Deploy a Web Application in AWS Elastic Kubernetes Service

Kawtar Oukil

Table des matières

I. Race Objectives

In this project, we are going to focus on three learning objectives:

1. *Build a container image and deploy it to Elastic Container Registry*
2. *Create an Elastic Kubernetes cluster and deploy a web application*
3. *Scale the Elastic Kubernetes Cluster*

One of the most popular methodes to deploy a container is using Kubernetes. Kubernetes is a continued orchestrator with a lot of features to deploy and manage thousands of containers in a faux tolerance in high scalability environment, like an open-source project, Kubernetes is managed by cloudy Native Foundation and this use it for a lot of companies and is being sold by many vendors as a managed service.

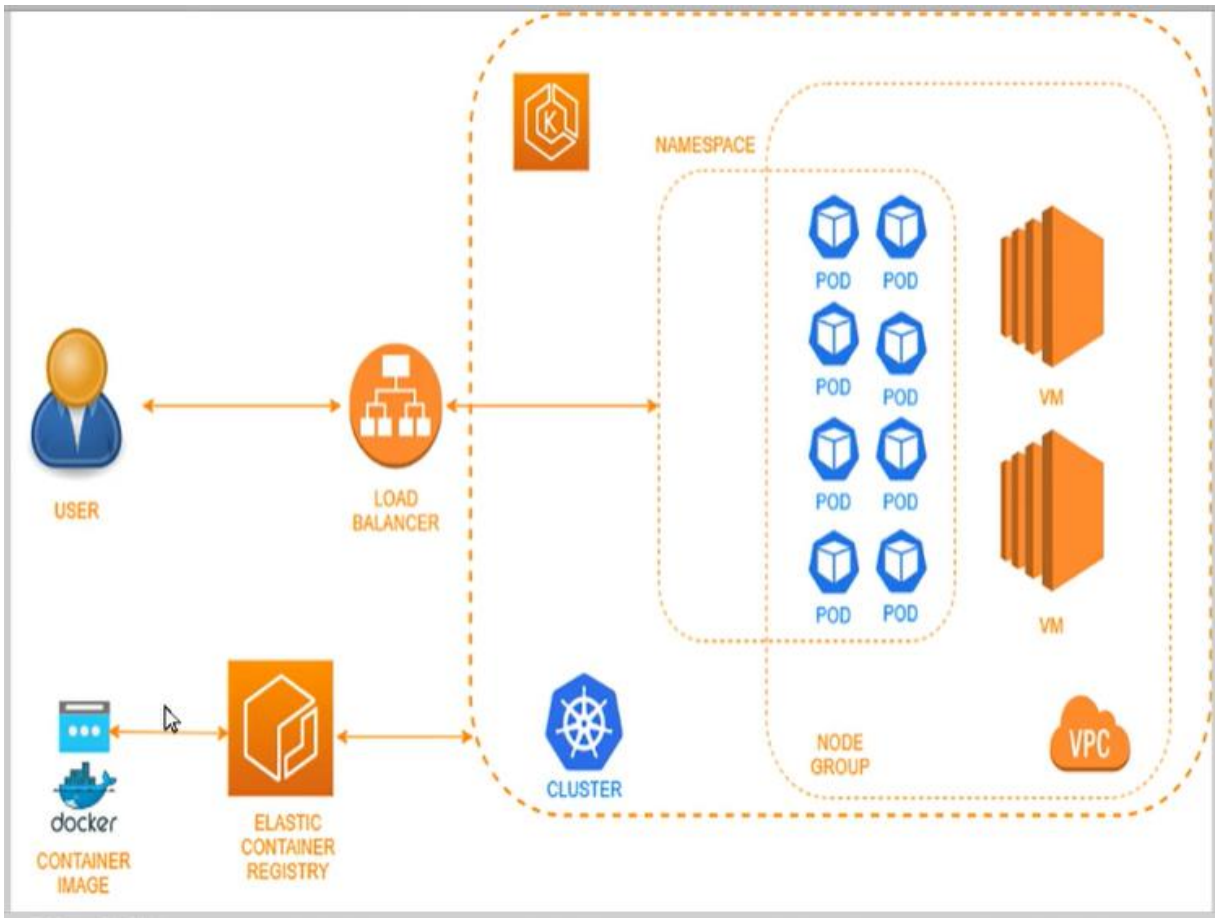
Amazon Web services has a set of management services that provides an entirely ecosystem to create, store and deploy containers in Kubernetes.

The main services is elastic Kubernetes services, which is Kubernetes offer as a services.

In this project we will understand how to clone a web application project, build a docker container image uploaded to AWS and deployed into Elastic Kubernetes servicing.

Task 1: Understand the entire project elements and scope as well as be able to describe basic aspects of Amazon Elastic Kubernetes Services

In this first task, we gonna see our project architecture:

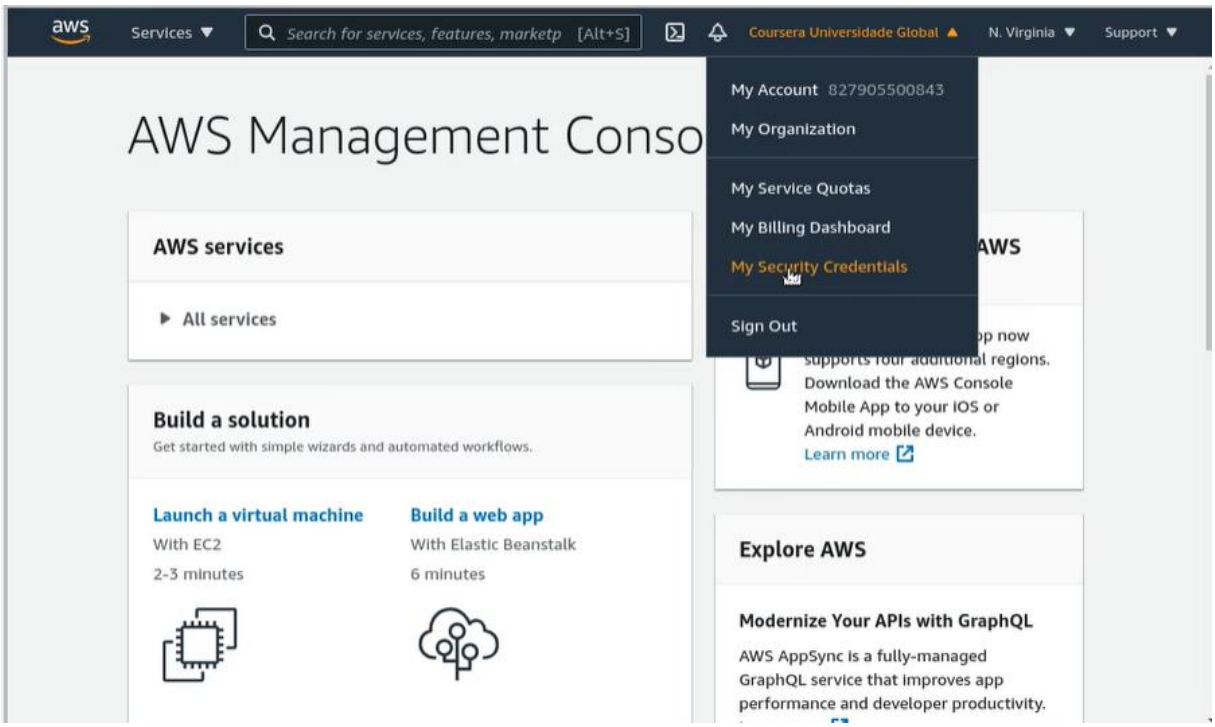


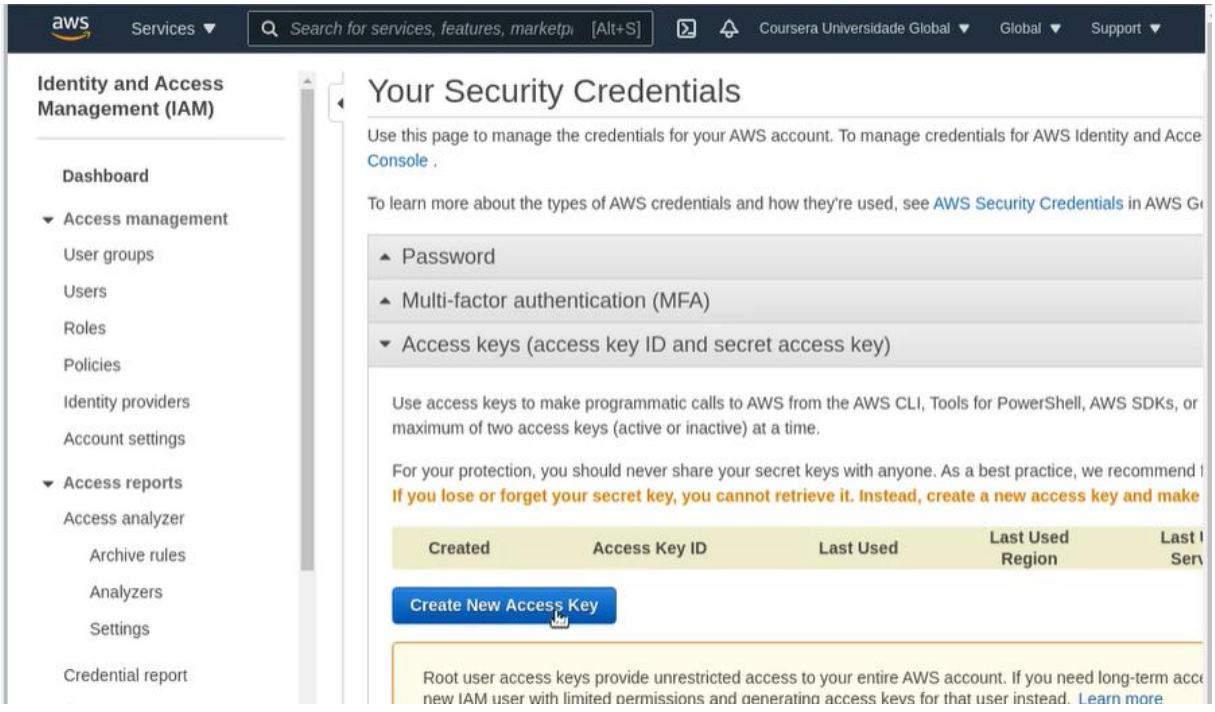
In this diagram above, we see the elements that we will create in AWS. Some elements will be created for us and some by AWS based on the choosing configuration on the bottom, we can see we have a container image for our application. We will build the image and push it to elastic container has history which will be our depository for our image. Later we will use this image started in Alaska container history for deployment in our Kubernetes cluster. We will also create a Kubernetes cluster with one node group. This node group, we have at least to our machines and we will be able to have thousands of pods in this empty group.

How elastic Kubernetes services is a managed services will create for us other elements in the VPC networking where the virtual machines will be deployed like the load balancer to support the end user in grass and other configurations.

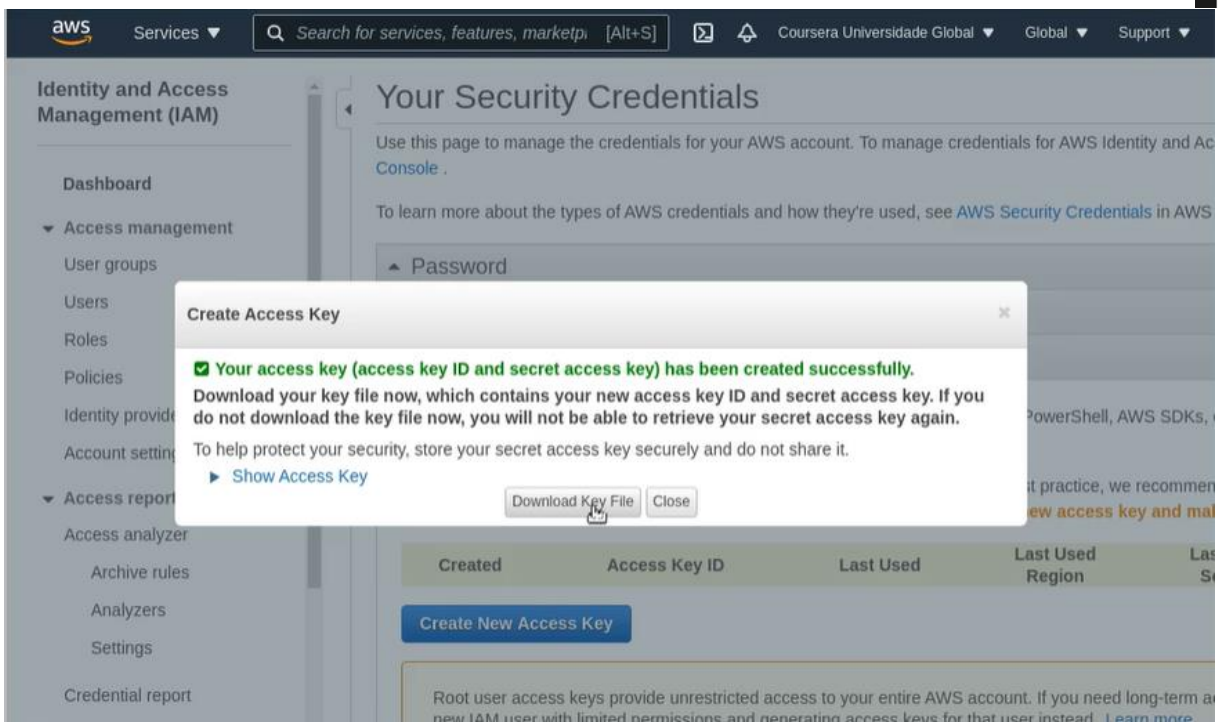
Task 2: Set all permission needed for our project, create roles and policies

The first thing we need is to create one AWS access key:





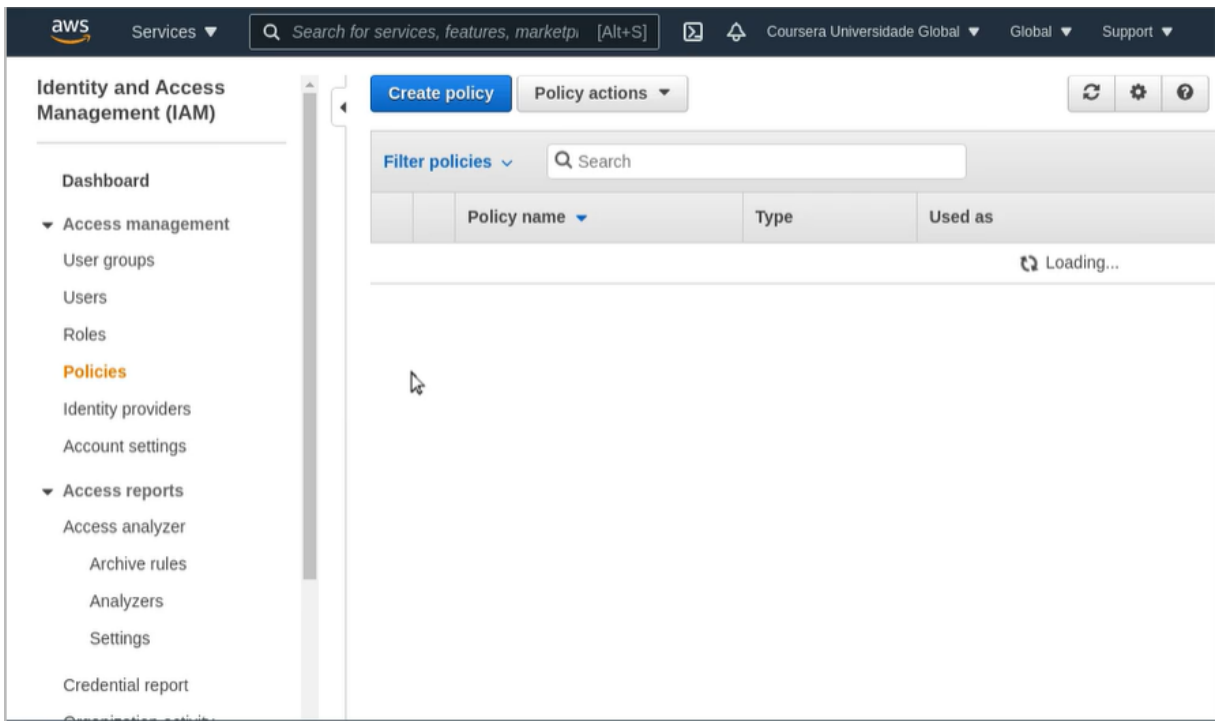
And we open Access Area, and we create a new key using the creating the ‘creating new access key’ button



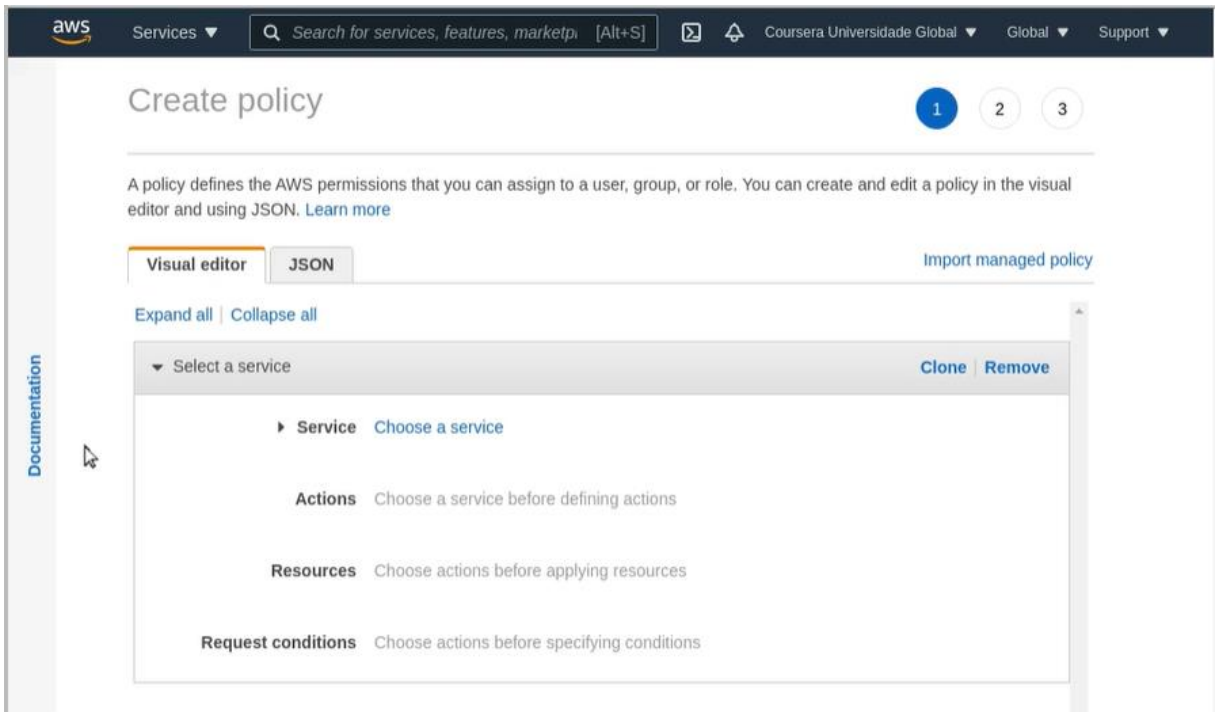
Then the access key was created successfully. And later, we click on 'Download Key File' button to save the credentials to our cloud desktop. This key give access to our AWS services, that 's why it is recommended to delete this key after finishing our project.

