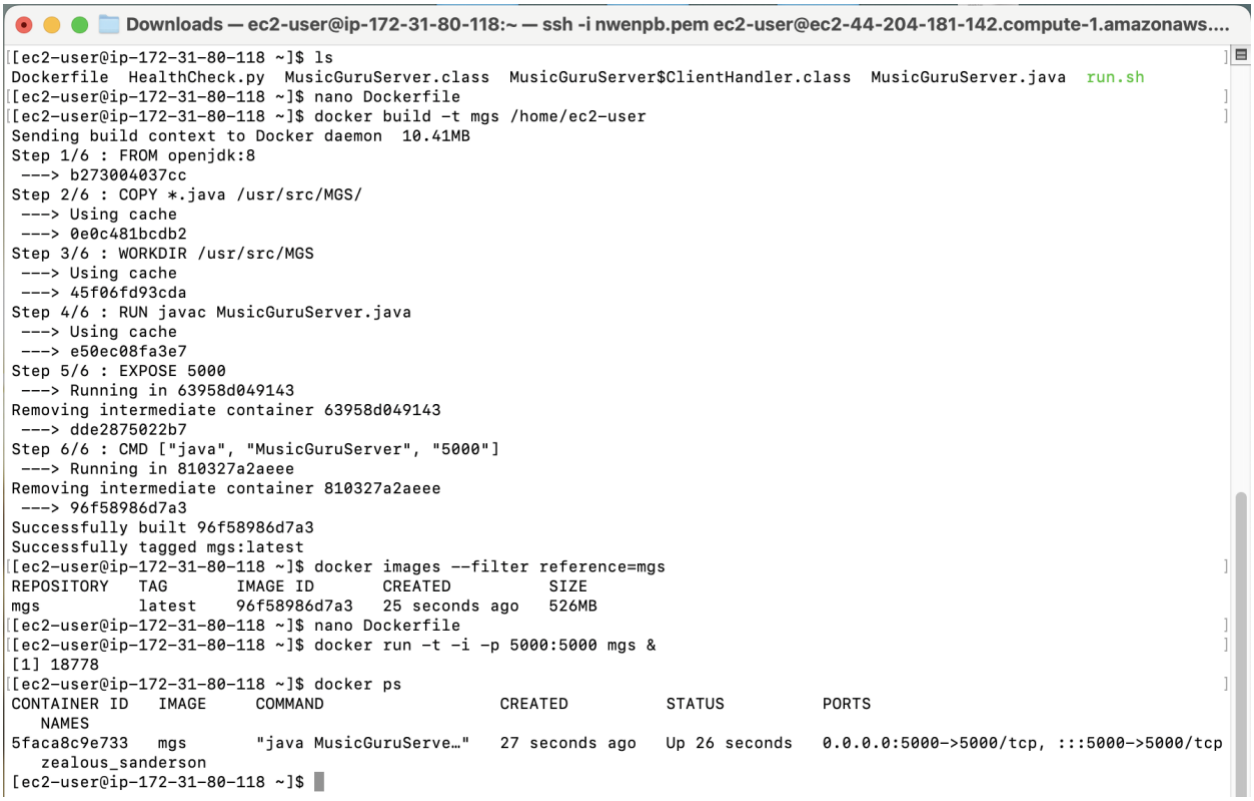


From here, we build the docker using the command 'docker build -t mgs /home/ec2-user'. As we can see, the build was successful

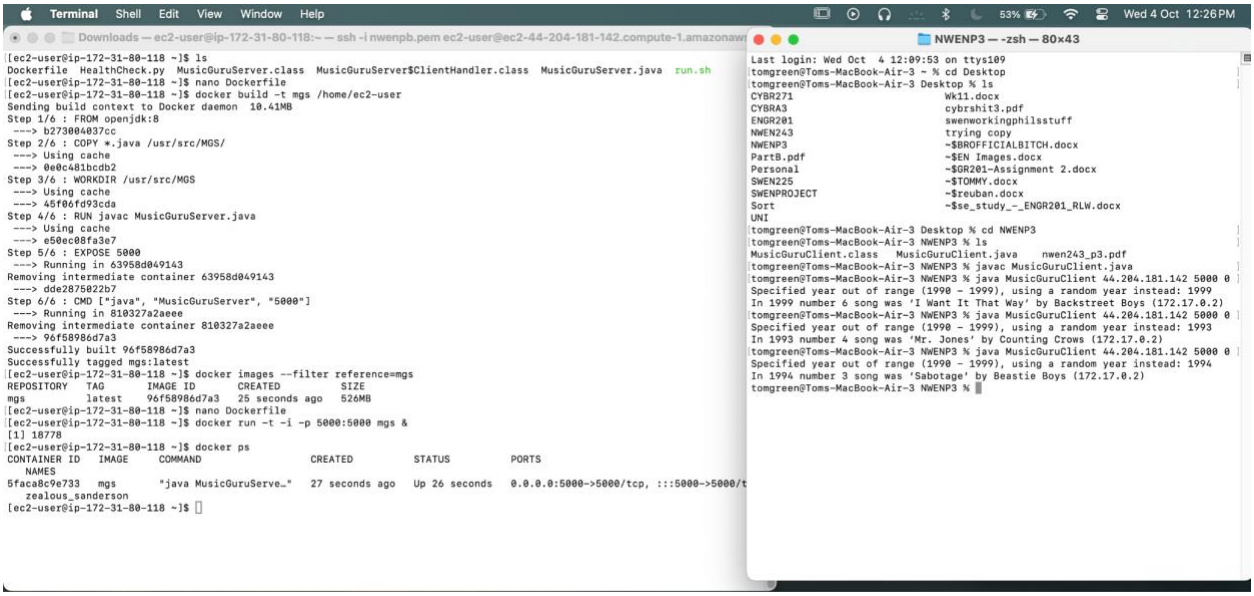


From here, we can check if the docker builds the image properly using the command 'docker images –filter reference=mgs.' As we can see, it made our mgs successfully. Next, we can finally run the built image. We exposed the dockerfile at port 5000. We need to map the exposed port on the container to a port on the host machine. We do that by running the command 'docker run -t -i -p 5000:5000 mgs &'. This gives us a response that says we have job one running in the background, and its PID is '18778'.

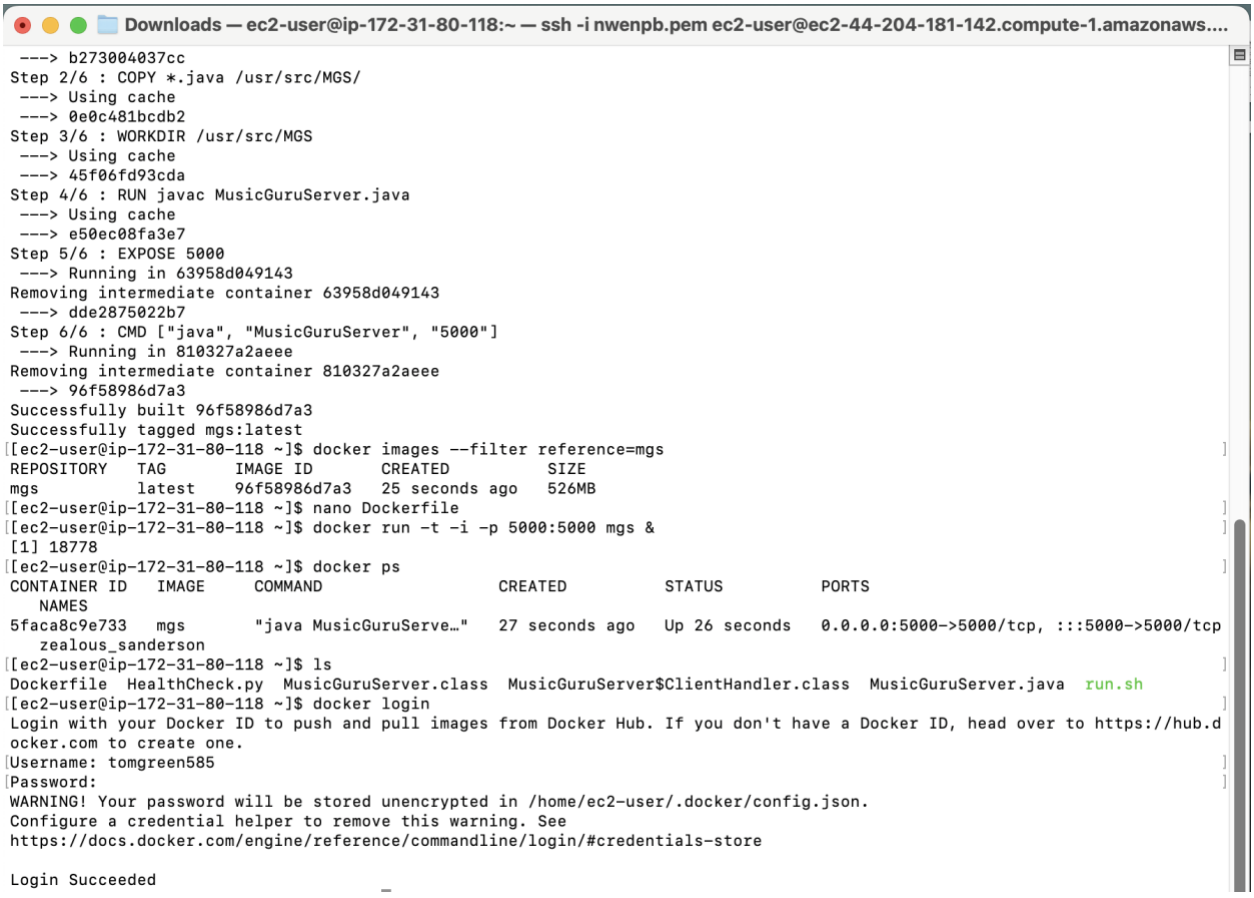
From here, we run the command docker ps to see if everything is running. We can see the port mapping, the command image, and the container ID showing us that this was successful.