Now, we need to create some roles and policies to be used in Kubernetes services

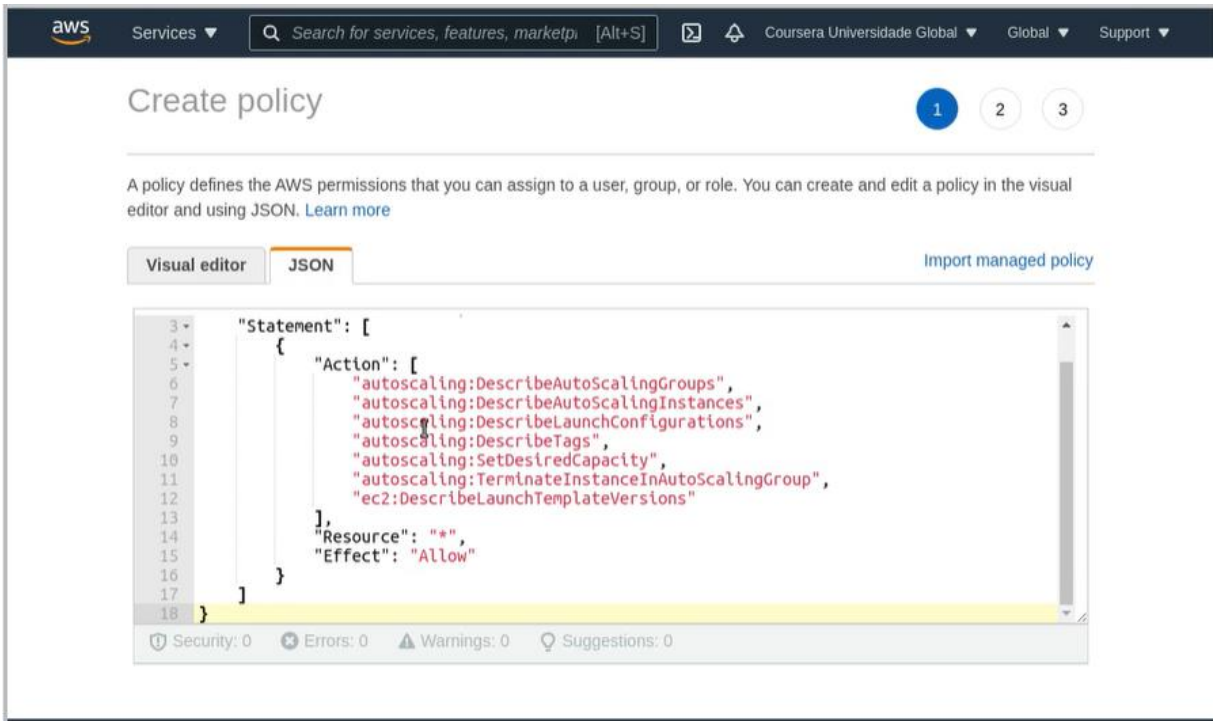
In the left of the menu, we select 'Policies':



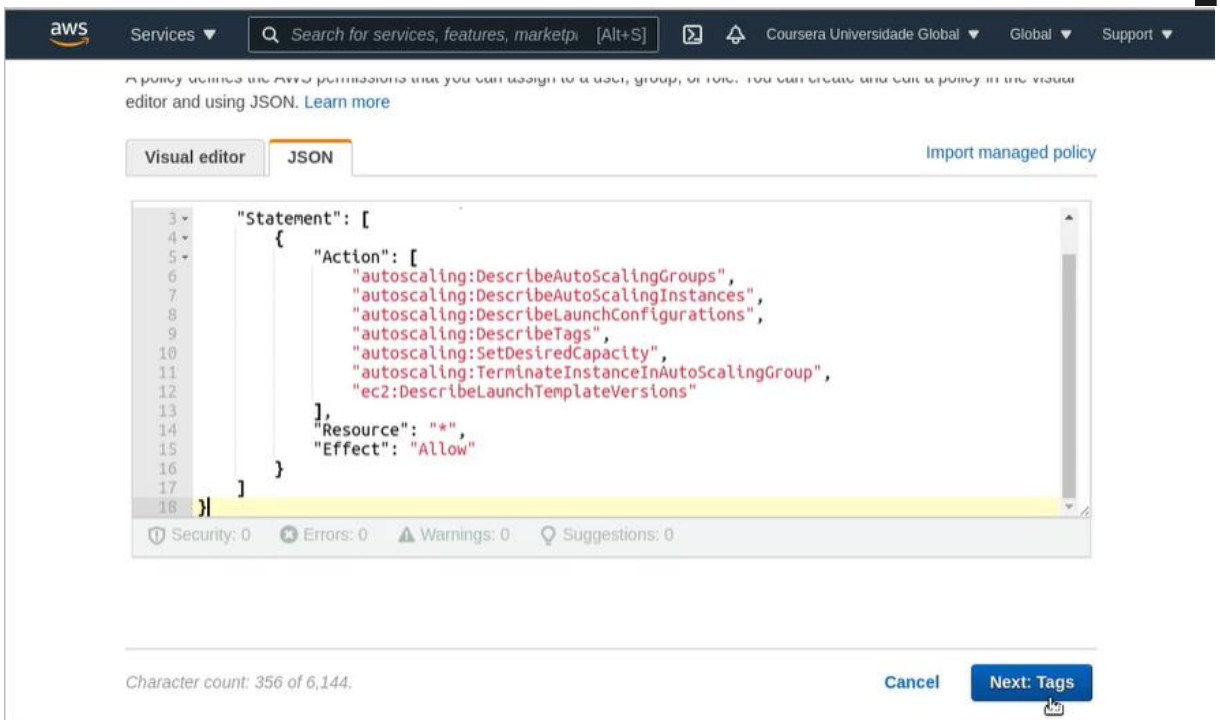
And the we click on 'Create Policy' to create a new policy:



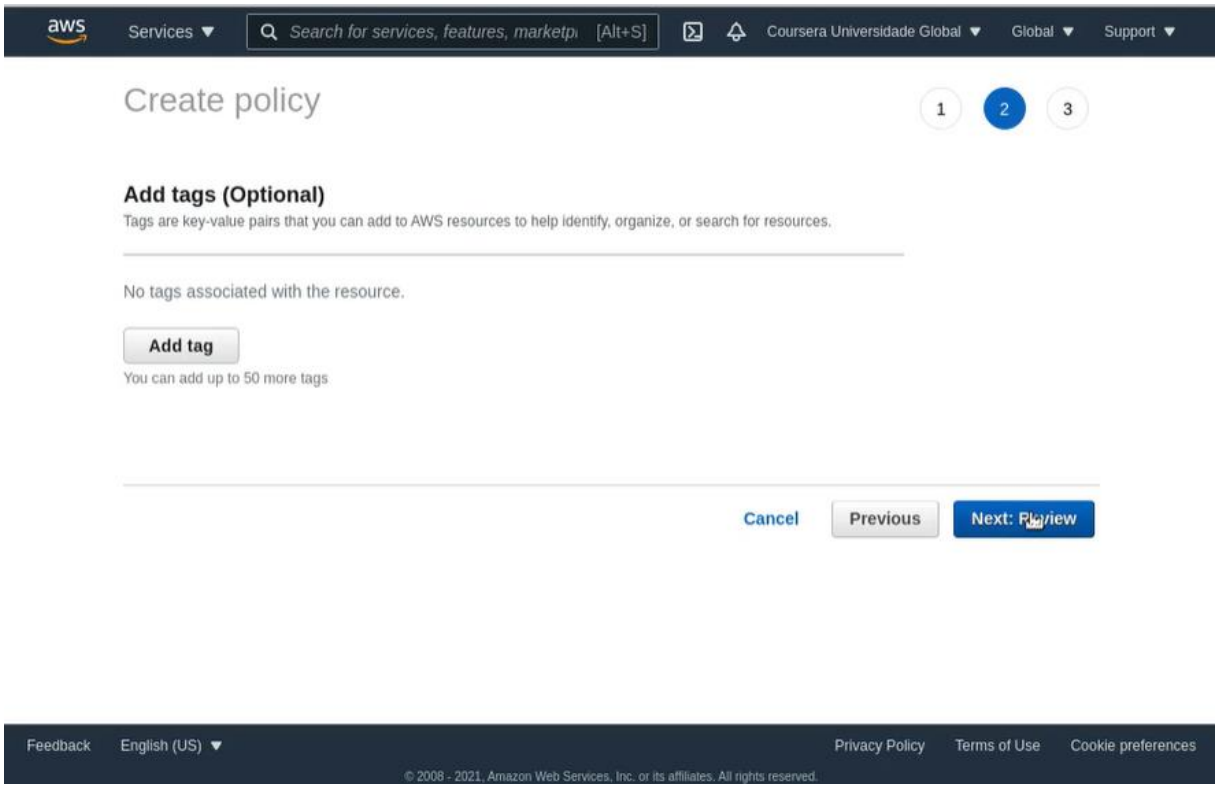
And here, in the JSON space, we gonna add a code:



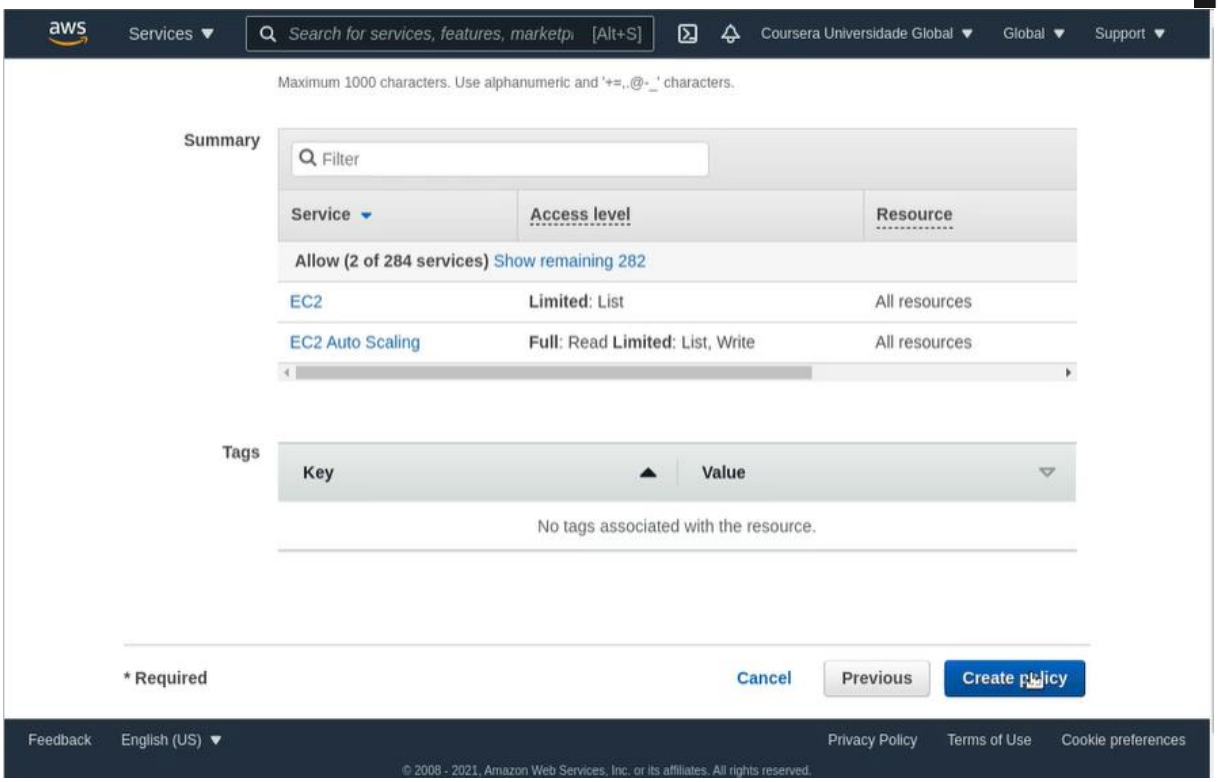
Next we click on: ‘next Text’:



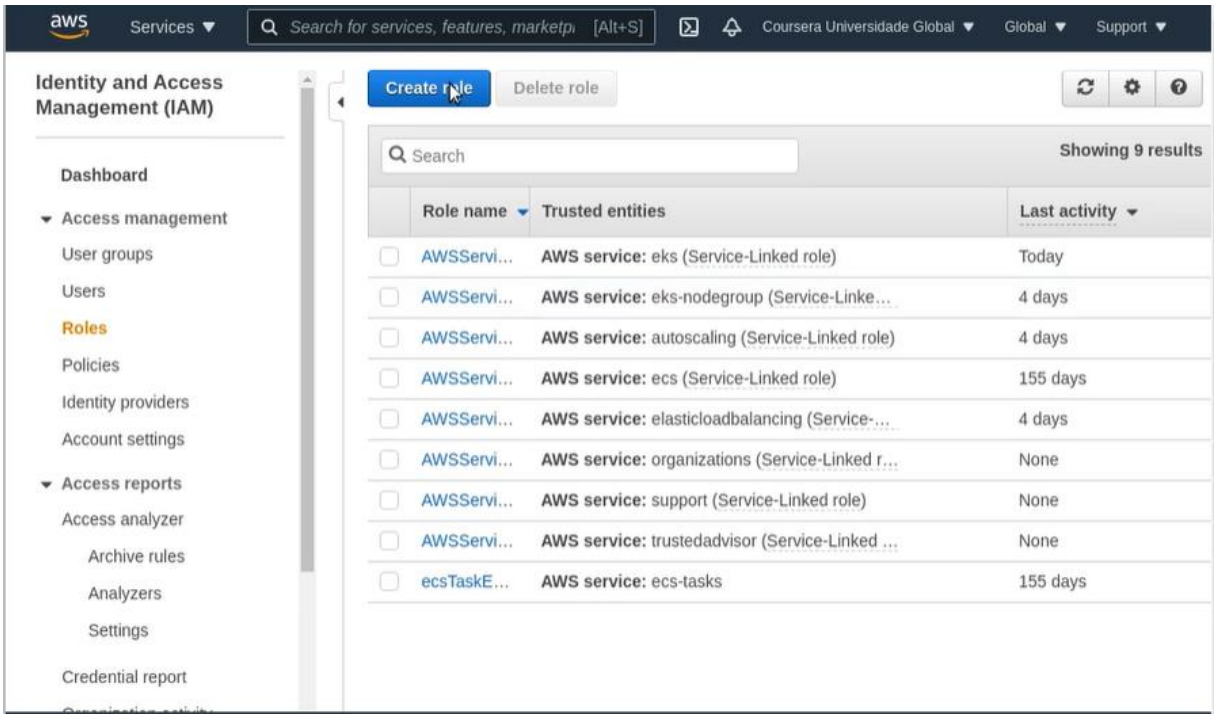
And 'Next Review':



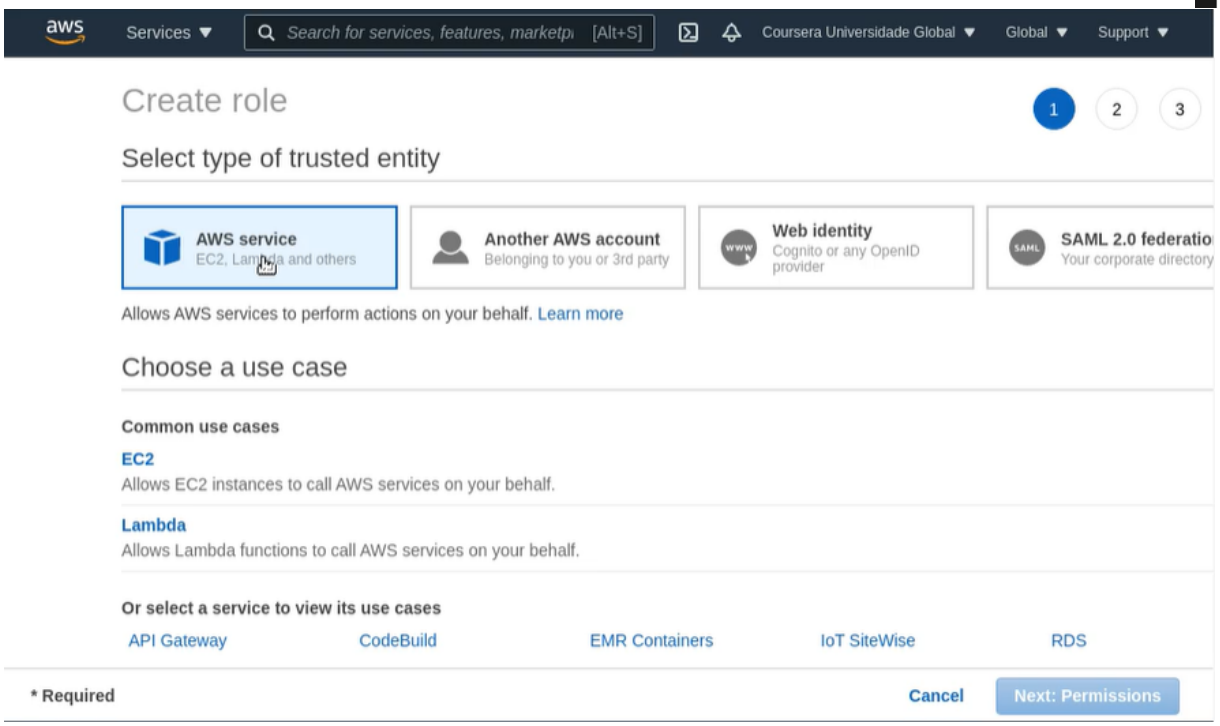
After, we add a name to the policy, and we click on 'create policy':