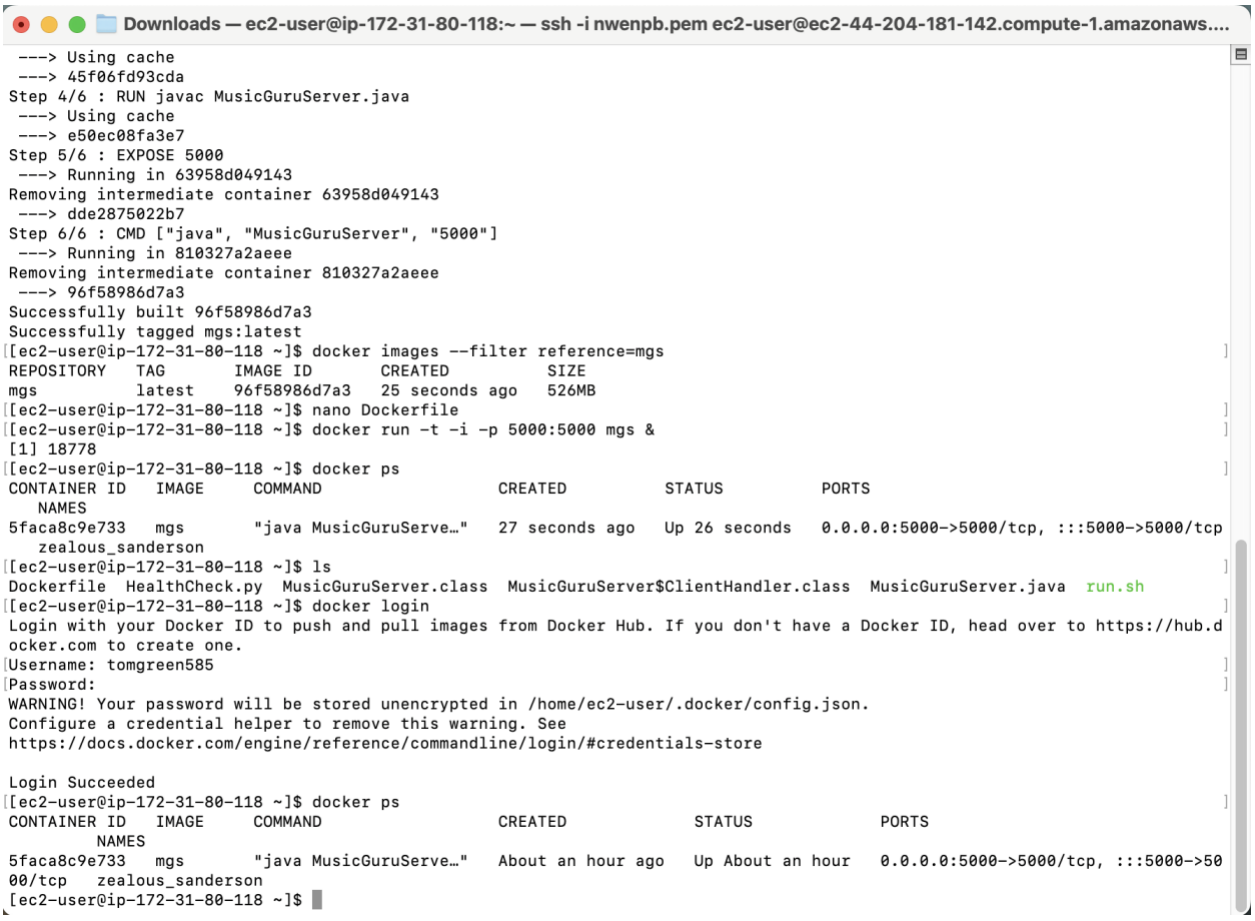
From here, we run the original client from my local machine using the public IP of the instance, the port number that the docker is running on, and the year. As we can see, this was successful.



The next thing we do is create a hub.docker.com account. I was signed up with the username 'tomgreen585'. I made a repository under 'mgs' to push and pull my docker from the AWS instance. From here, I signed into my docker account on my instance by running the command' docker login'. As we can see, it was successful.

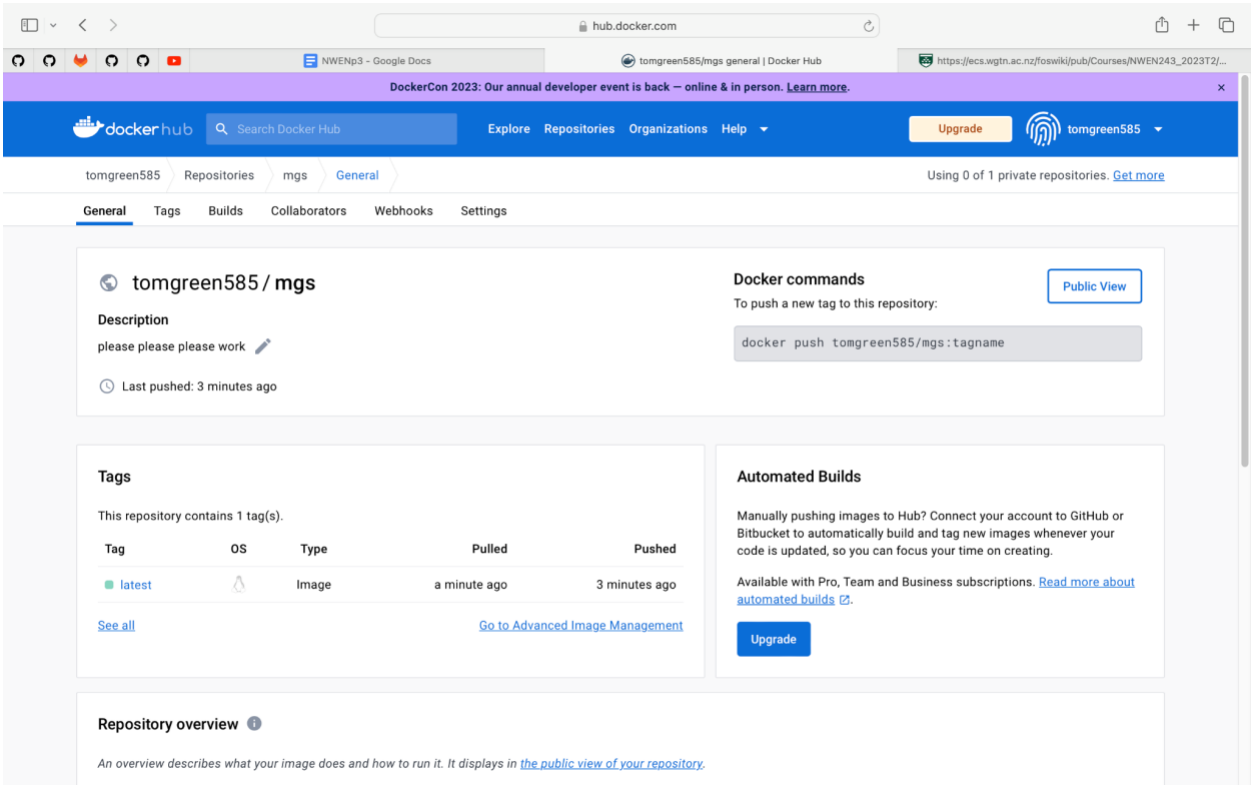


They checked that the server was still running using the command 'docked ps.'

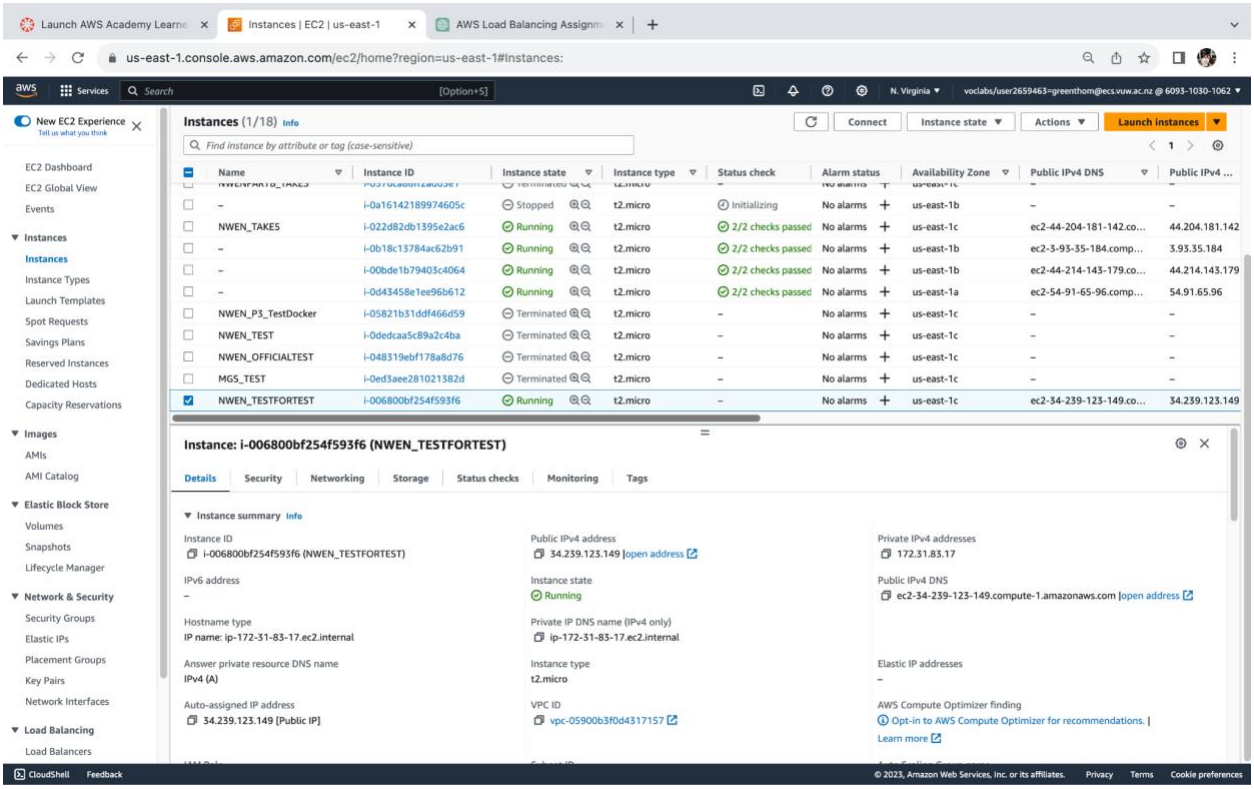


From here, I tagged my docker by using the command' docker tag mgs tomgreen585/mgs:latest'. 'mgs' is my docker name, and I am tagging it with my username/reponame:latest, which gets the which as you can see above is the tag given to my docker image. From here, we push the idea to my repository in DockerHub using the command 'docker push tomgreen585/mgs:latest'. From here, we can see that it pushed my mgs docker successfully. ‘docker’ is the tag used to create a new tag for an existing image. ‘mgs’ is the name of the existing docker image. ‘tomgreen585/mgs’ is the repository I created and ‘latest’ is the tag of my mgs repository.

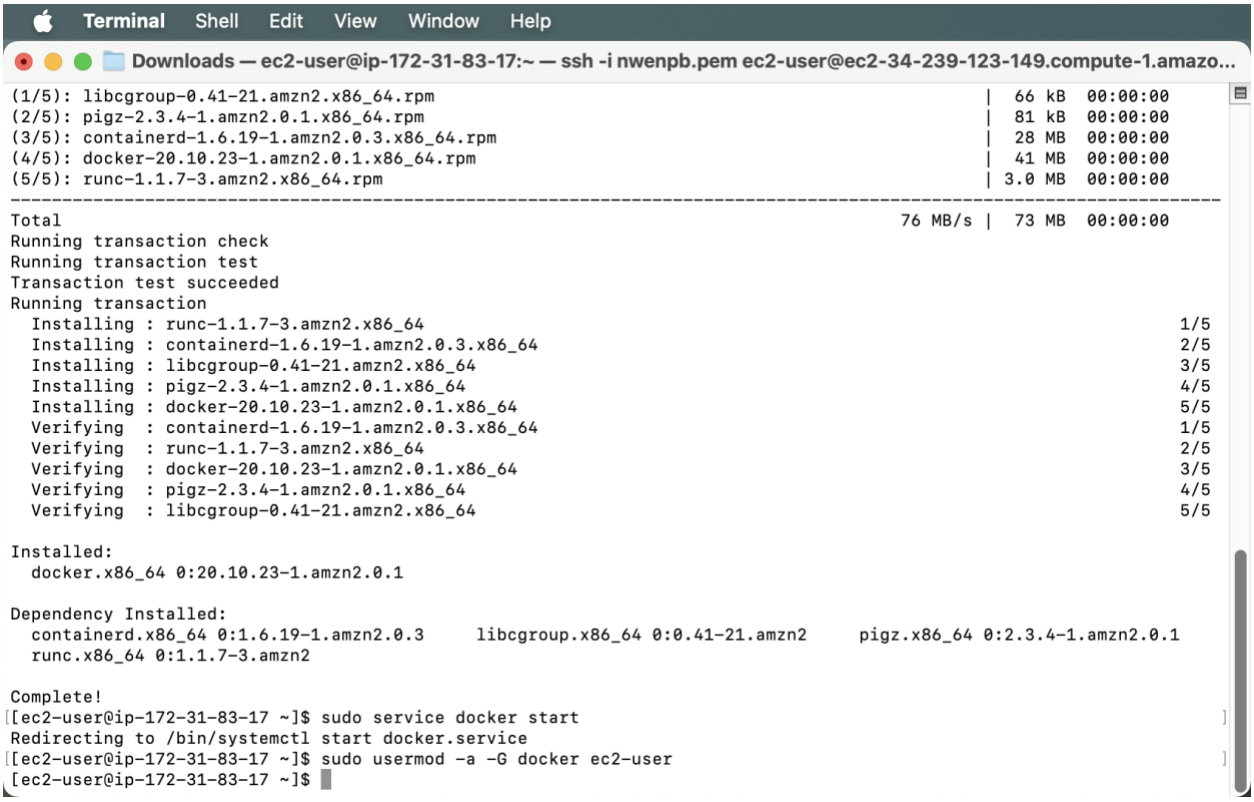
Here, we can see that our docker was pushed successfully to my repository on DockerHub



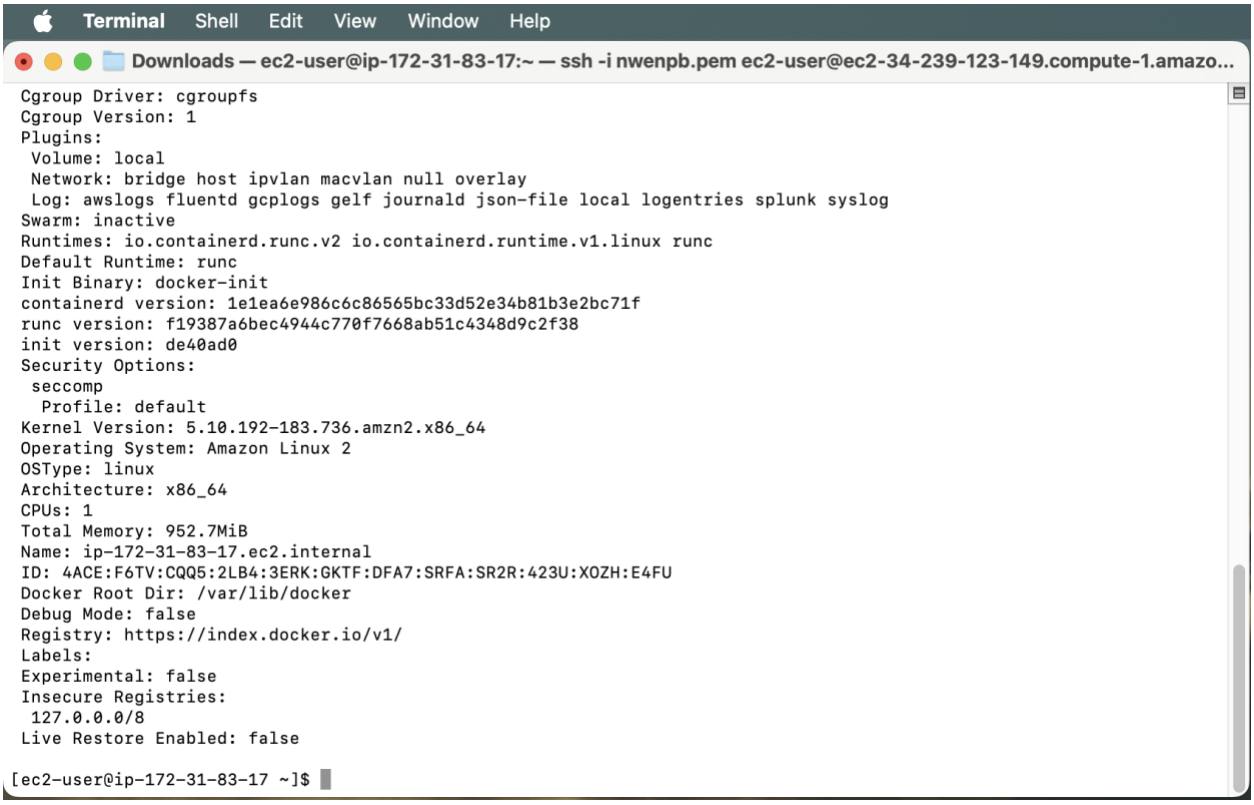
Next, we made a new EC2 instance called 'NWEN_TESTFORTEST' (sorry about the other tests; I just wasn't setting it up correctly) to check that the pushed image in DockerHub was correct.



Use commands from the handout 2a.b.c.d



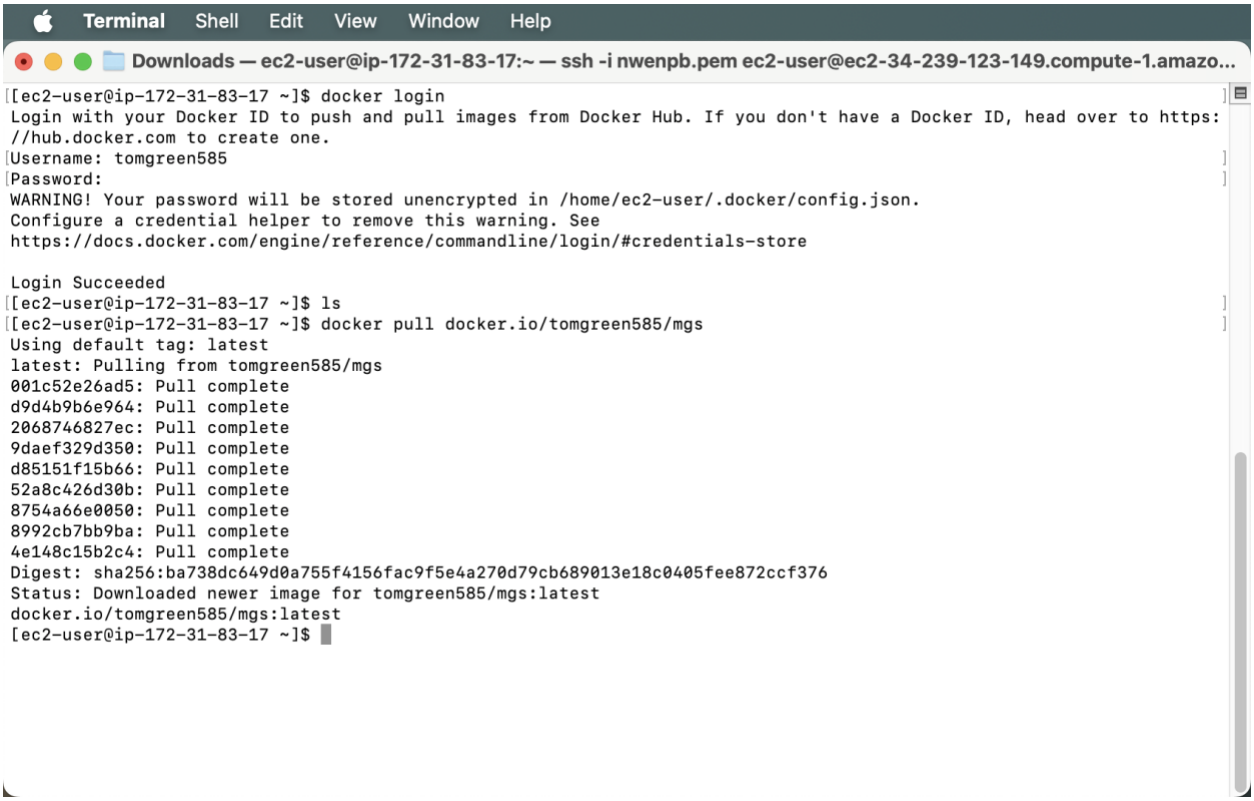
We exit the ssh and return to the instance to refresh the permissions we set. We then run the command' docker info' to see a status dump and to check that this was successful.