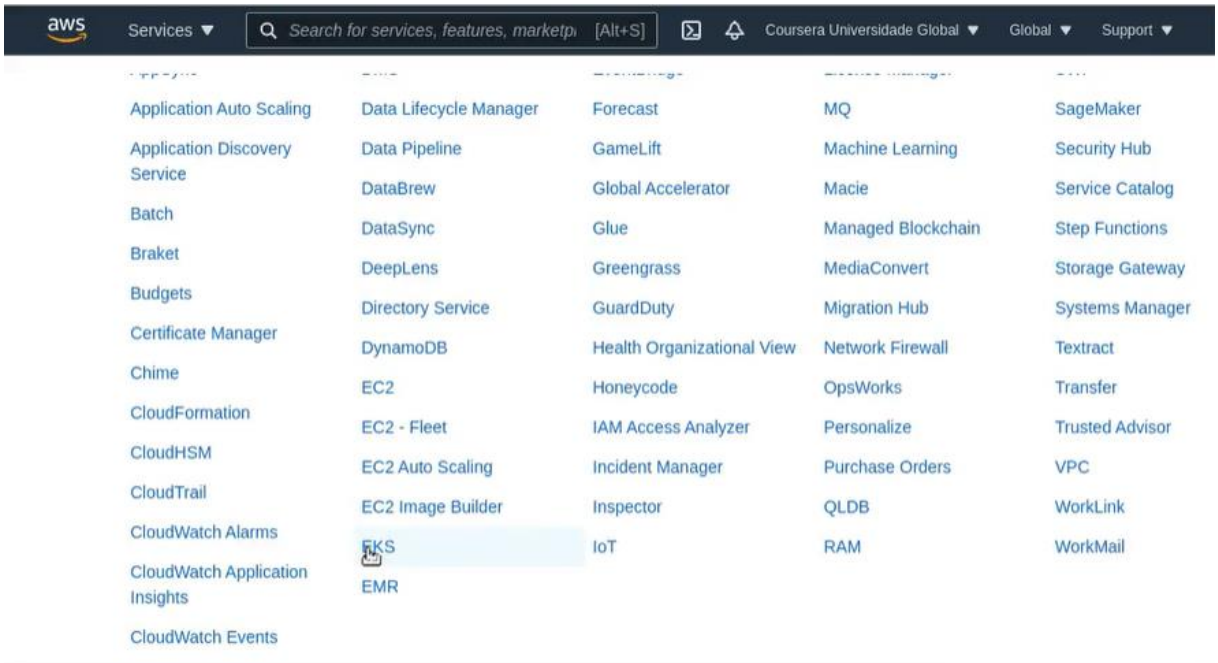
Similarly to policy, we gonna create a role, then we gone on 'roles' option in the menu, and we click on 'create role' button:



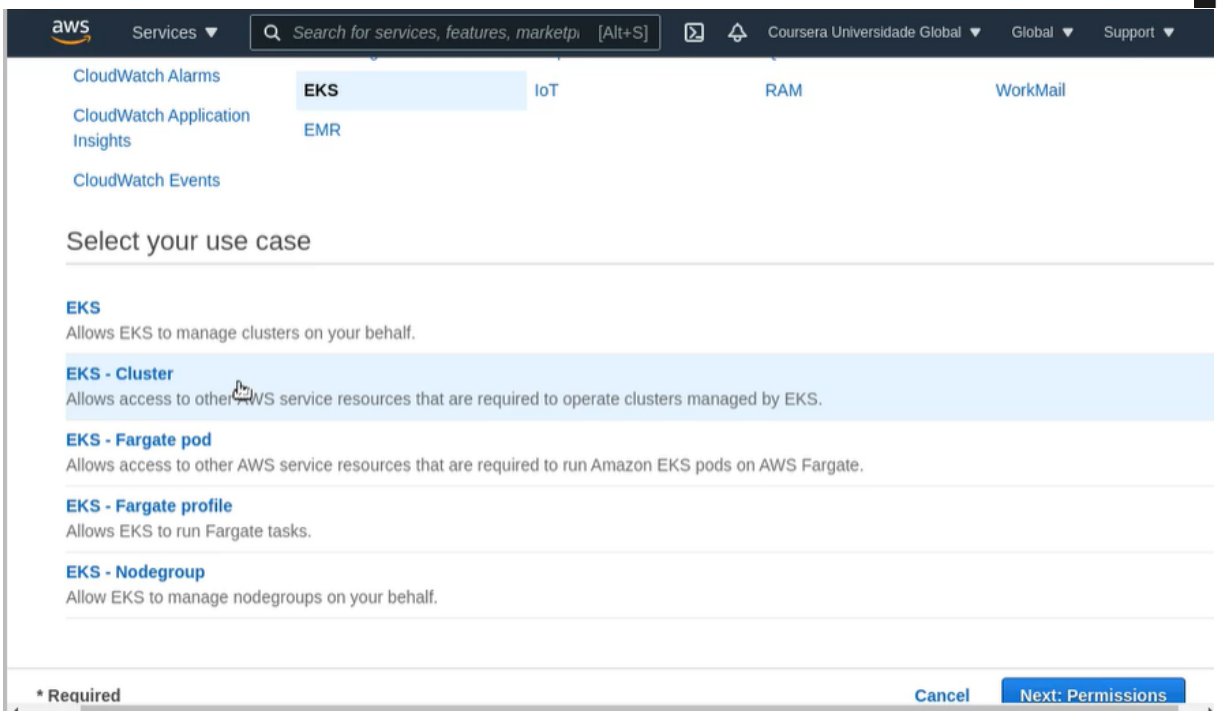
Then we choose the AWS service option:



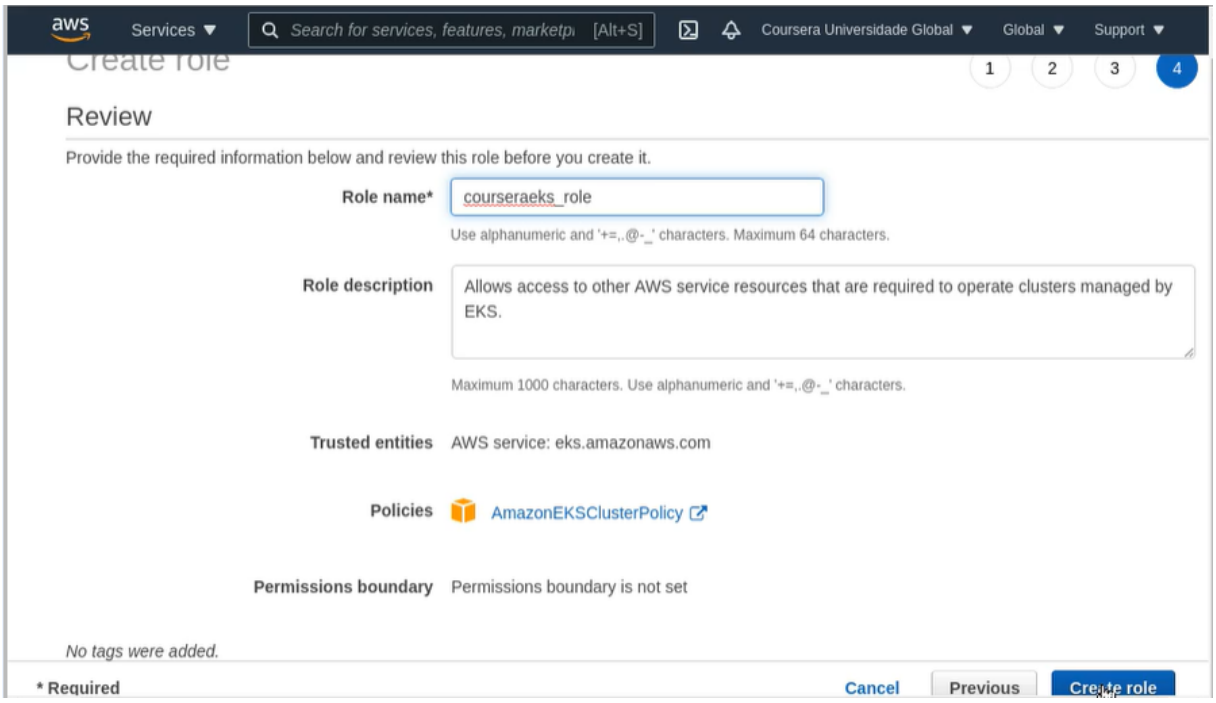
And, in the use case we select the EKS option:



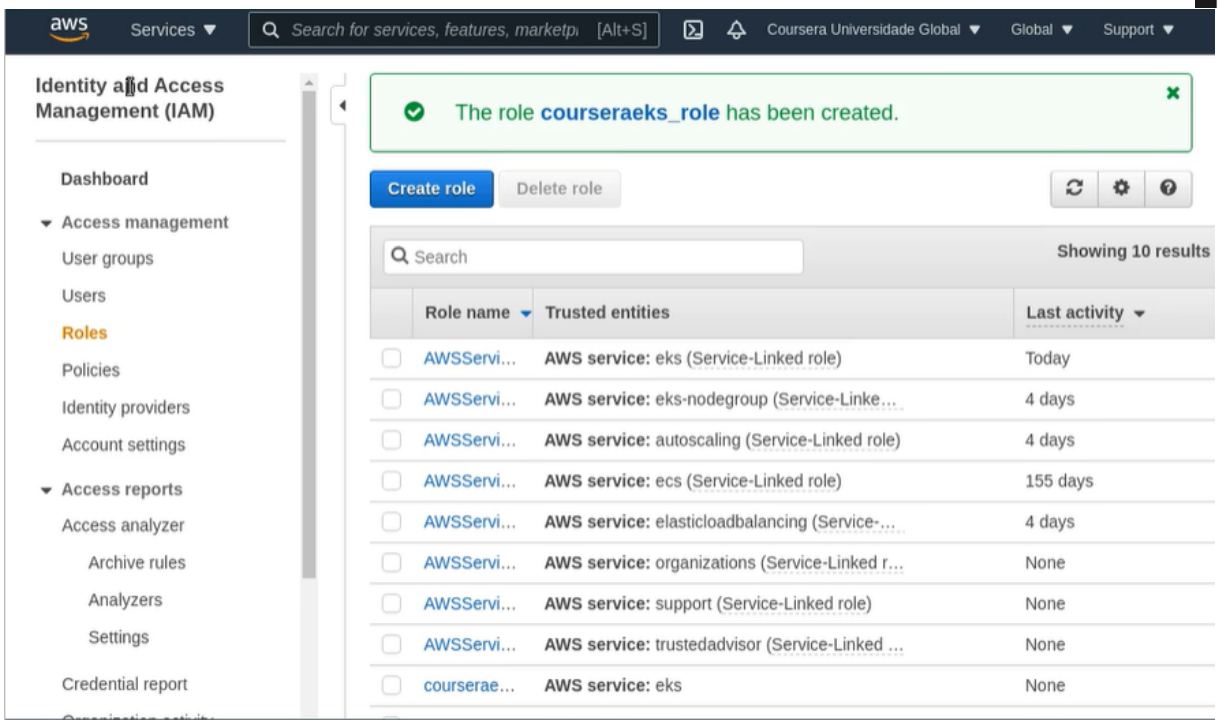
Later, we select ‘EKS cluster’:



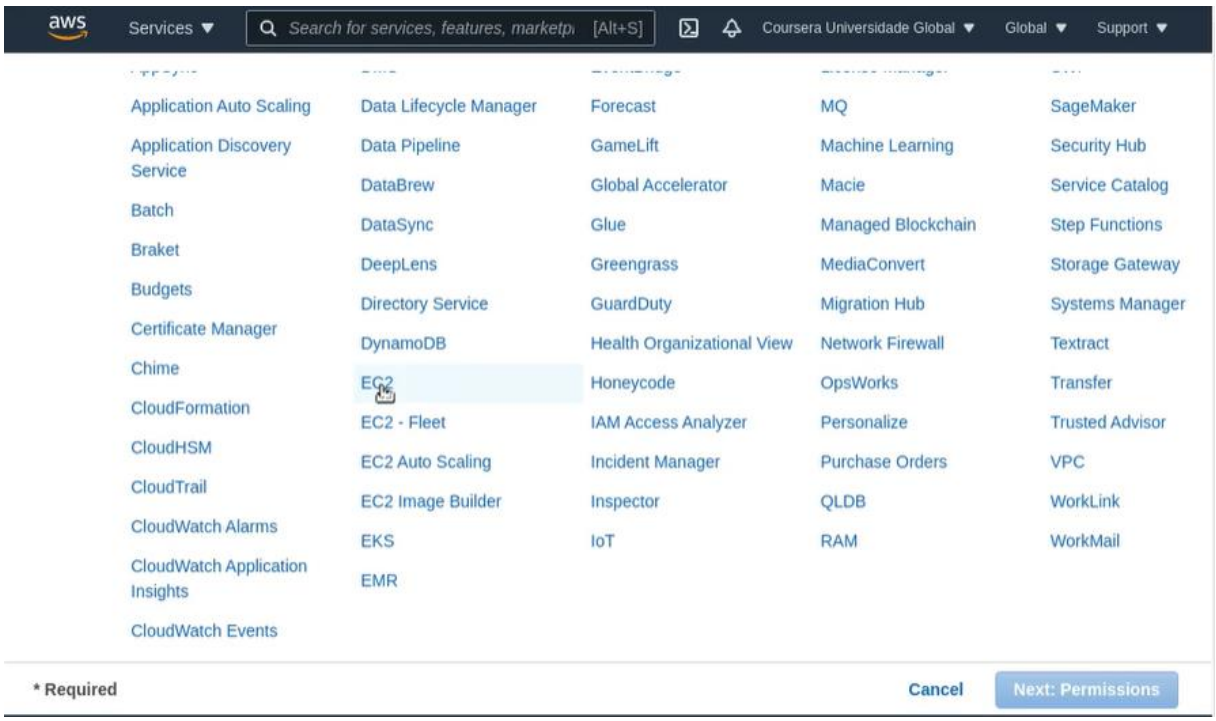
And, then click 'Next Permission', and then we add a name to the role, and we click on 'create role':

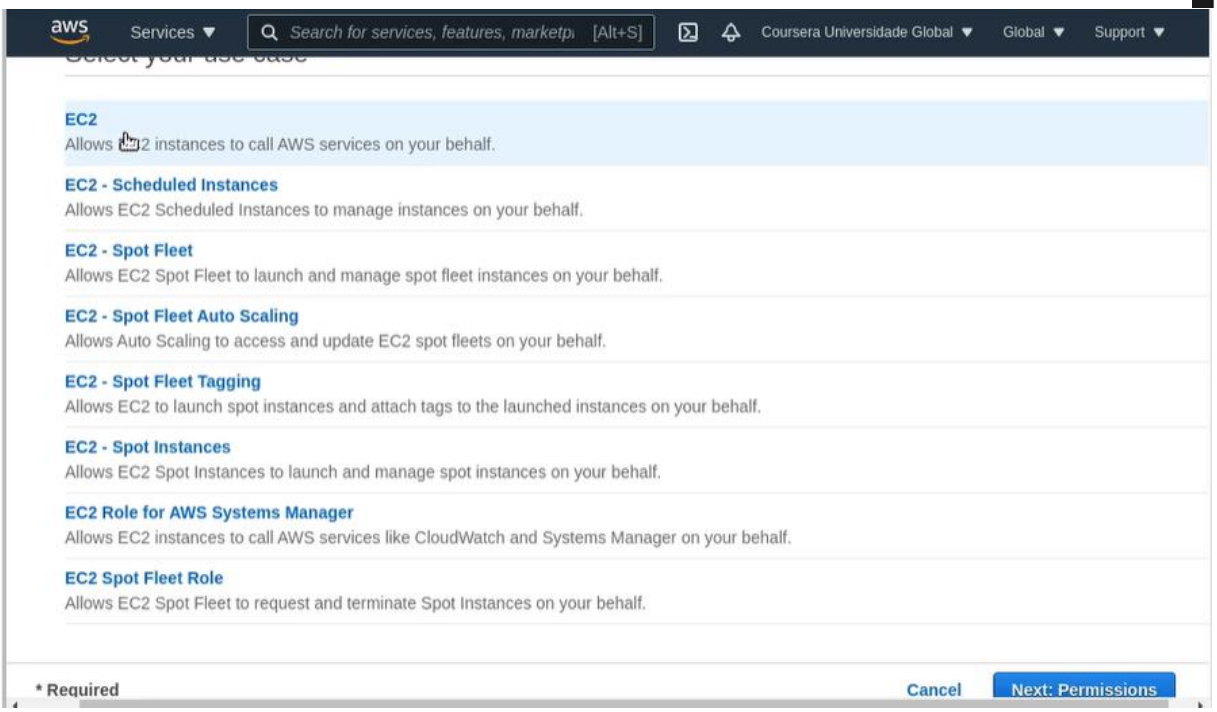


And, then our role was created successfully:

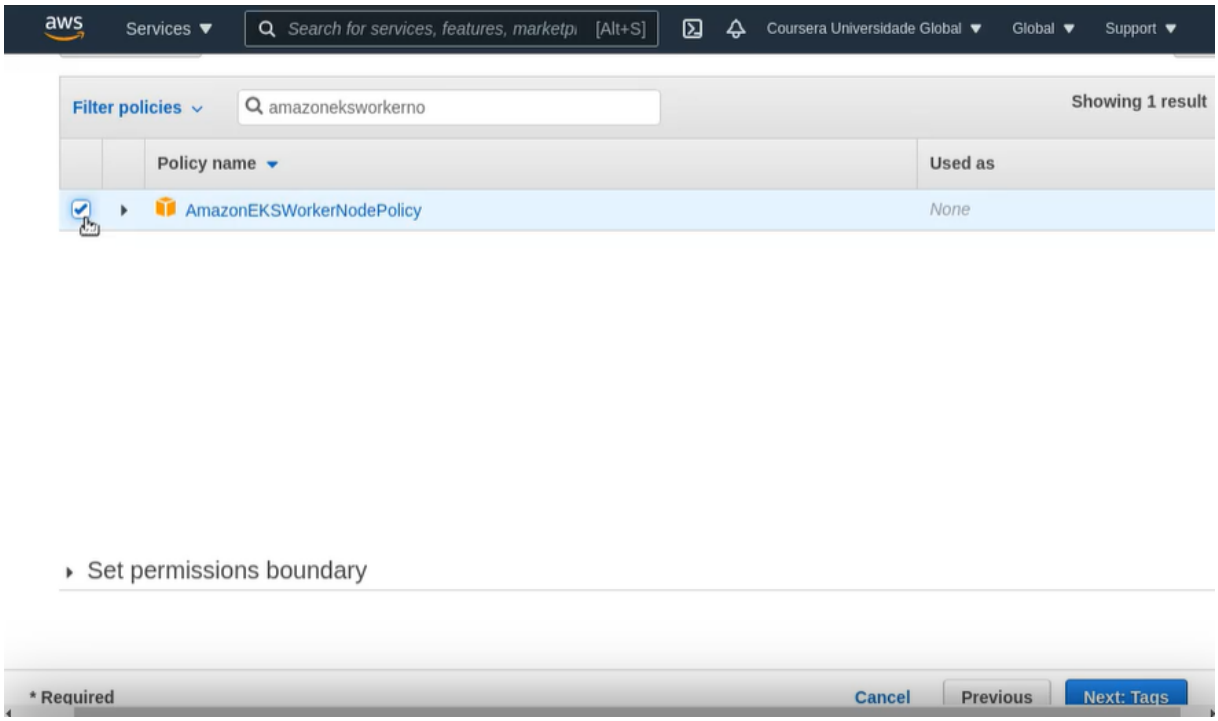


In this project, we will to create a second role, with another option of use case called '**EC2**',