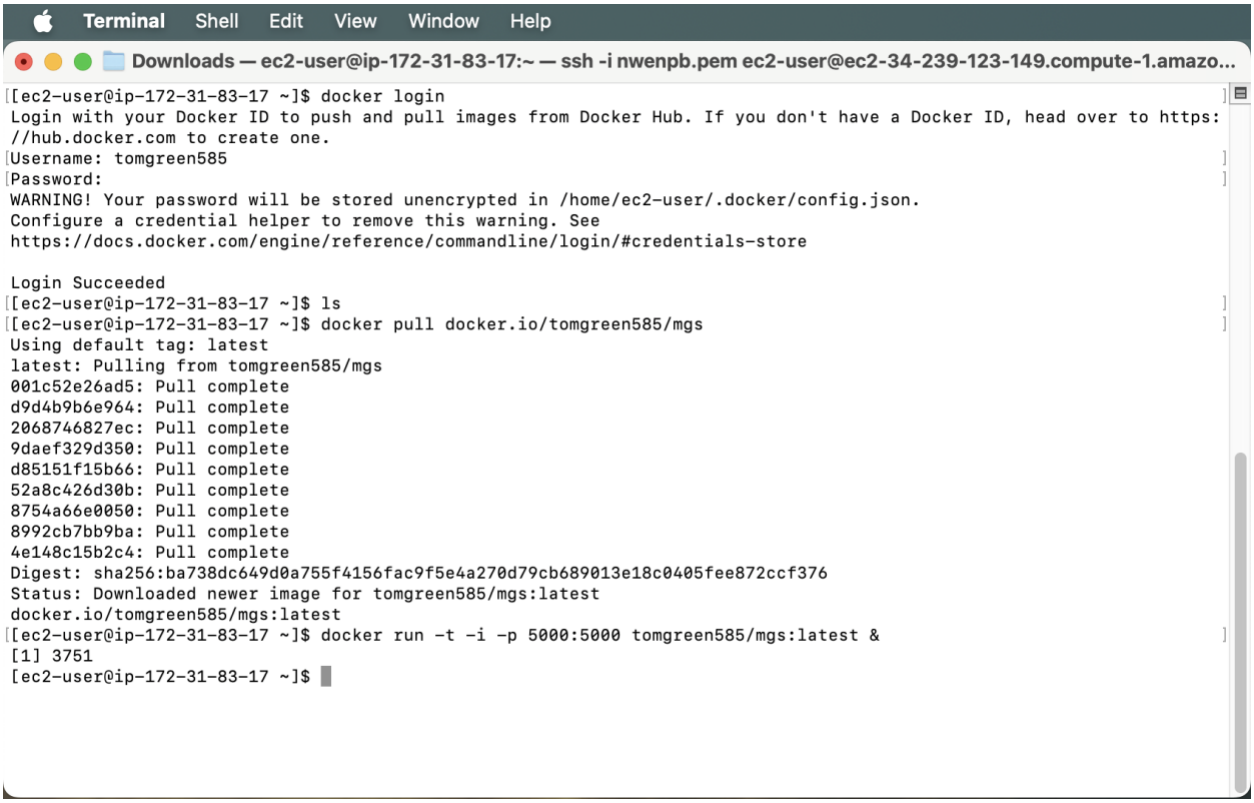
Next, we sign into our DockerHub account by running the command docker login' so we can pull our docker image from our repository

Here, we run 'ls' to show that our repository has nothing in it

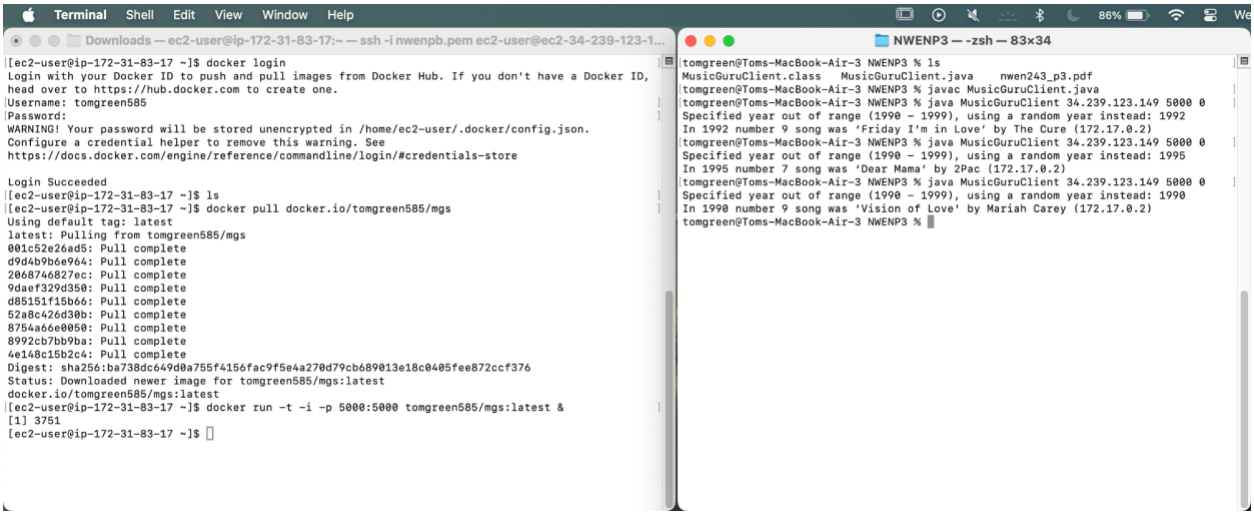
Next, we run the command 'docker pull docker.io/tomgreen585/mgs', which pulls the repository docker I pushed earlier using my username and repository name. As we can see, it was successful, and downloaded a newer image, 'latest,' which was the tag we gave our mgs docker. ‘docker pull’ is used to fetch a docker image from a docker repository. ‘docker.io/tomgreen585/mgs:latest’ is the complete name and tag of the docker image file that we are pulling it specifies that we want to obtain the ‘latest’ docker from the repository tomgreen585/mgs. This allows it to be obtained.



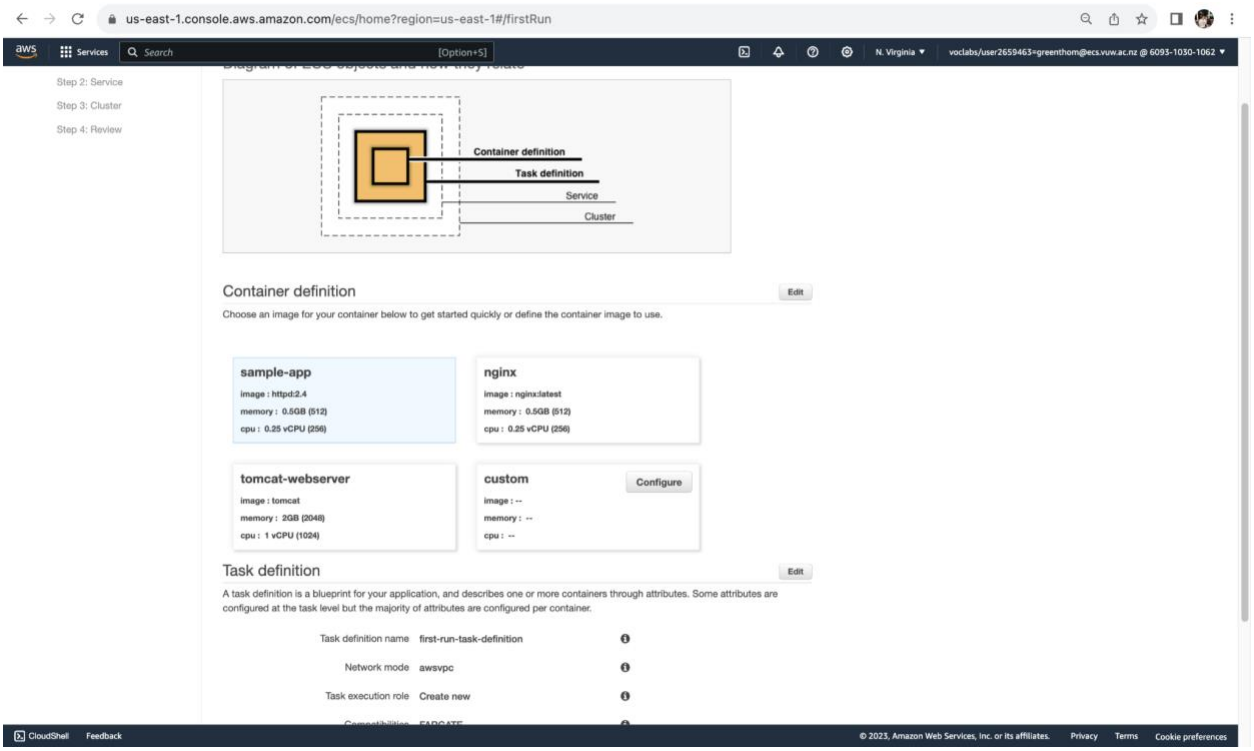
We then run the docker by setting it to a port map using the command 'docker run -t -i -p 5000:5000 tomgreen585/mgs:latest &'. I'm mapping port 5000 on my EC2 instance to port 5000 within the Docker container. This mapping is essential because it enables me to access services running inside the container via port 5000 on my EC2 instance. I specify the docker image I want to run 'tomgreen585/mgs:latest &'. The '&' allows the docker container to run in the background.