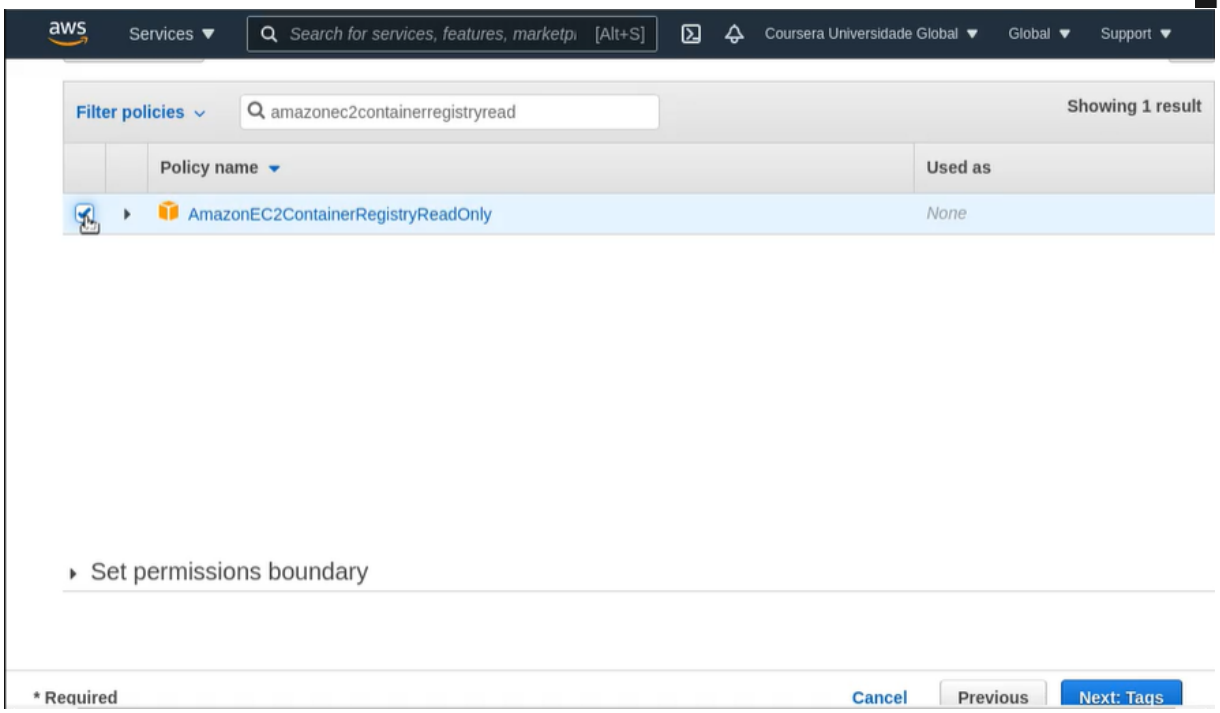




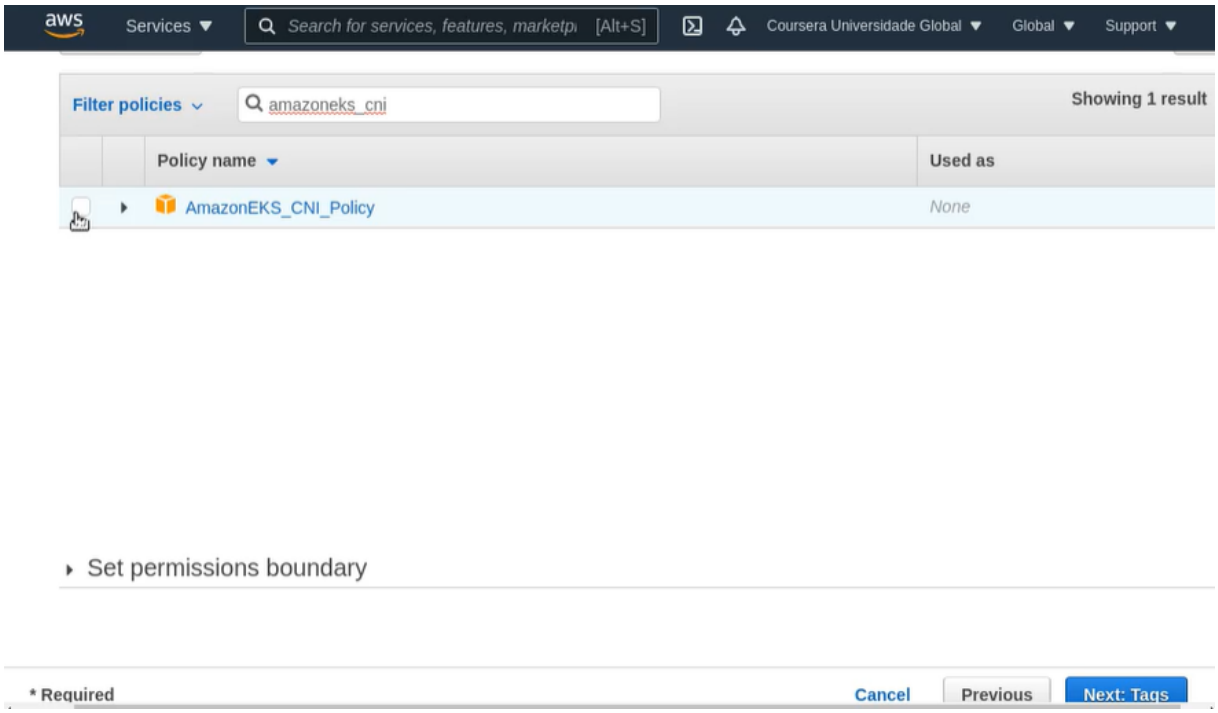
Later, we need to choose some policies, the first one **is AmazonEKS worker node policy**:



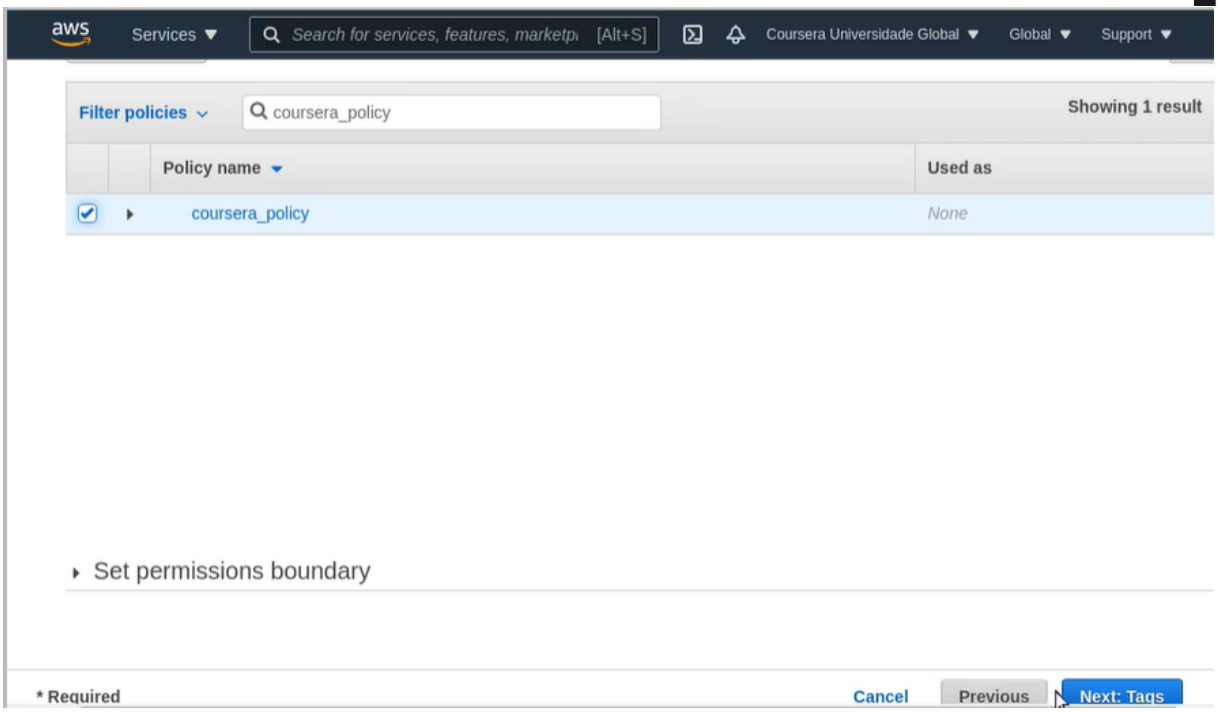
Next one is **AmazonEC2ContainerRegistryReadOnly**:



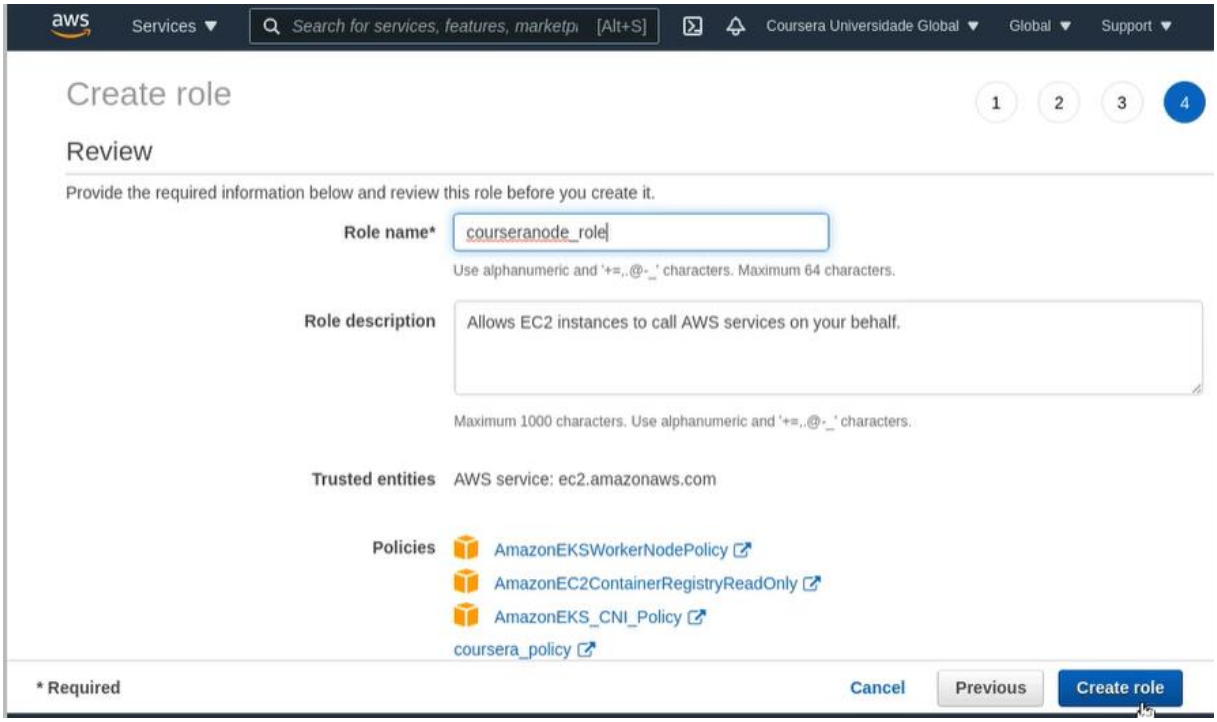
And the next one is **AmazonEKS_CNI_Policy**:



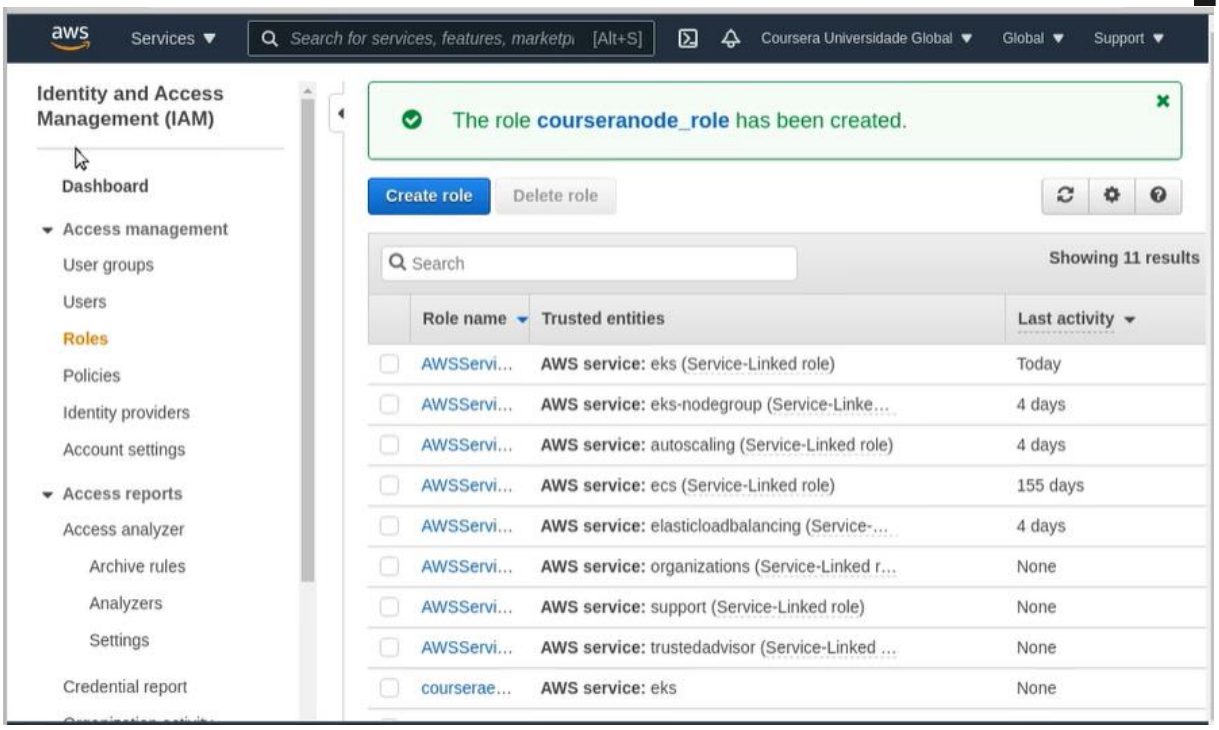
And now, the next one is our policy created a few seconds ago, called '**coursera_policy':**



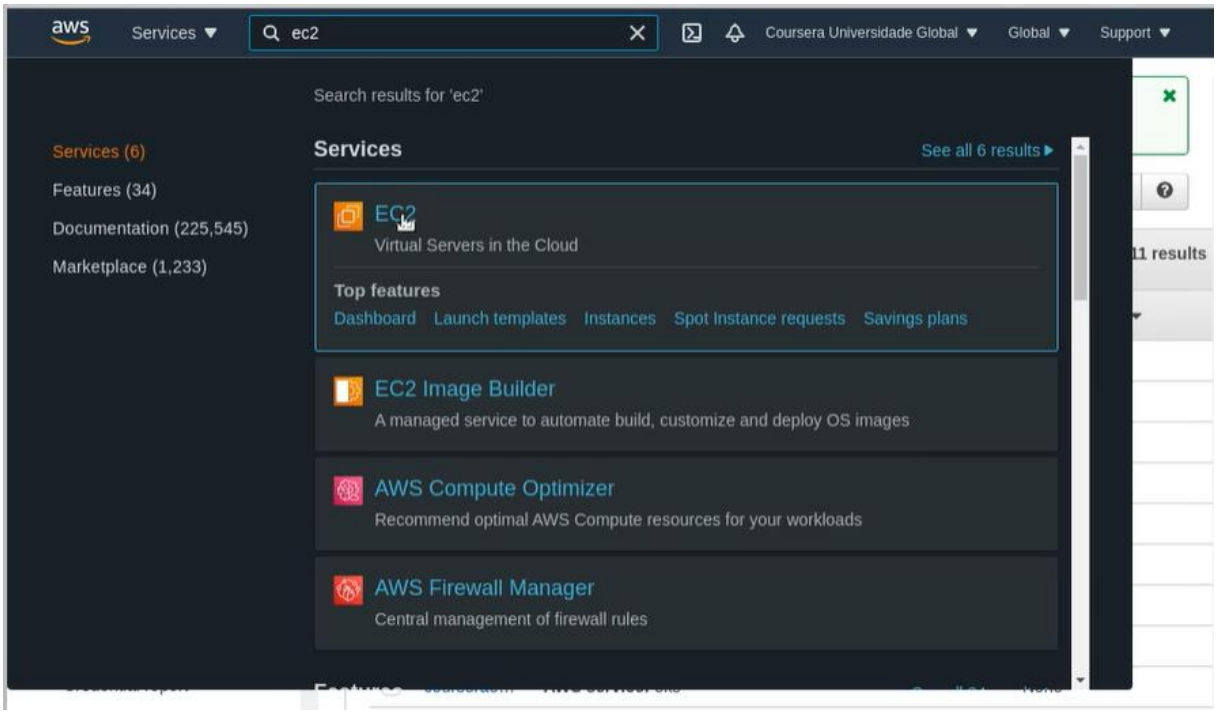
And then we click on 'next tags' -> 'next review', and confirm if our 4 policies are in the list, and we will give a name of the role, and click on 'create role':