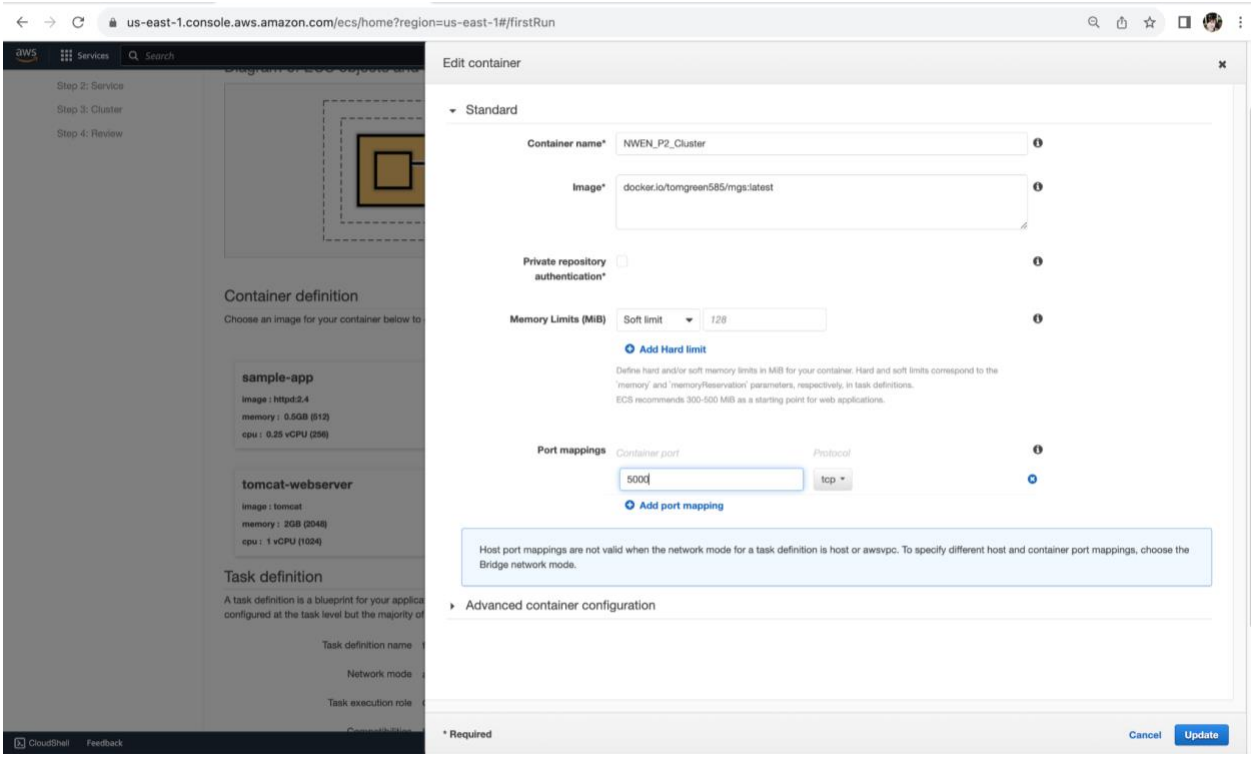
Now we run our MusicGuruClient connecting to our new instance running the pulled docker using the instance IP, the port number we set, the docker image, and the year. As we can see, it was successful.



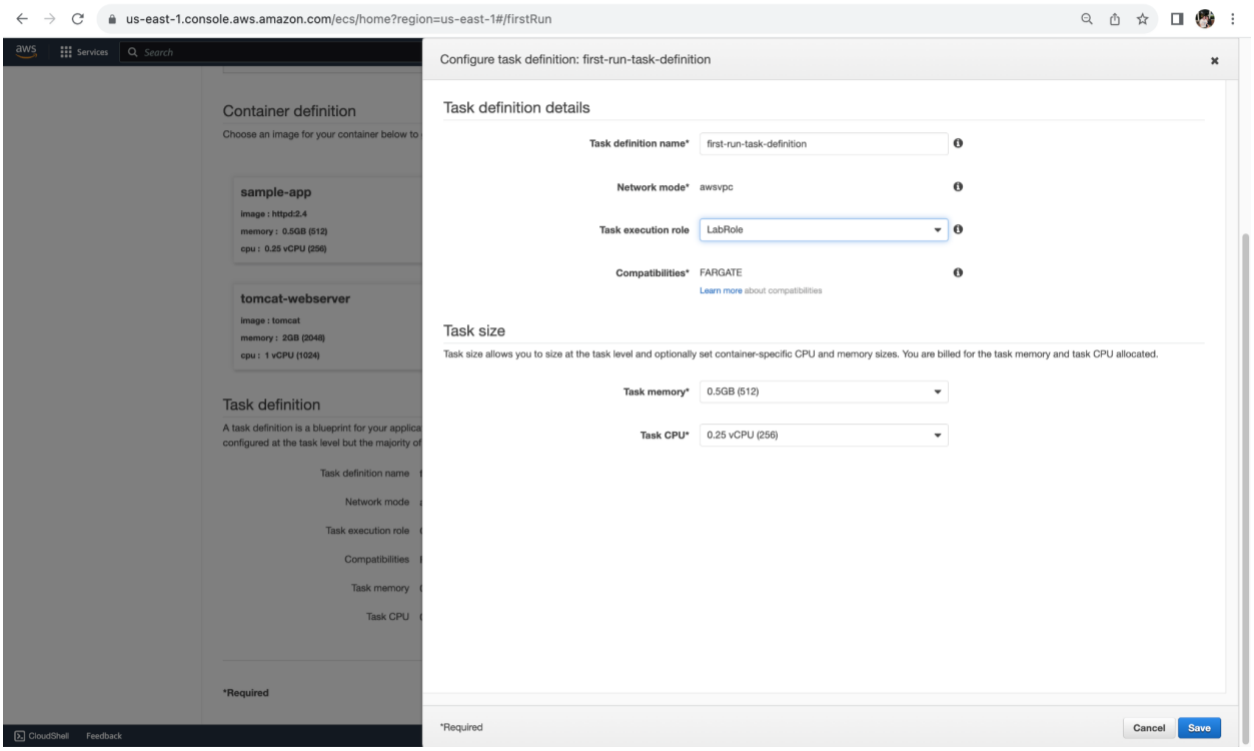
Next, we make an Amazon Elastic Container Service (ECS) using Fargate



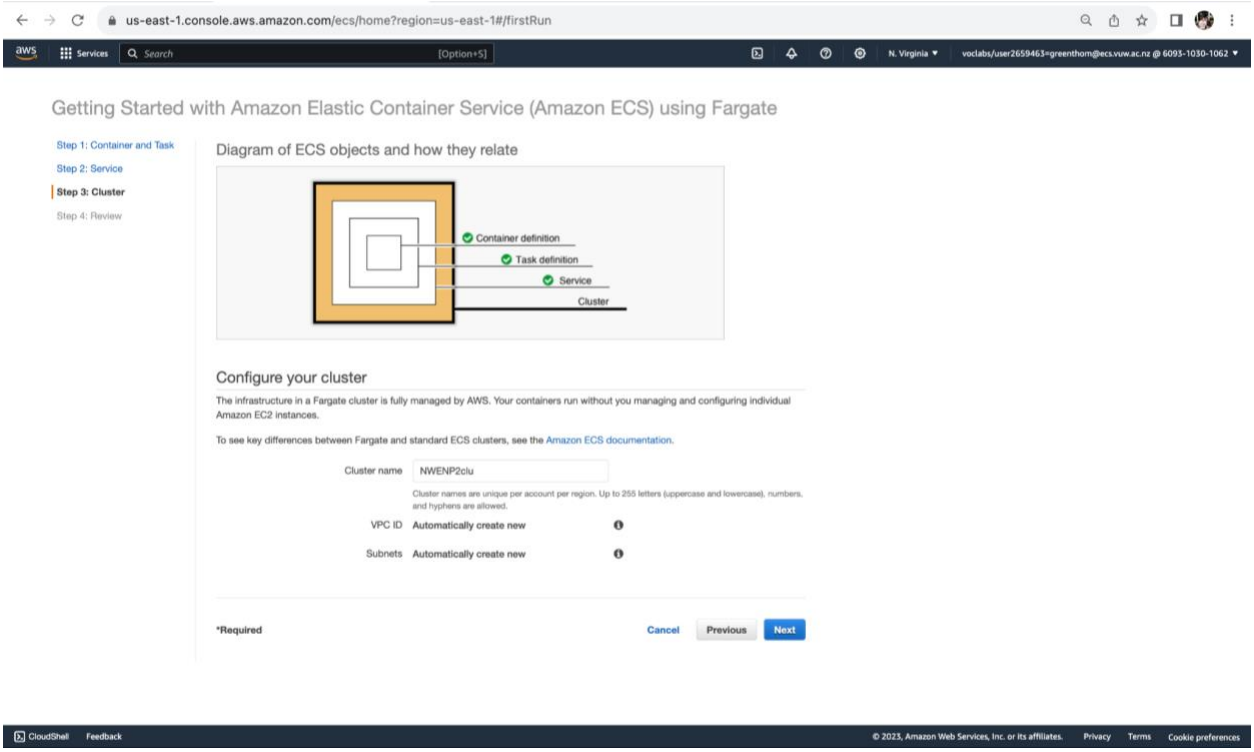
We then edit the container, setting it a name 'NWEN_P2_Cluster', our docker.io repository 'docker.io/tomgreen585/mgs:latest'. Ensured the protocol is TCP and the port set to '5000'.