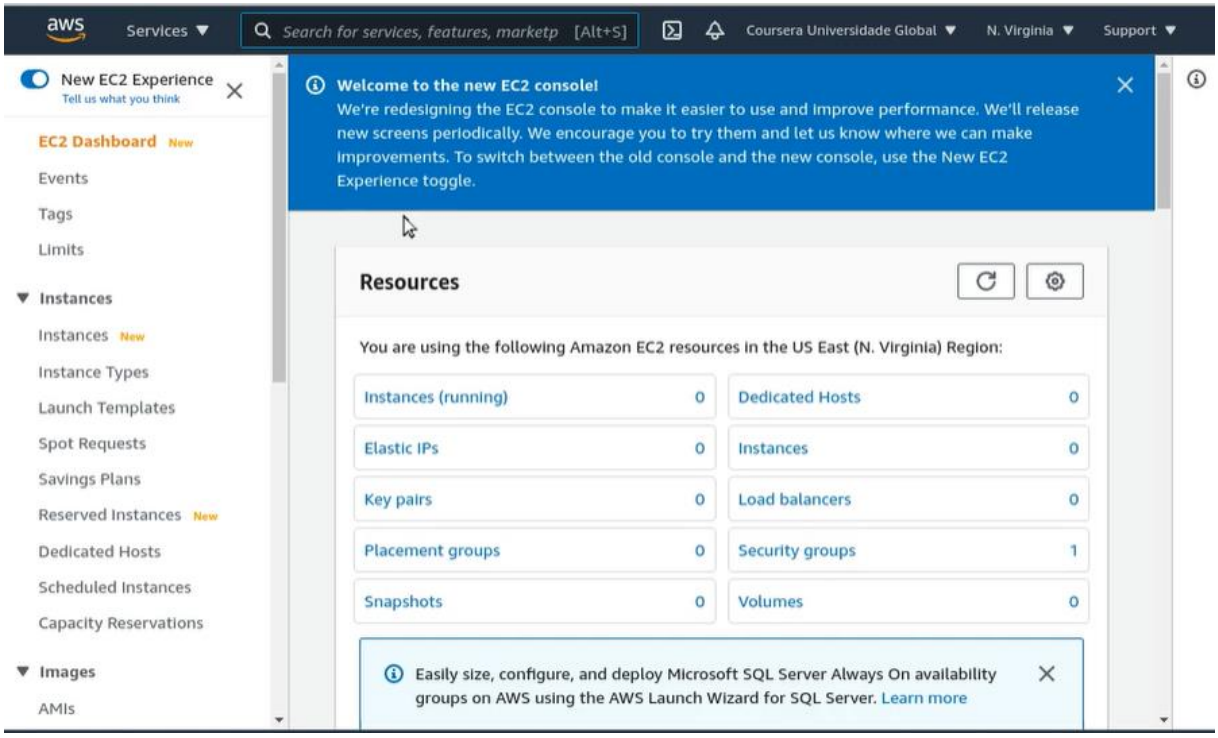
Then our role was created successfully:



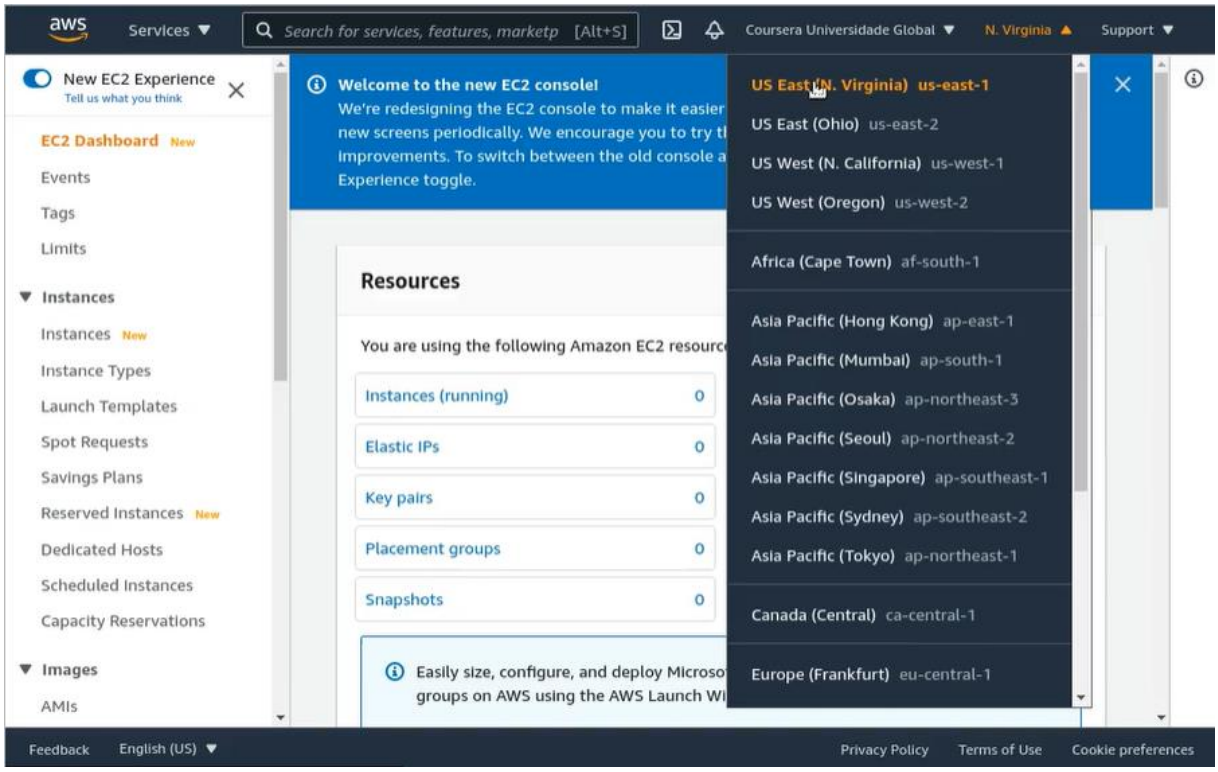
Then, now we need to create an EC2 key pair, it is used to authenticating EC2 to off our machines and it is used to authenticate in our node group later if we needed. So, to do that we need to go to the institute dashboard page on the search bar, and we click on '**search on institute'** to open the institute dashboard:



EC2 is a region basic in services in AWS:

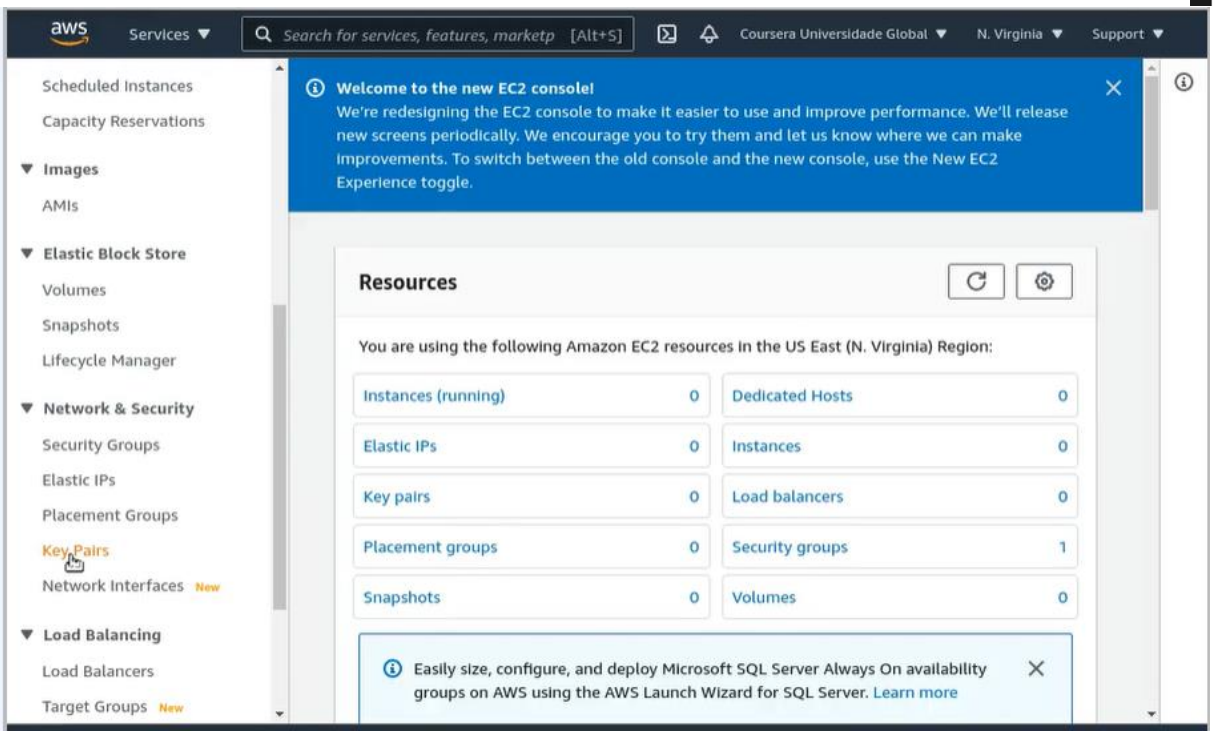


The first thing we need to do is to choose the region where we will create our resources in this project. On top right we gonna choose our region

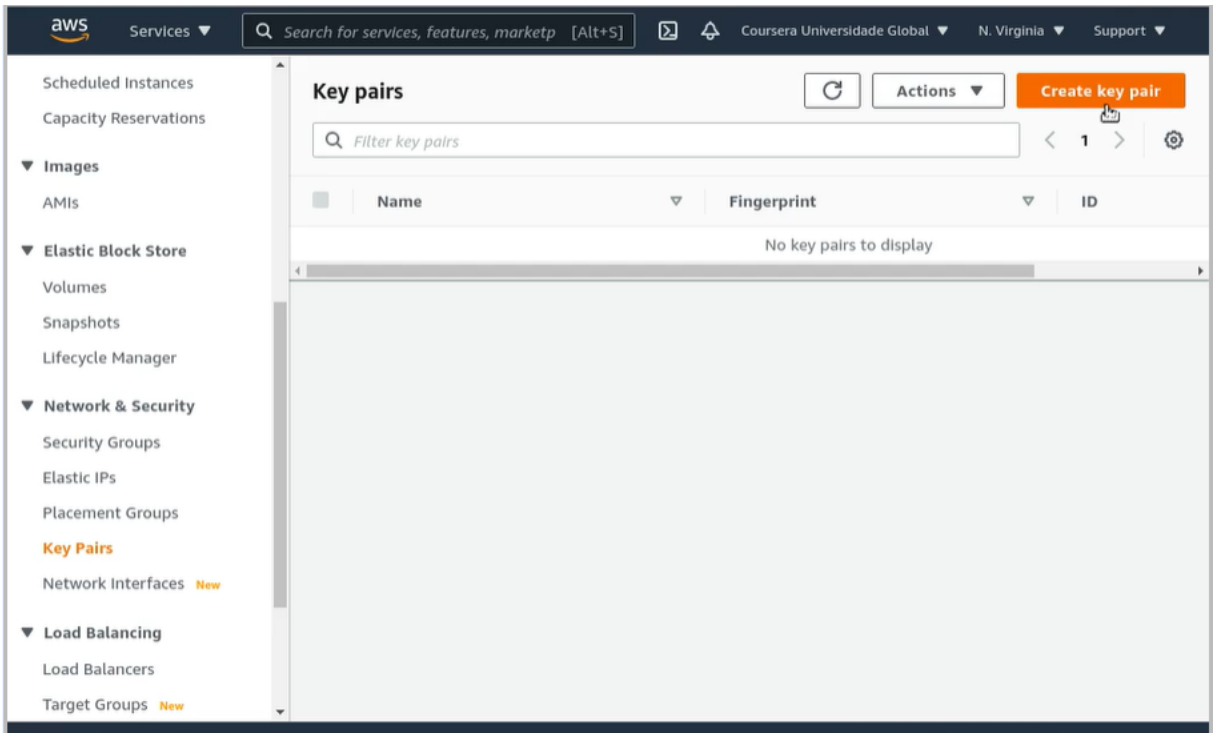


We will choose the US East.

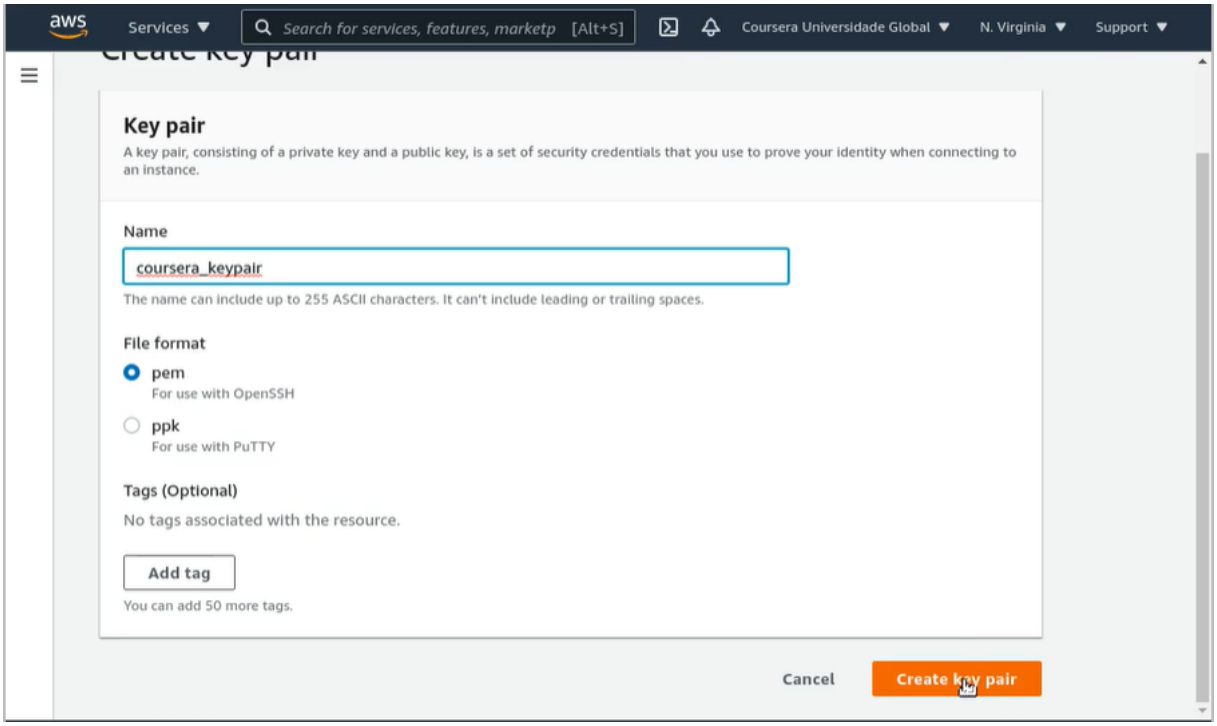
Later, in institute dashboard, in the left menu, we gone to key pairs option below network and security, and we click on '**key pairs':**



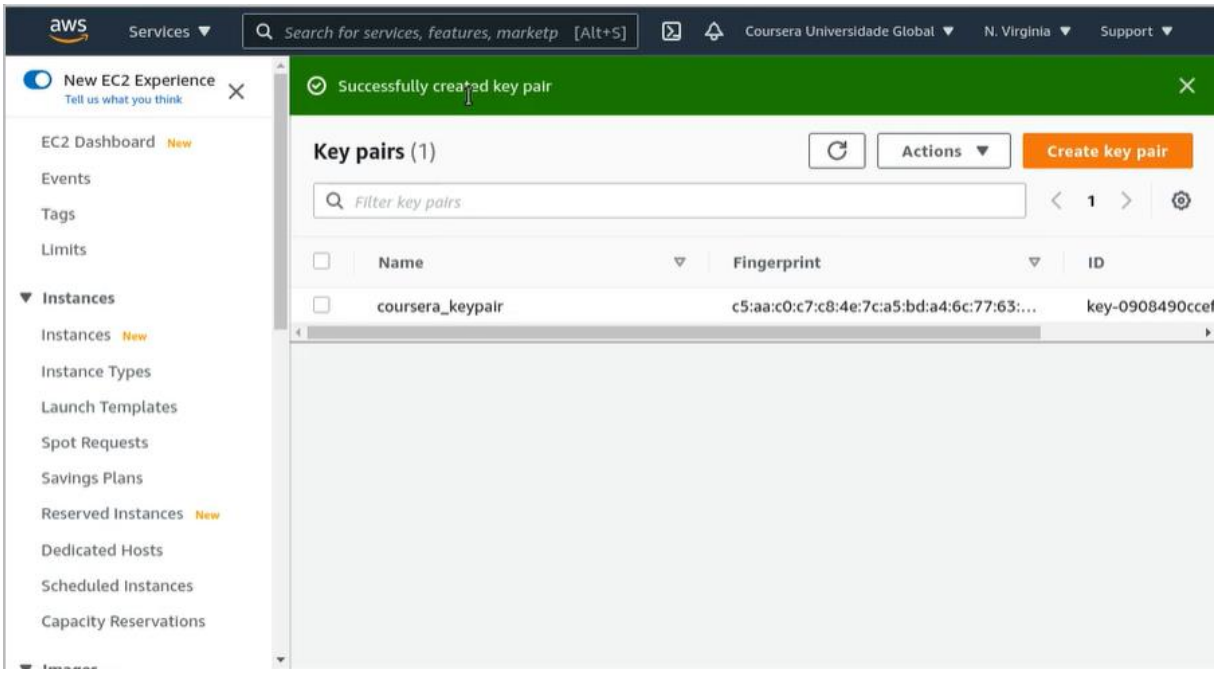
Now, we click on '**create key pair'**:



Now, we choose a name to our key pair, and we select the format, and then we click on '**create key pair':**