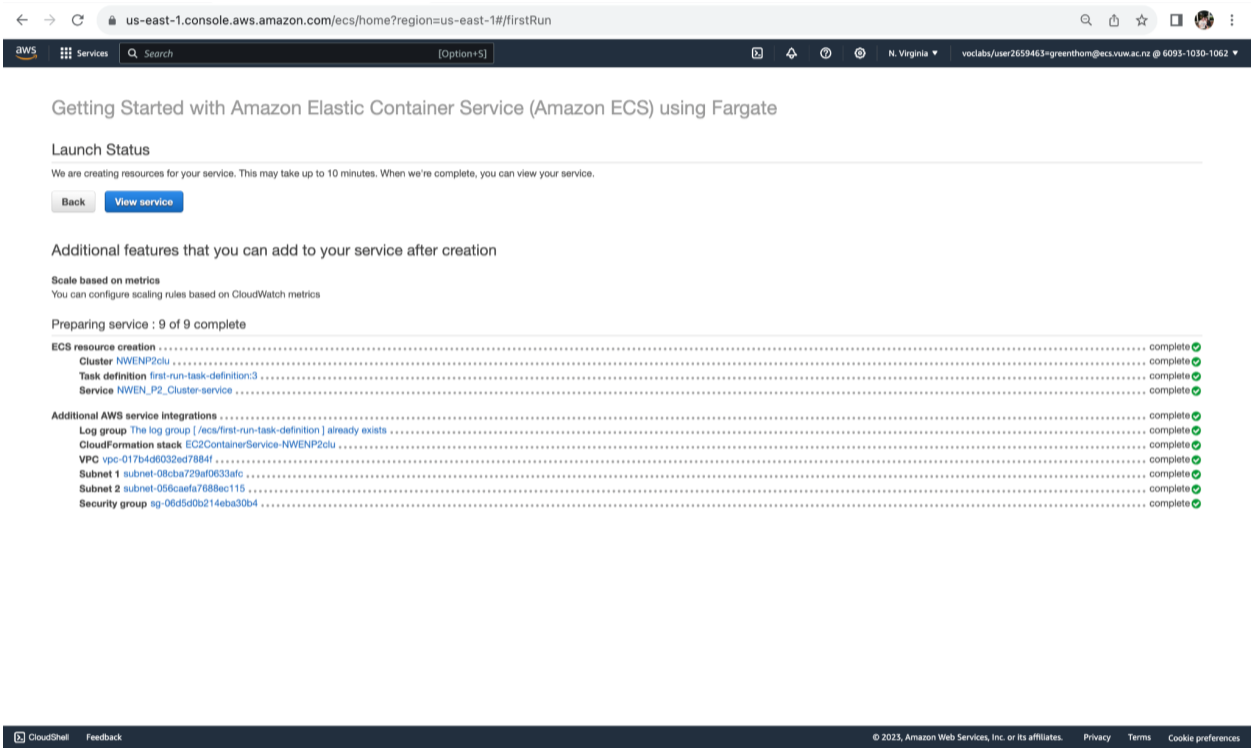
Next, we edit the task definition. We keep the definition name. We changed the task execution role to LabRole.



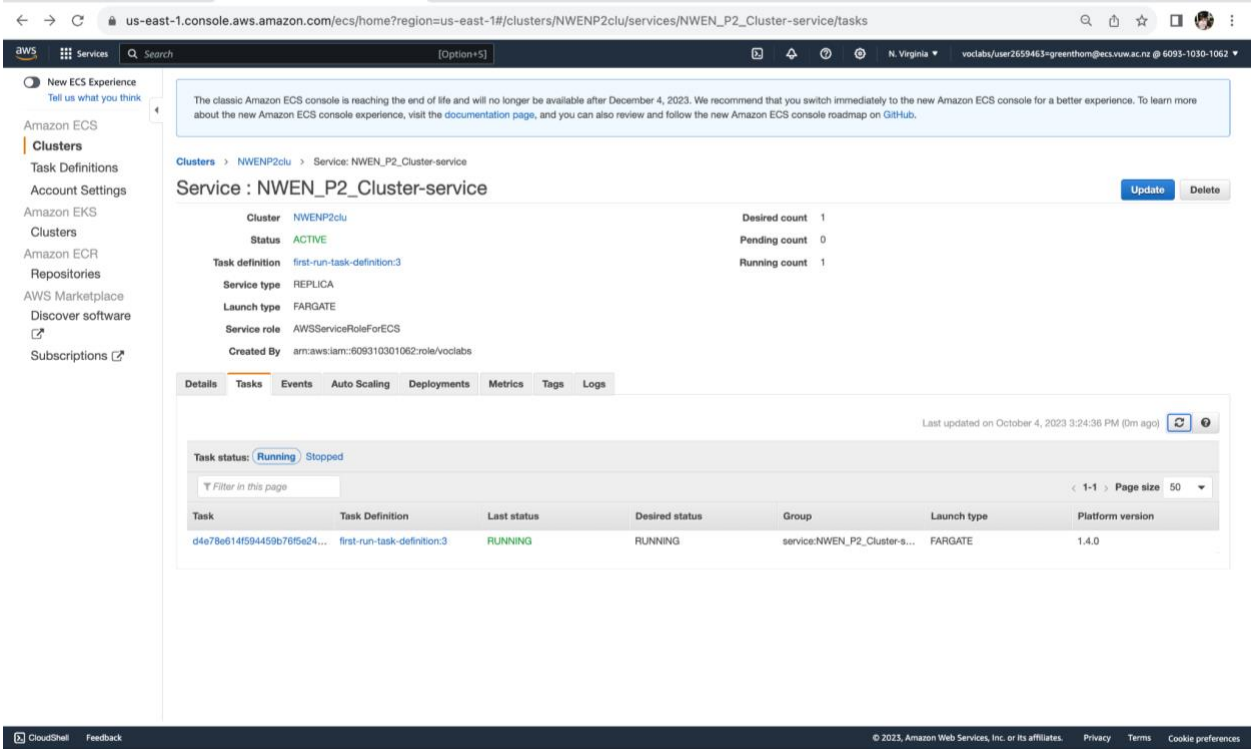
We set the cluster name to NWENP2clu



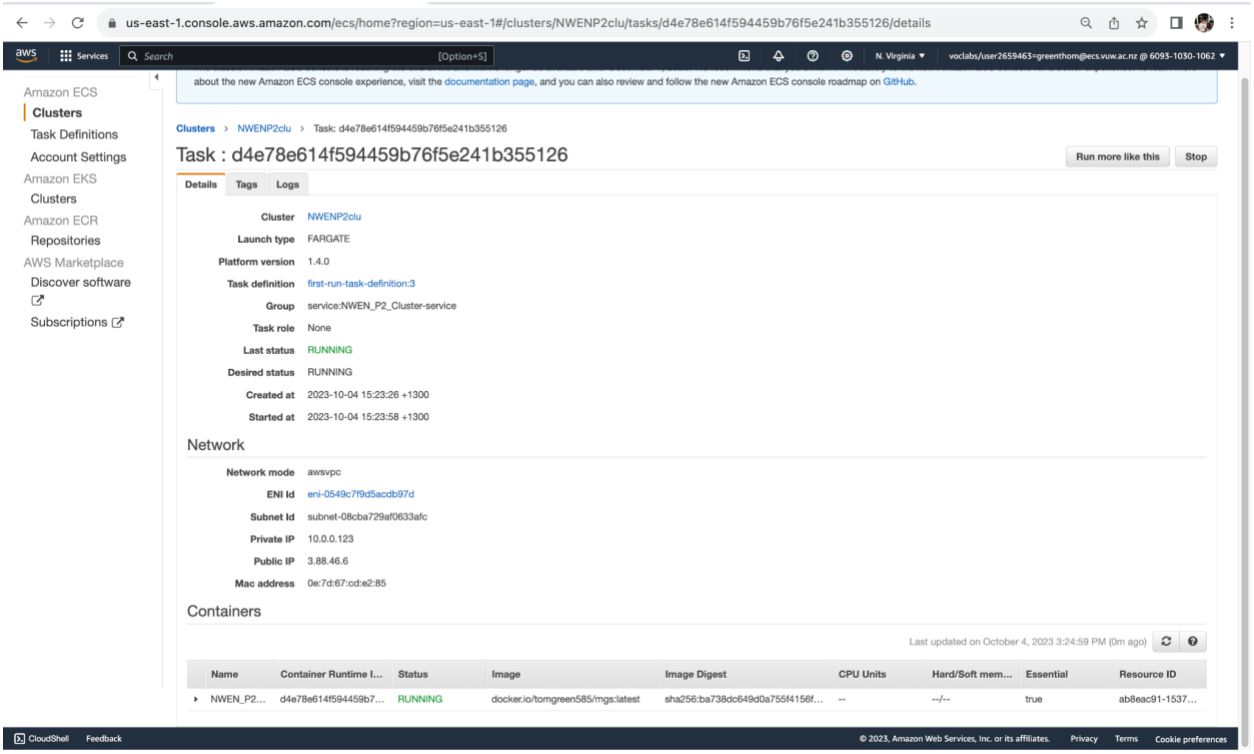
We check the ports and roles are correct and then create it as we can see it completed successfully.