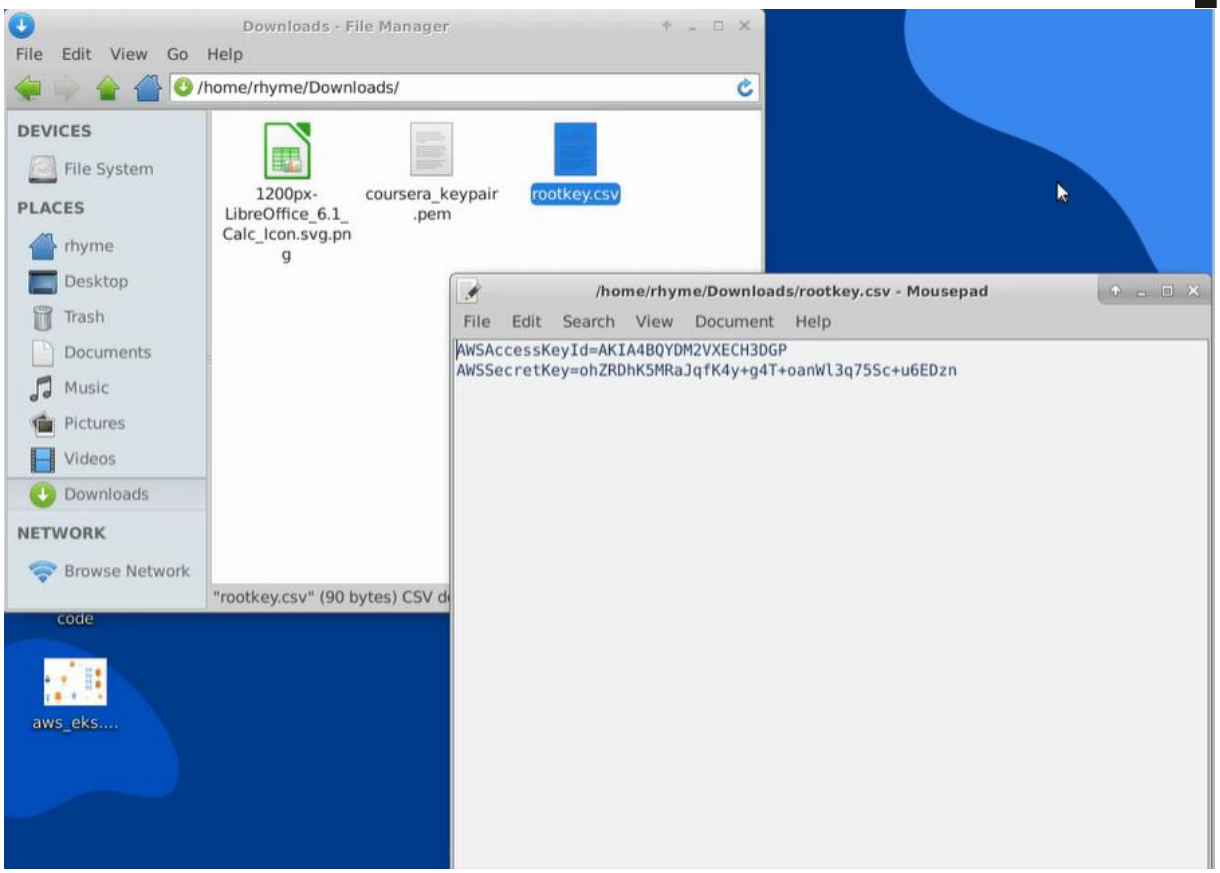


So, now we have our key pair created:

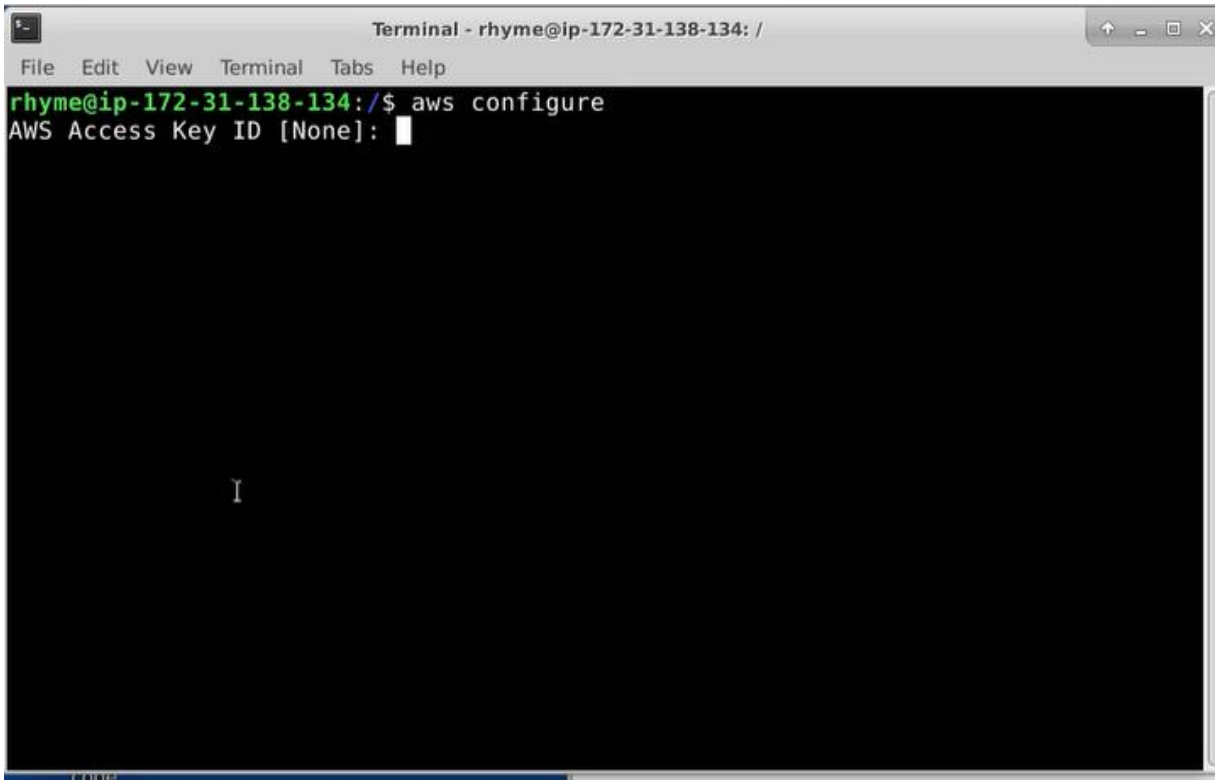


Let's now configure the AWS credentials in command line interface.

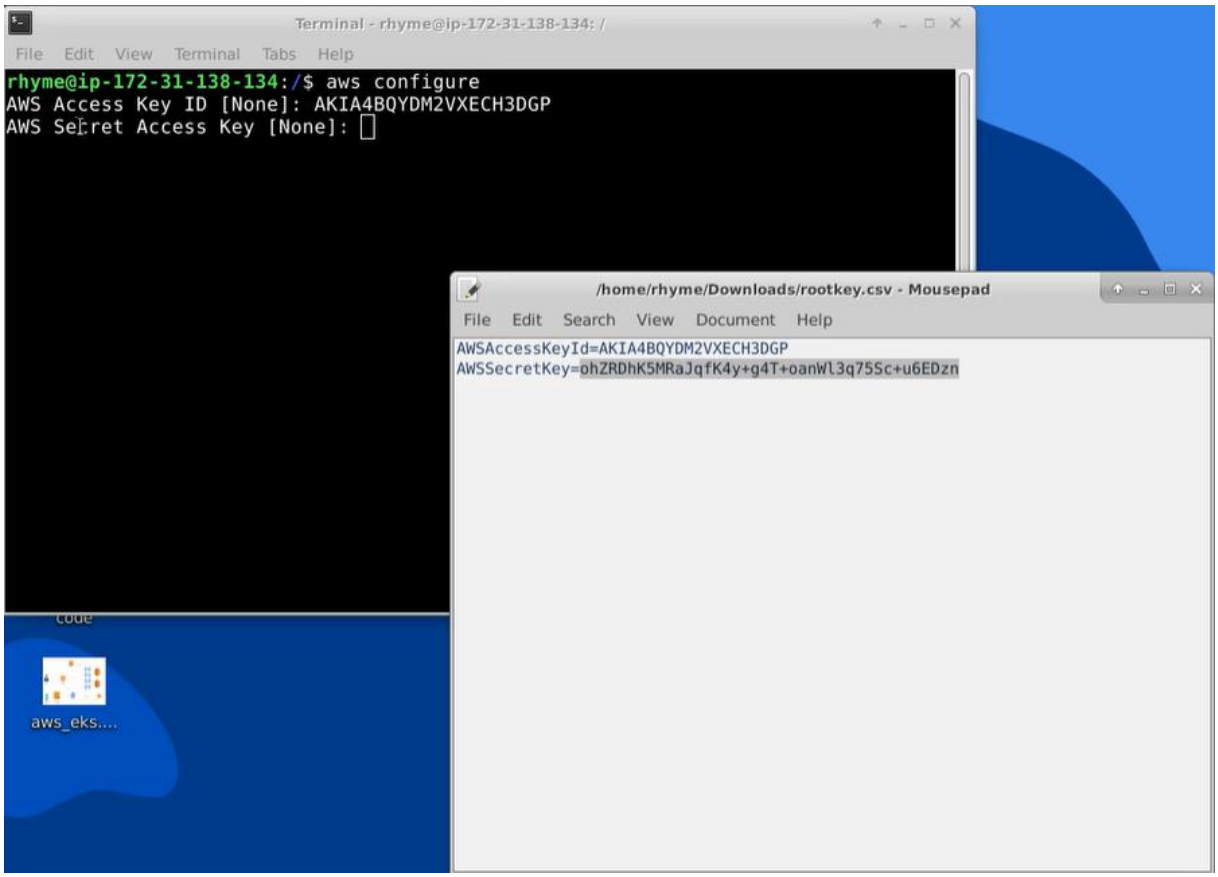
AWS common interface is already installed and we our access keys created and downloaded; it is called '**rootkey.csv'**:



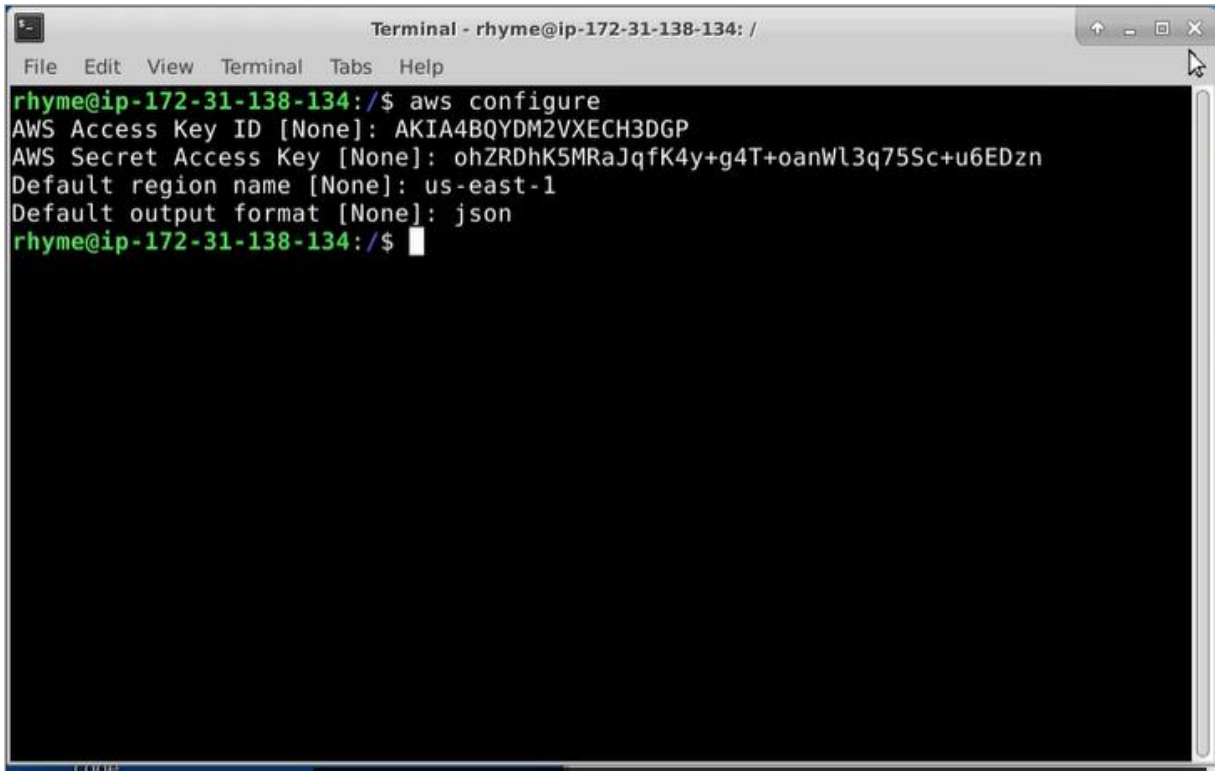
We gone to the terminal and we type the command: **#aws configure:**



And here we paste the key ID and the Secret Access key from the csv file:



Later, we type the region name and the output format:

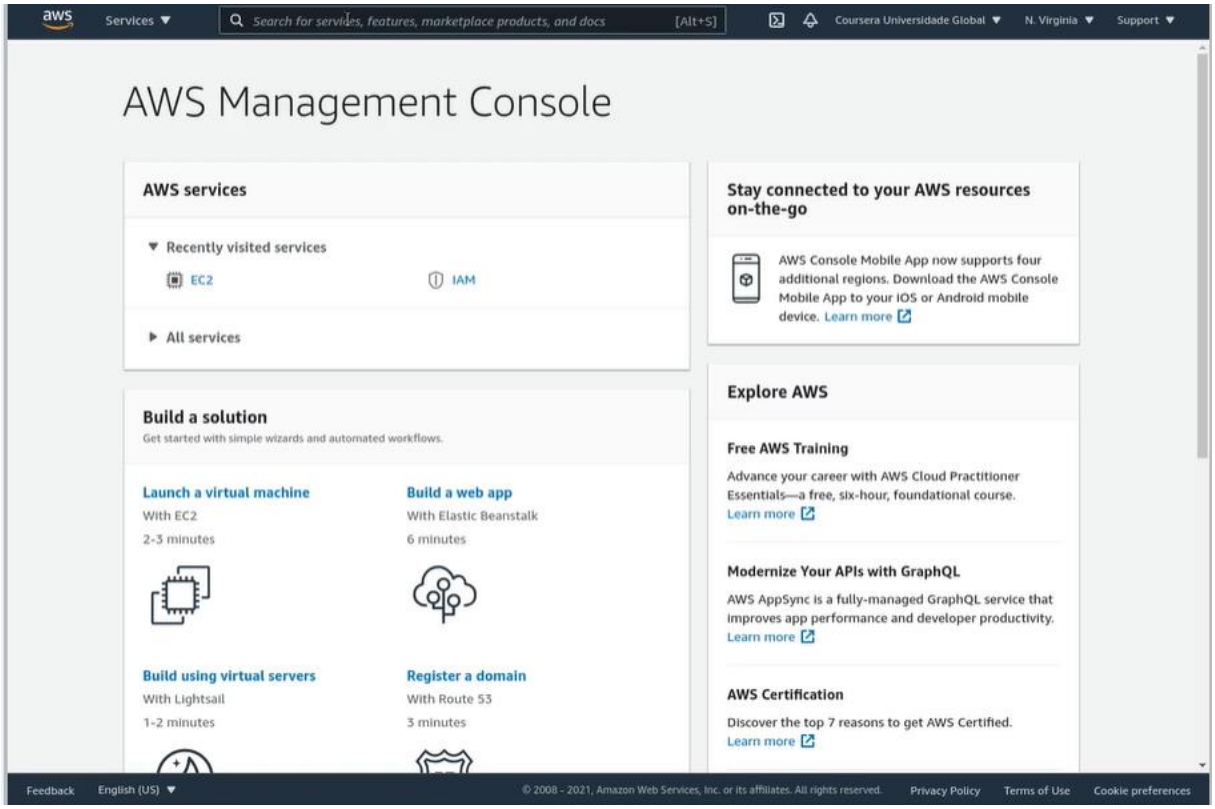


Finally, all thing is configured.

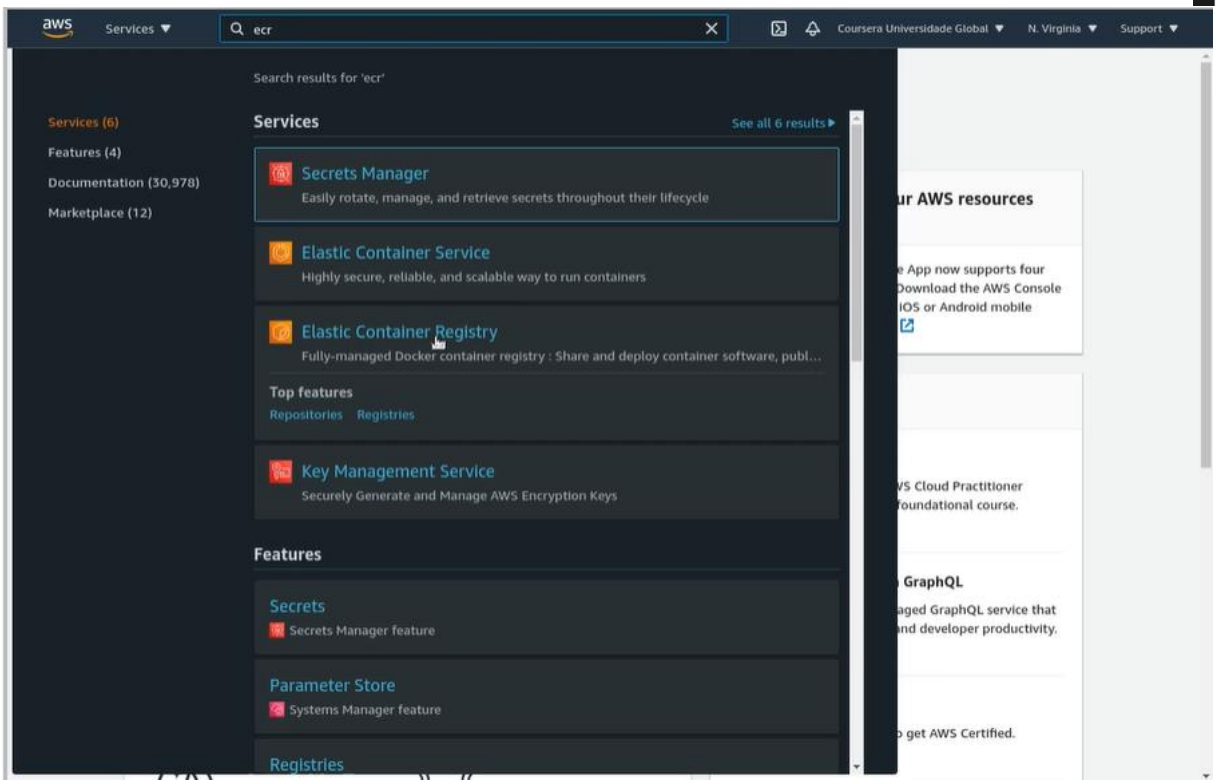
Task 3: Create an Elastic Container Registry to store our container images, and set some identities configuration need for the project

Elastic Container Registry (ECR) is the place where we can create our major repositories I public or private mode. In this project we will used to store our web application container image in AWS console.