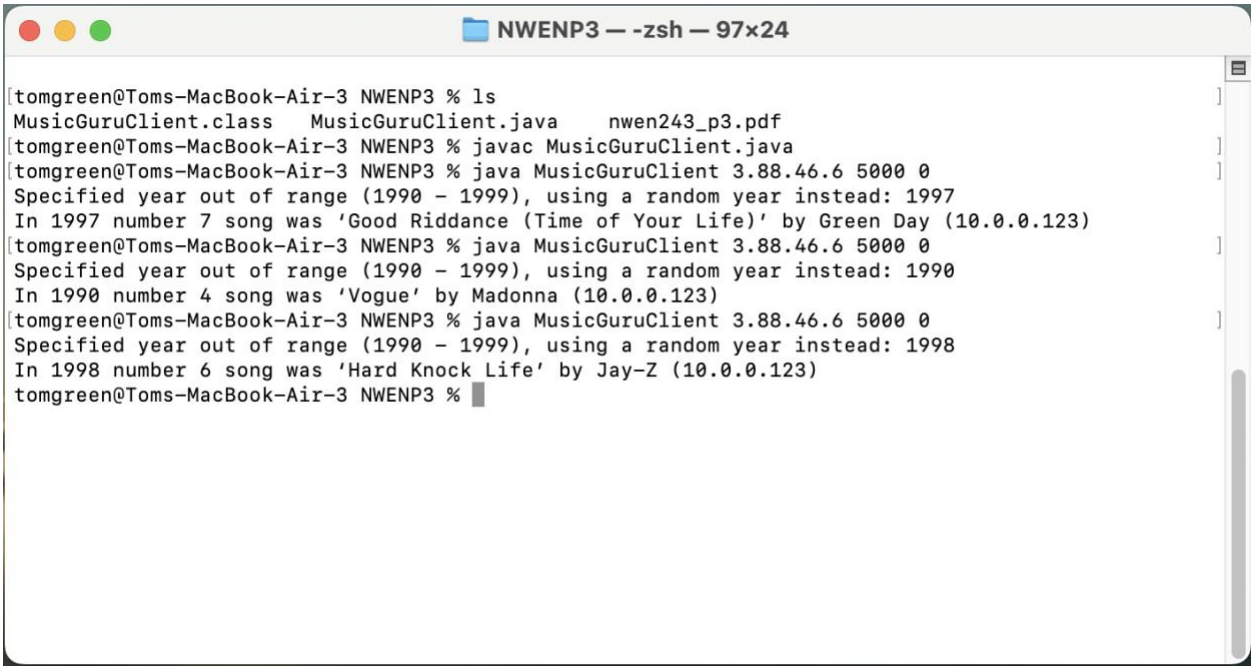
As we can see, it pulled the docker from the repository correctly and is running



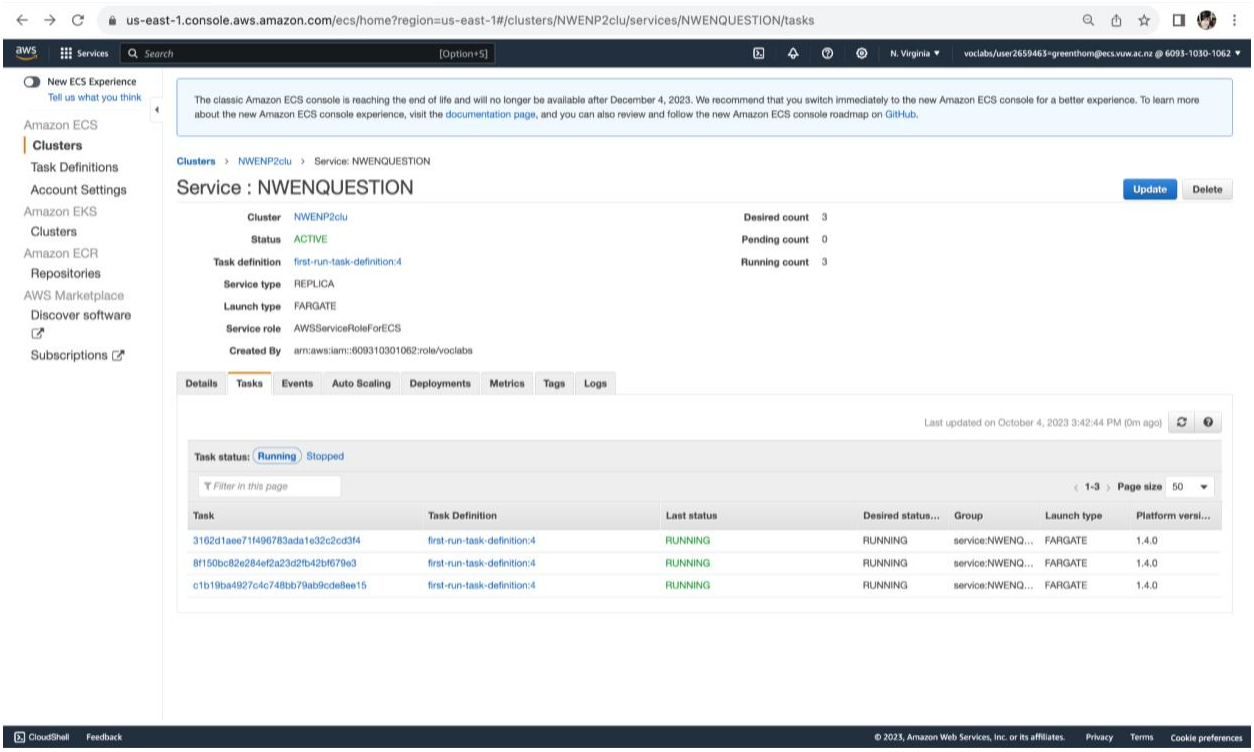
Next, we need to find the IP address for the serverless service. As we can see here for this task, our public IP is 3.88.46.6