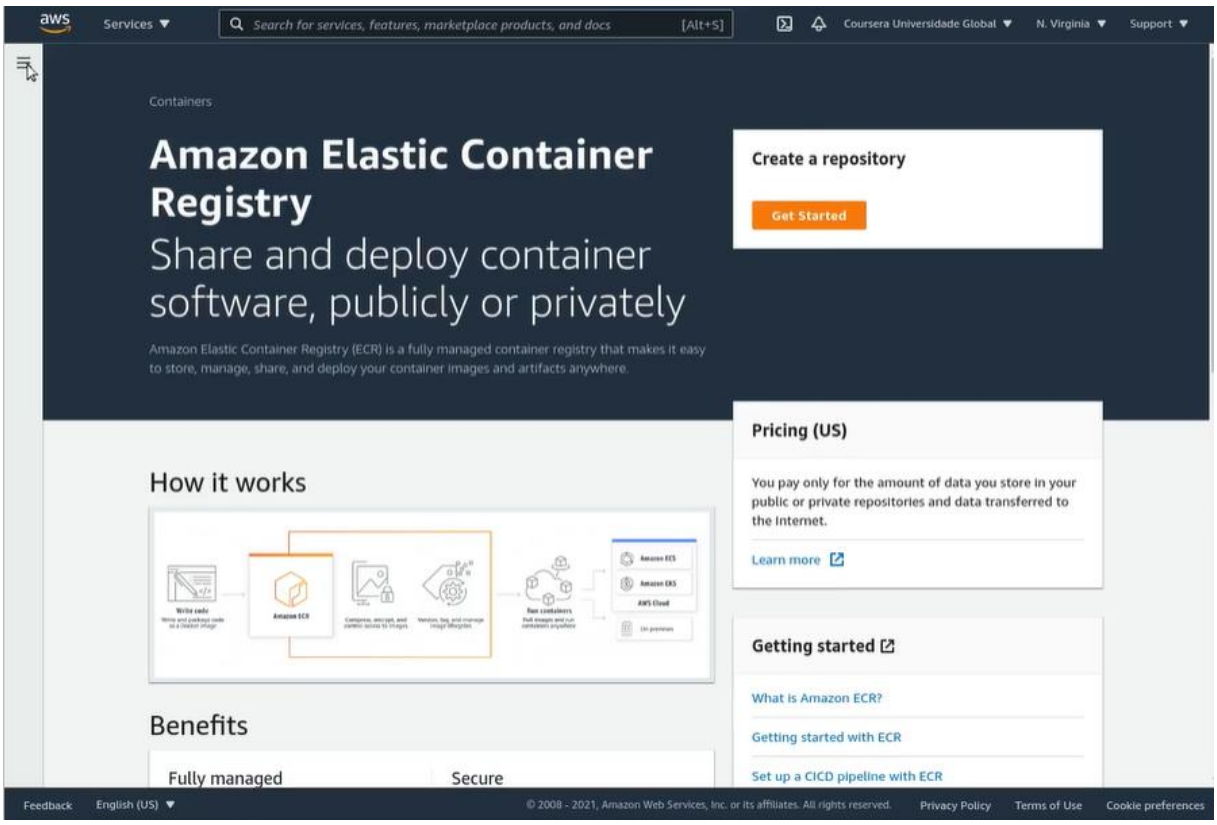
On the search bar on the top of the page of AWS console:



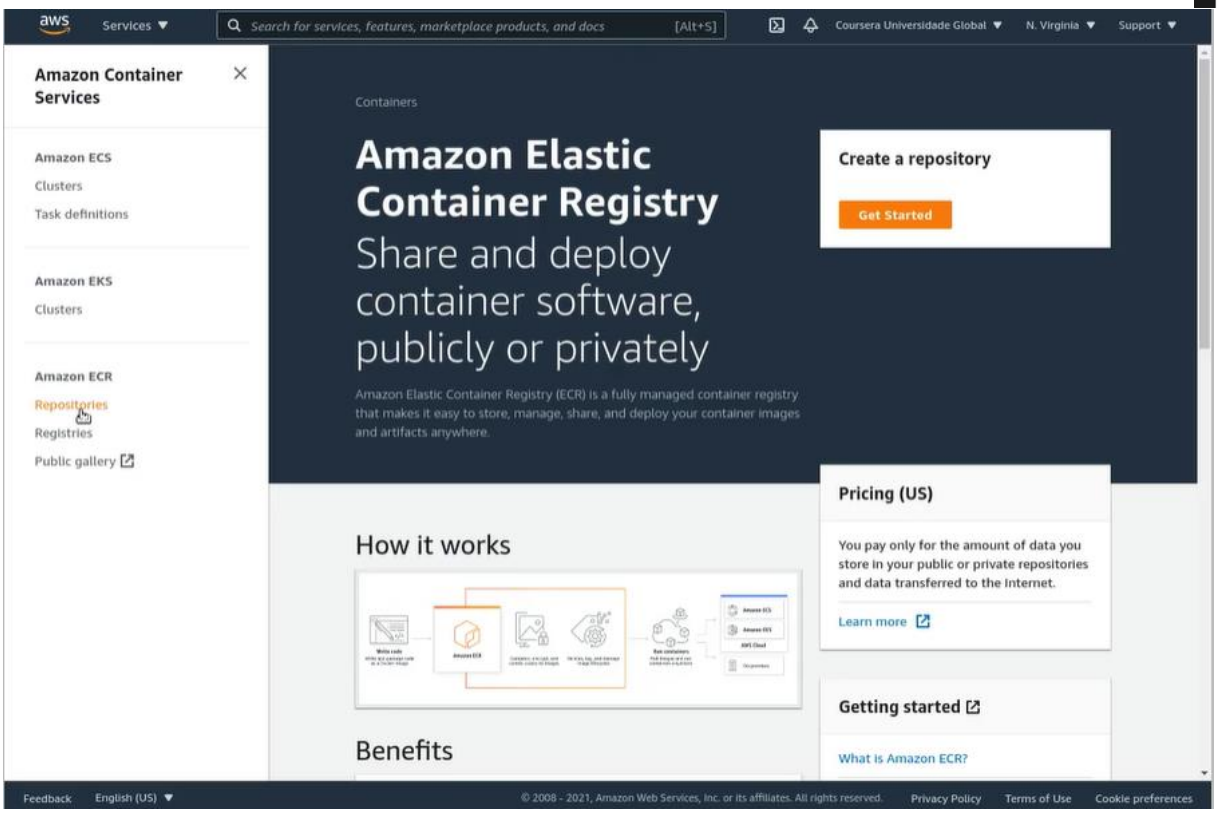
we gonna search for ECR and click on **Alaska Container history**:



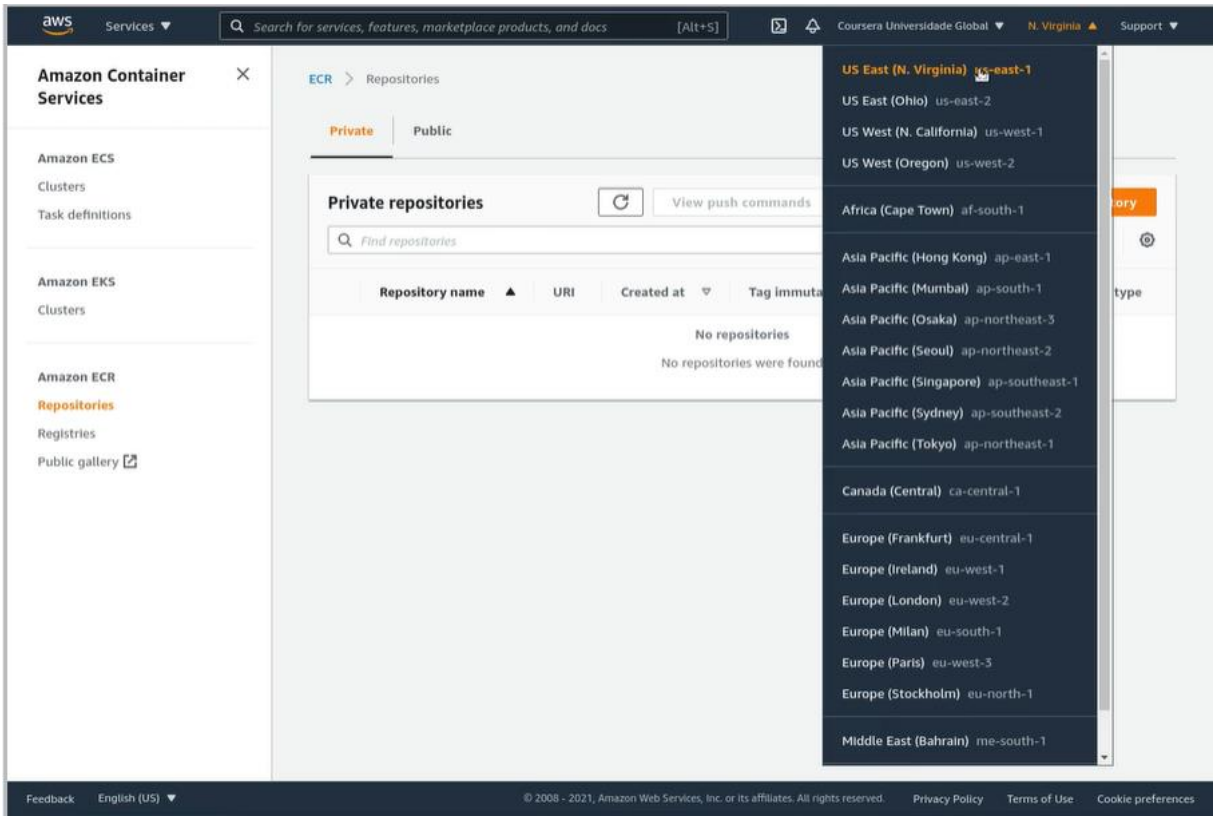
Then, we got this page:



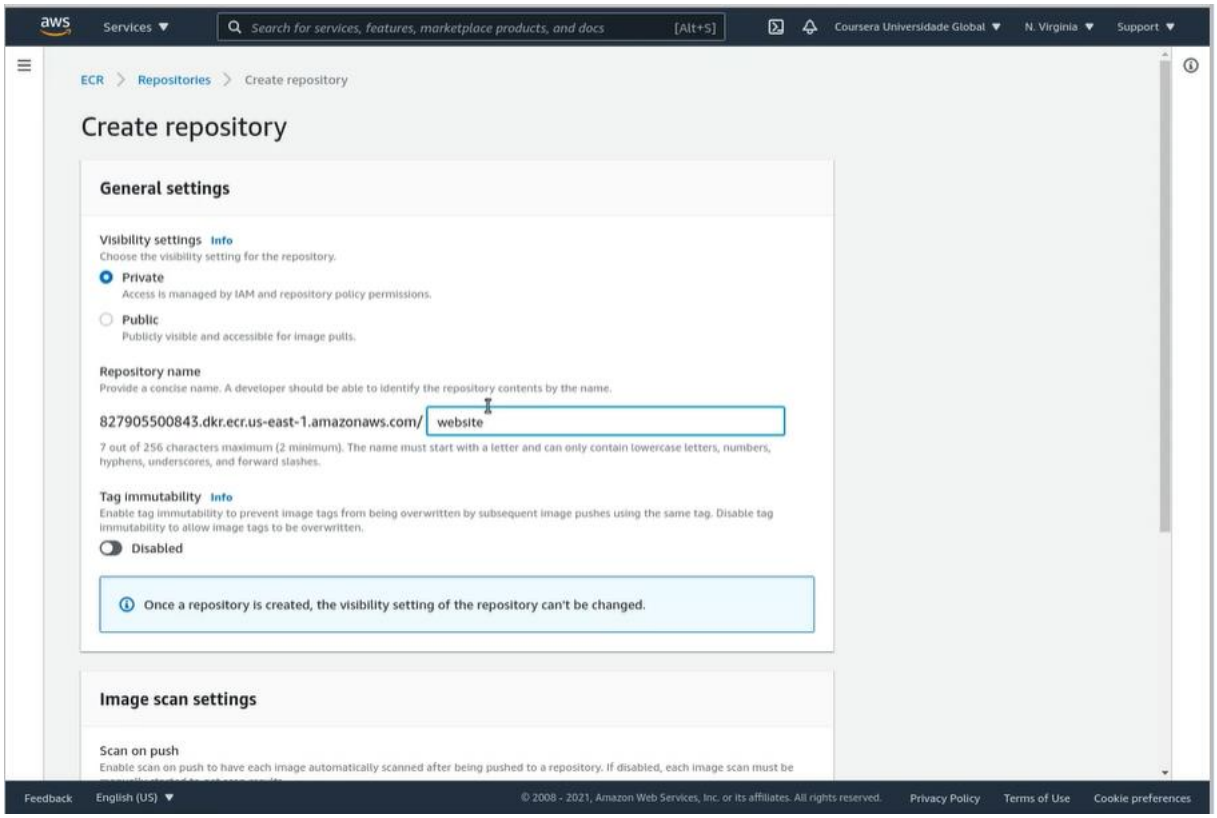
Then now in this page, we go to the left menu, and below Amazon ECR, we click on **Repositories:**



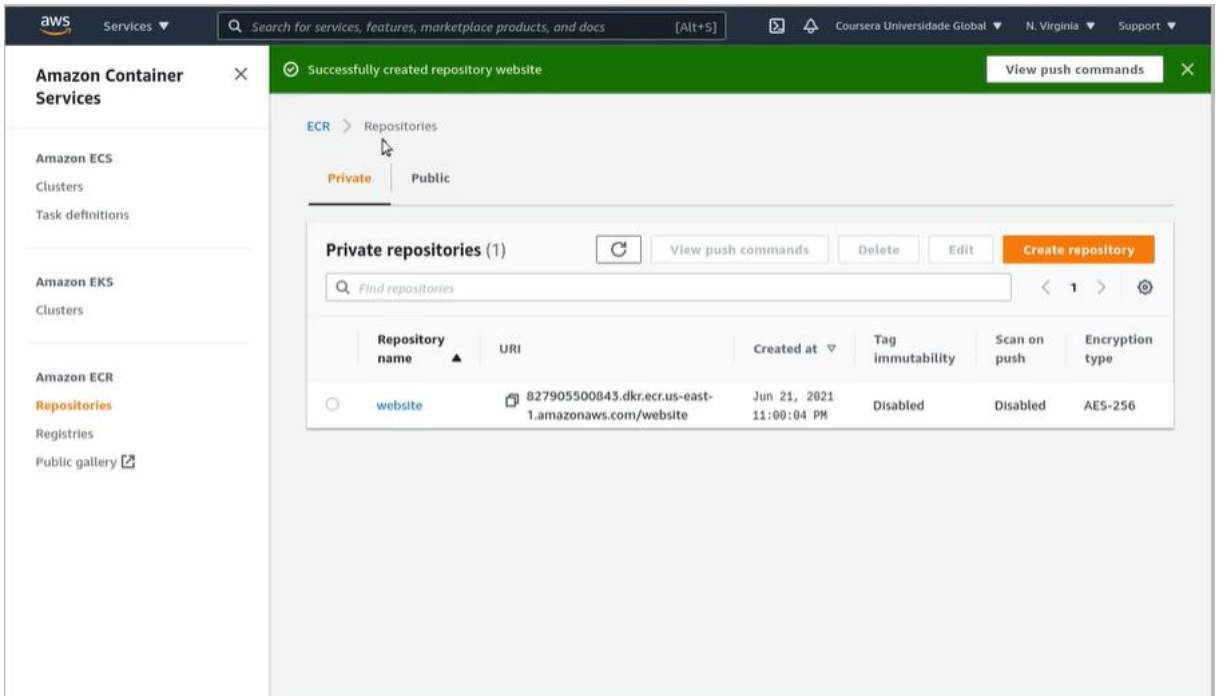
We check that we are in the AWS region we are choosing for this project:



On clicking on **'Create Repository',** we got this page below, And there we add the name of our repository, and make the visibility settings to **private:**

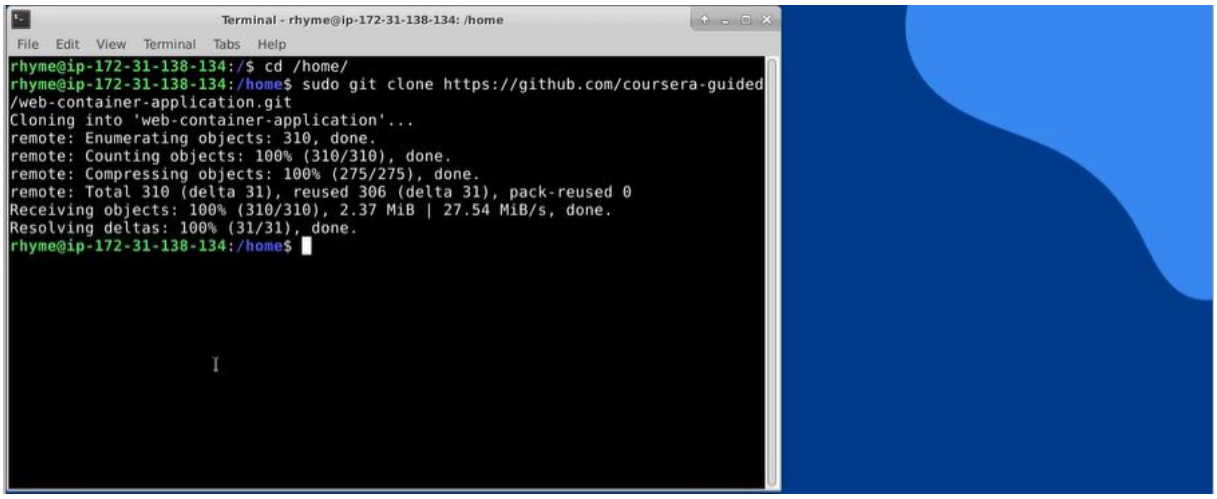


After, we click on 'Create Repository' button, and then the repository was created successfully:



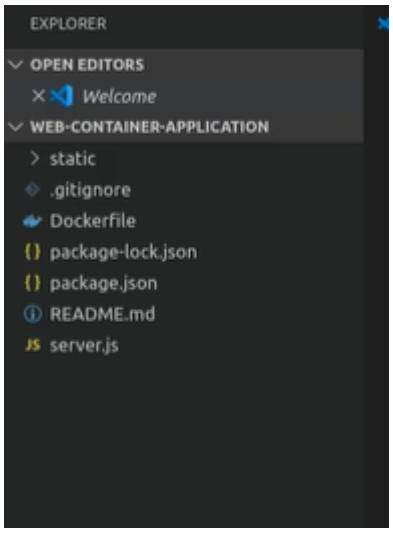
Task 4: Clone an application from GitHub, build a Docker image, run a container and push the image to Elastic Container Registry

We need to create a container image for our web application, we will clone the application files and GitHub in the cloud desktop