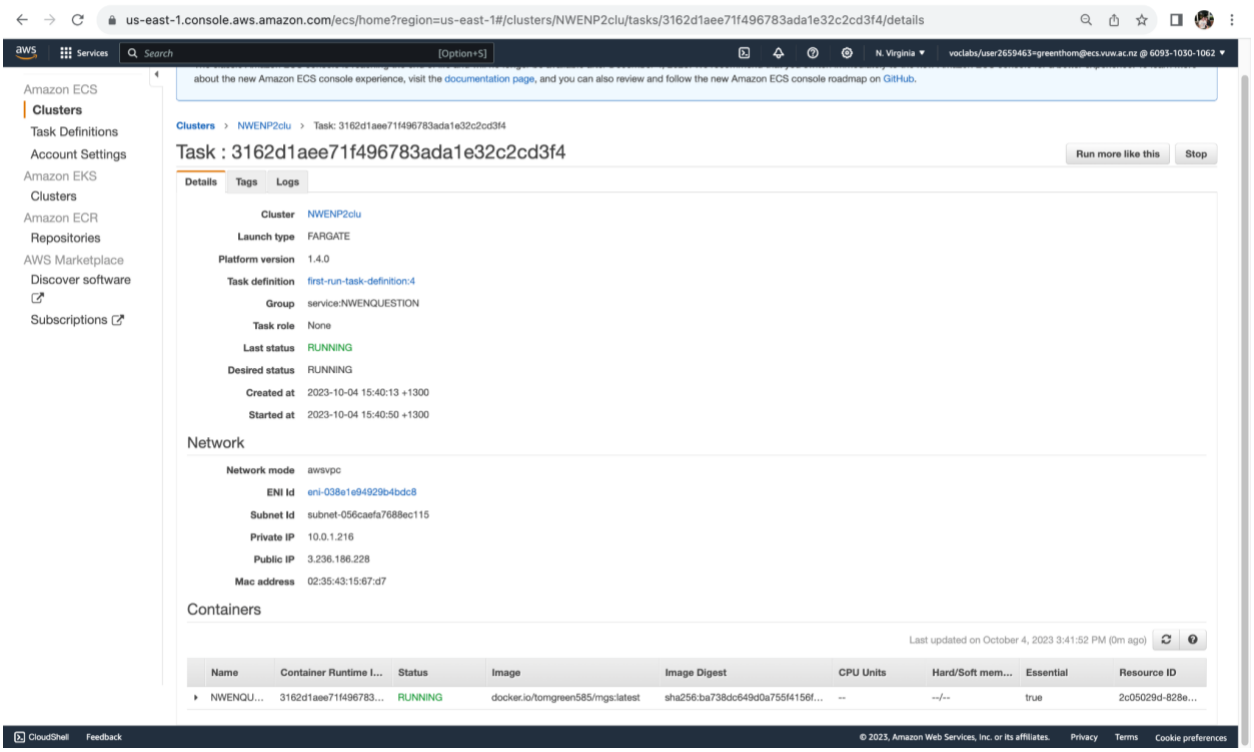


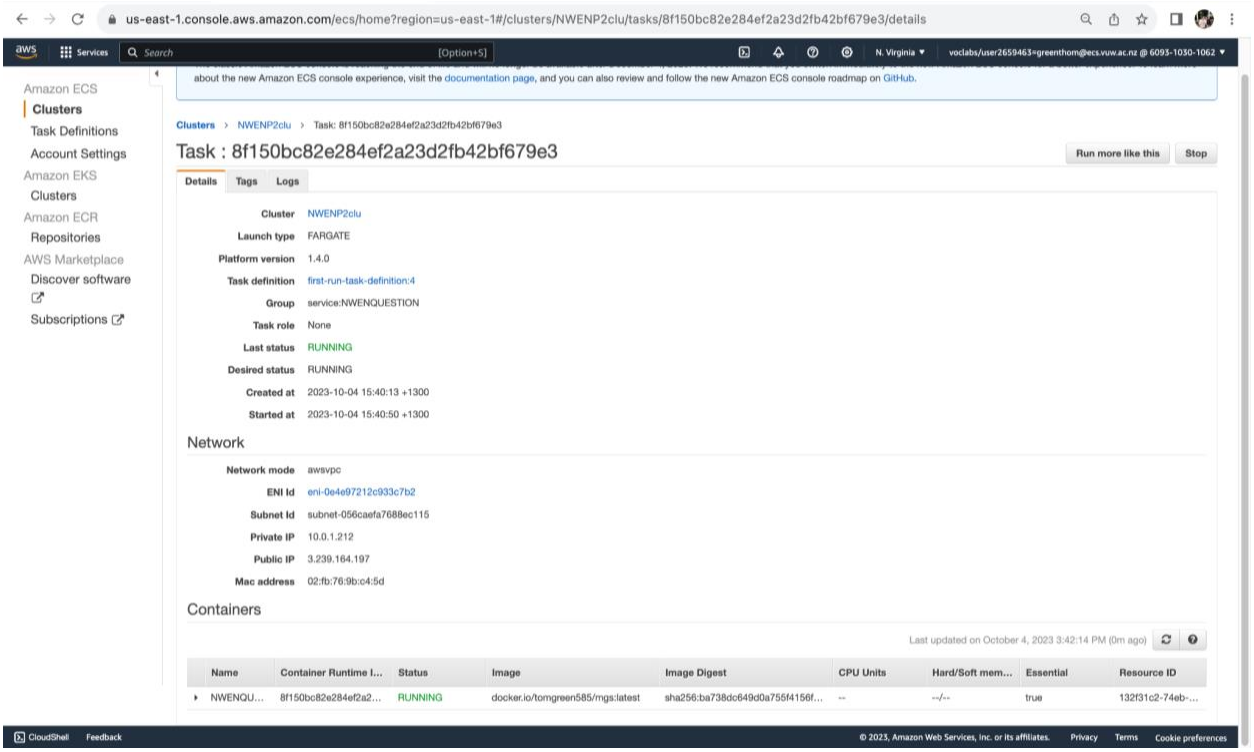
From here, we run the MusicGuruClient on our local machine, connecting it with the IP we found. As you can see, this was successful.

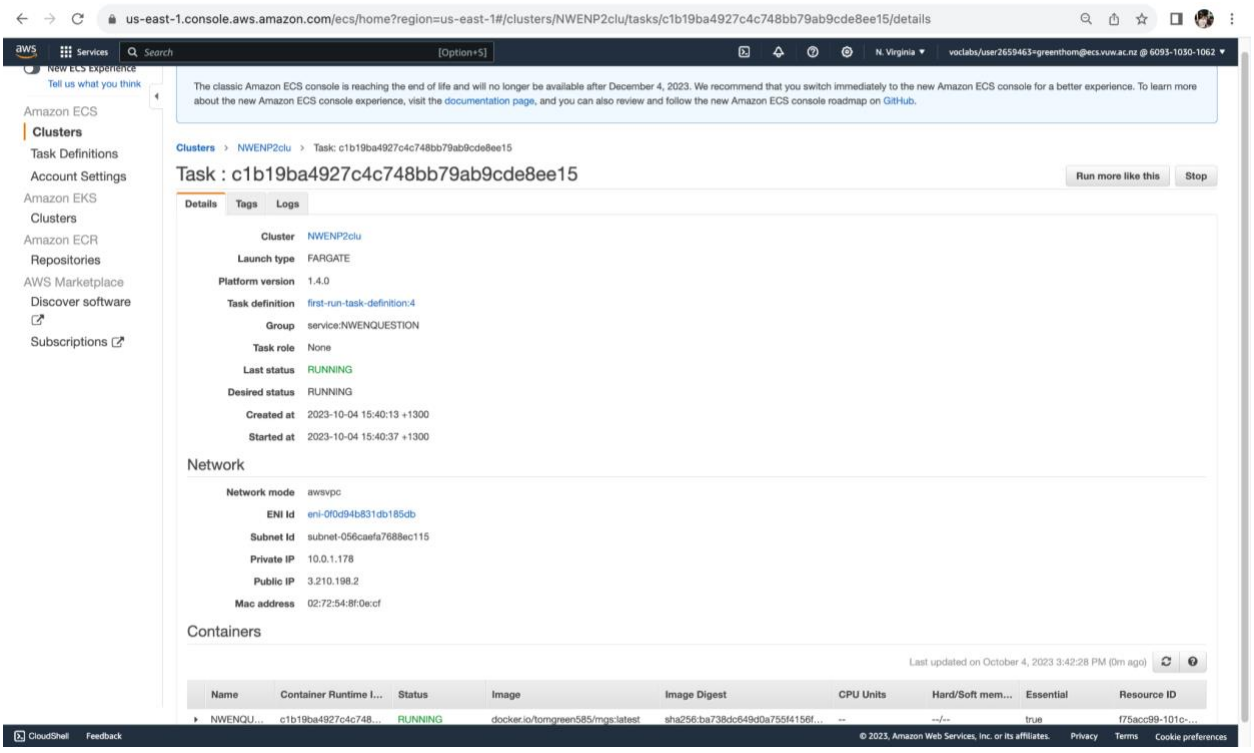


QUESTIONS for B






us-east-1.console.aws.amazon.com/ecs/home?region=us-east-1#/clusters/NWENP2clu/tasks/8f150bc82e284ef2a23d2fb42bf679e3/details
Services
Search
[Option+S]
N. Virginia
voclabs/user2659463=greenthom@ecs.vuw.ac.nz @ 6093-1030-1062
about the new Amazon ECS console experience, visit the documentation page, and you can also review and follow the new Amazon ECS console roadmap on GitHub.
Amazon ECS
Clusters
Task Definitions
Account Settings
Amazon EKS
Clusters
Amazon ECR
Repositories
AWS Marketplace
Discover software
Subscriptions
Clusters › NWENP2clu › Task: 8f150bc82e284ef2a23d2fb42bf679e3
Task : 8f150bc82e284ef2a23d2fb42bf679e3
Run more like this
Stop
Details
Tags
Logs
Cluster NWENP2clu
Launch type FARGATE
Platform version 1.4.0
Task definition first-run-task-definition:4
Group service:NWENQUESTION
Task role None
Last status RUNNING
Desired status RUNNING
Created at 2023-10-04 15:40:13 +1300
Started at 2023-10-04 15:40:50 +1300
Network
Network mode awsvpc
ENI Id eni-0e4e97212c933c7b2
Subnet Id subnet-056caefa7688ec115
Private IP 10.0.1.212
Public IP 3.239.164.197
Mac address 02:fb:76:9b:c4:5d
Containers
Last updated on October 4, 2023 3:42:14 PM (0m ago)
Name
Container Runtime I...
Status
Image
Image Digest
CPU Units
Hard/Soft mem...
Essential
Resource ID
NWENQU...
8f150bc82e284ef2a2...
RUNNING
docker.io/tomgreen585/mgs:latest
sha256:ba738dc649d0a755f4156f...
--
--/--
true
132f31c2-74eb-...
CloudShell
Feedback
© 2023, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/ecs/home?region=us-east-1#/clusters/NWENP2clu/tasks/c1b19ba4927c4c748bb79ab9cde8ee15/details
Services
Search
[Option+S]
N. Virginia
voclabs/user2659463=greenthom@ecs.vuw.ac.nz @ 6093-1030-1062
New ECS Experience
Tell us what you think
The classic Amazon ECS console is reaching the end of life and will no longer be available after December 4, 2023. We recommend that you switch immediately to the new Amazon ECS console for a better experience. To learn more about the new Amazon ECS console experience, visit the documentation page, and you can also review and follow the new Amazon ECS console roadmap on GitHub.
Amazon ECS
Clusters
Task Definitions
Account Settings
Amazon EKS
Clusters
Amazon ECR
Repositories
AWS Marketplace
Discover software
Subscriptions
Clusters › NWENP2clu › Task: c1b19ba4927c4c748bb79ab9cde8ee15
Task : c1b19ba4927c4c748bb79ab9cde8ee15
Run more like this
Stop
Details
Tags
Logs
Cluster NWENP2clu
Launch type FARGATE
Platform version 1.4.0
Task definition first-run-task-definition:4
Group service:NWENQUESTION
Task role None
Last status RUNNING
Desired status RUNNING
Created at 2023-10-04 15:40:13 +1300
Started at 2023-10-04 15:40:37 +1300
Network
Network mode awsvpc
ENI Id eni-0f0d94b831db185db
Subnet Id subnet-056caefa7688ec115
Private IP 10.0.1.178
Public IP 3.210.198.2
Mac address 02:72:54:8f:0e:cf
Containers
Last updated on October 4, 2023 3:42:28 PM (0m ago)
Name
Container Runtime I...
Status
Image
Image Digest
CPU Units
Hard/Soft mem...
Essential
Resource ID
NWENQU...
c1b19ba4927c4c748...
RUNNING
docker.io/tomgreen585/mgs:latest
sha256:ba738dc649d0a755f4156f...
--
--/--
true
f75acc99-101c-...
CloudShell
Feedback
© 2023, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences

Question 1:
IP public and private as well as MAC addresses are unique across. This is to distinguish between each of them.

Question 2:
Load balancer, better communication between client and server it would make it horizontally scalable and handle more or less users. Therefore, as users connect and disconnect, new replicas could be created. This would satisfy the demand and not waste resources. It will evenly distribute client requests across multiple replicas, preventing any single replica from getting overloaded. This helps with sharing the workload and ensuring redundancy. Health checks monitor the status of each replica. If a replica isn't healthy, its automatically taken out of the rotation, so clients are directed only to the healthy ones. Scaling policies automatically adjust the number of replicas based on how much work is coming in. This ensures that clients always have the right number of replicas, not matter how busy things get.