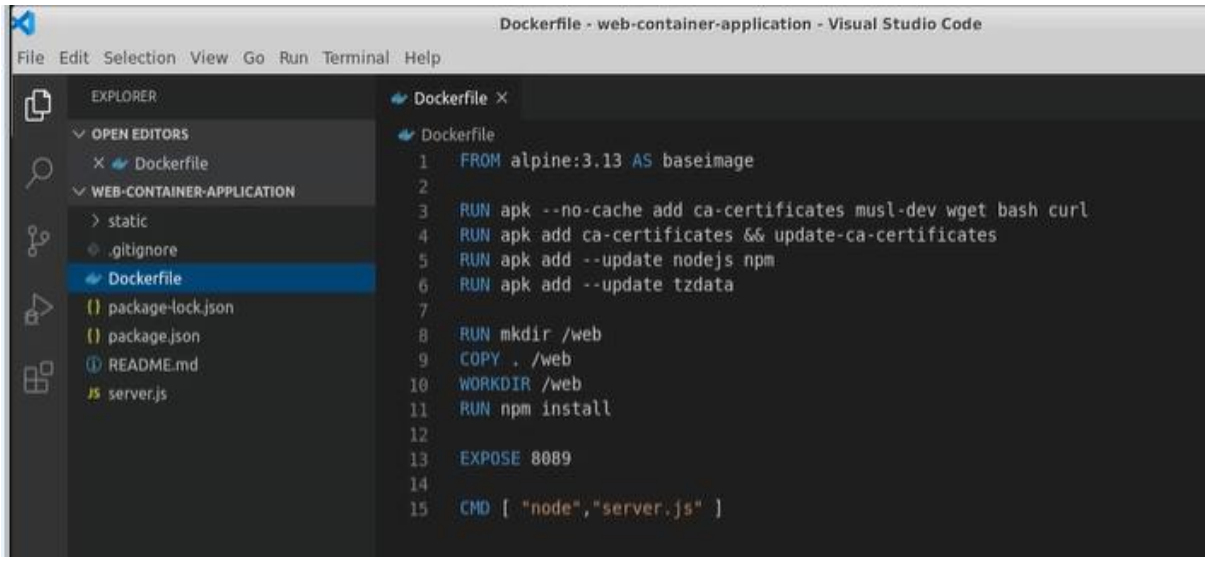


And on typing ls command, we can see that we have a web dash container dash application:

To explore the application, we gonna open it with an IDE, and then we see that It's a NodeJS JavaScript application



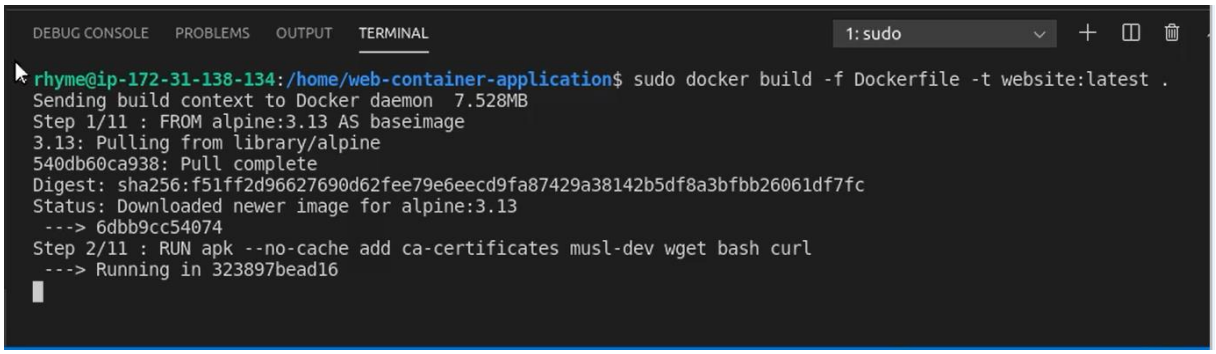
The application has a Docker file, it is the manifest file to create and build our container image. Let's see its content:



It's a very simple image where NodeJS is installed in our application is exposed on the port 8089.

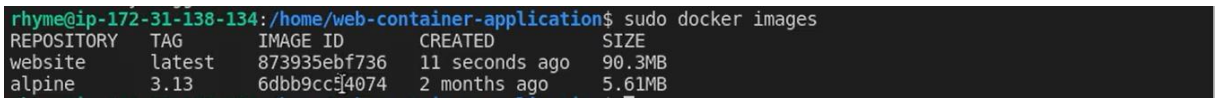
Now, let's build our image:

#sudo docker build -f Dockerfile -t website:latest .



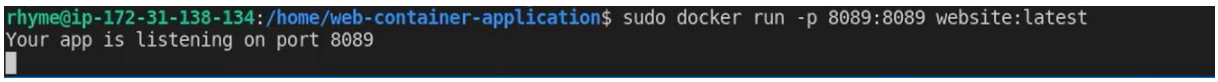
And to see our list of images, we can type:

#sudo docker images

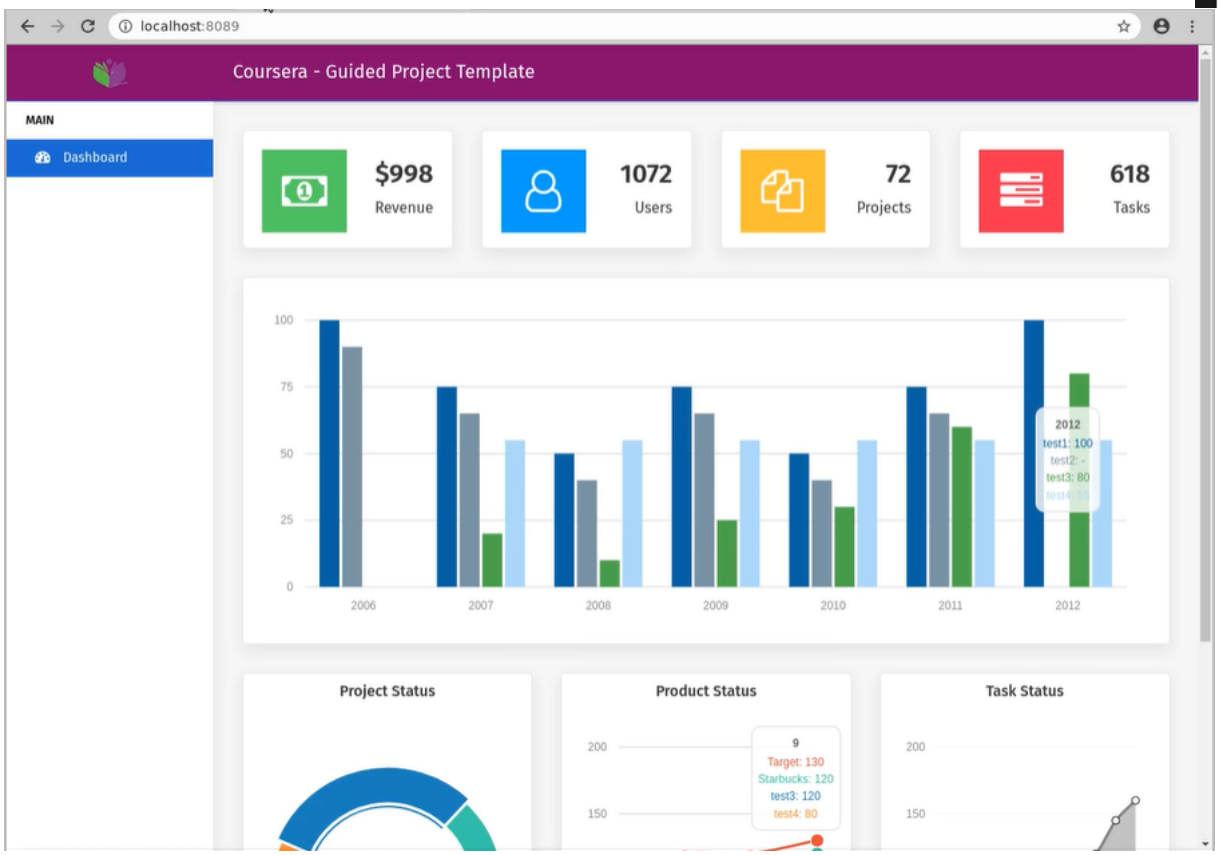


Then now, to run and validate image, we gonna use the command:

#sudo docker run -p 8089:8089 website:latest



Now we can see our application on the browser:



On typing the command: #sudo docker ps, we can see that the container is running:

```
rhyme@ip-172-31-138-134:/home/web-container-application$ sudo docker ps
CONTAINER ID   IMAGE            COMMAND            CREATED          STATUS          PORTS                    N
AMES
8528bc57f20c   website:latest   "node server.js"   53 seconds ago   Up 51 seconds   0.0.0.0:8089->8089/tcp   c
onfident_feynman
```

To stop the container, we use the command:

#sudo docker stop

```
rhyme@ip-172-31-138-134:/home/web-container-application$ sudo docker stop 8528bc57f20c
8528bc57f20c
```

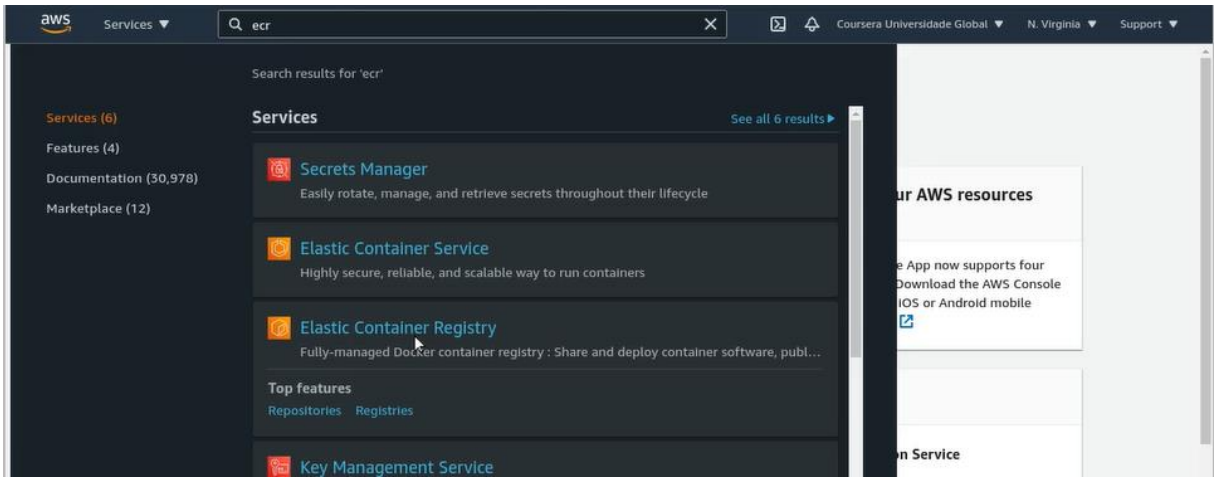
And we can check with:

```
rhyme@ip-172-31-138-134:/home/web-container-application$ sudo docker ps
CONTAINER ID   IMAGE     COMMAND   CREATED   STATUS    PORTS     NAMES
```

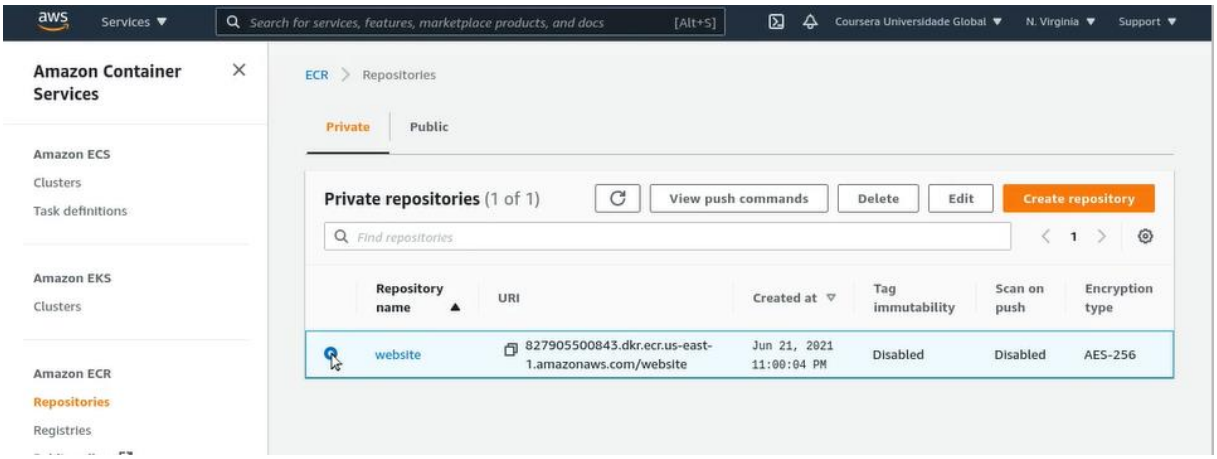
Like we can see, we don't have any container running.

Now let's push our image to our elastic container registry. Pushing the image to the last container registry repository allow us to use this image in the last Kubernetes service.

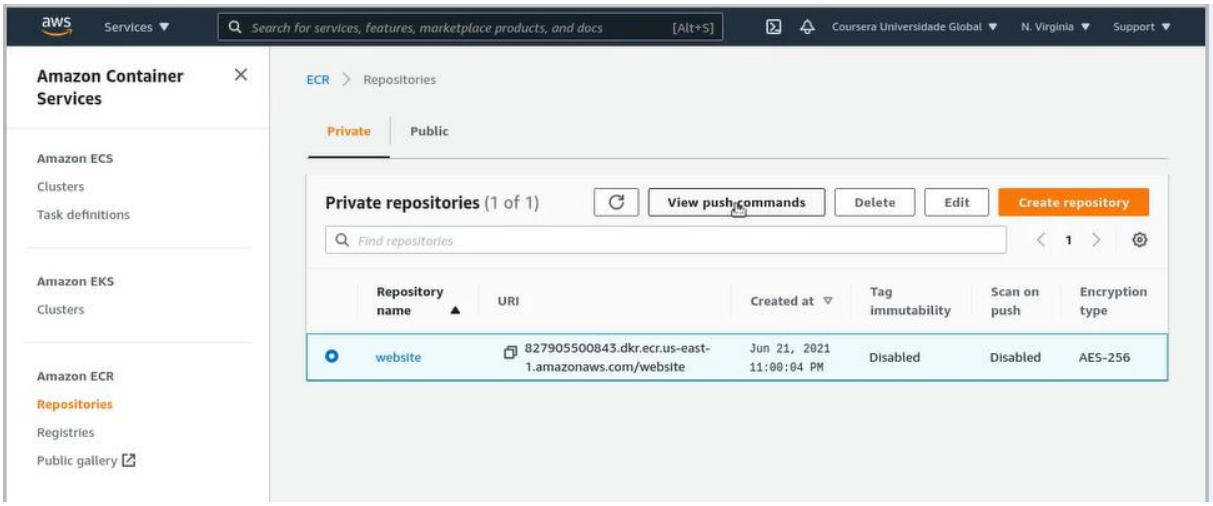
We go to the Elastic container registry repositories:



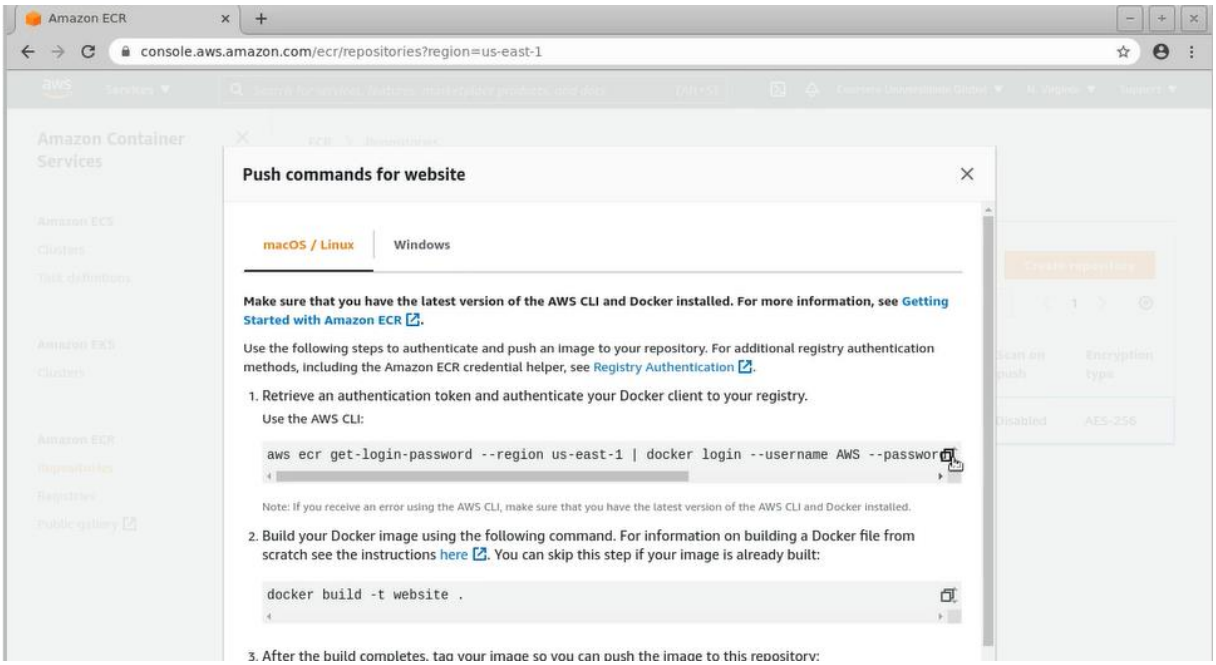
Later, we select our project called 'website'



And we click on '**View push commands':**



Then, we copy the first command in this page:



And now we paste it in the terminal of our IDE, with adding the term ‘sudo’ before the docker command:

```
rhyme@ip-172-31-138-134:/home/web-container-application$ aws ecr get-login-password --region us-east-1 |sudo
docker login --username AWS --password-stdin 827905500843.dkr.ecr.us-east-1.amazonaws.com
WARNING! Your password will be stored unencrypted in /home/rhyme/.docker/config.json.
Configure a credential helper to remove this warning. See
https://docs.docker.com/engine/reference/commandline/login/#credentials-store

Login Succeeded
```

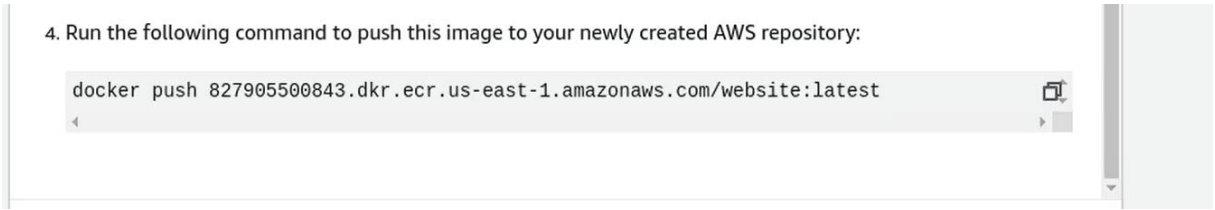
We don’t need to execute the second command because we are ready that not a few steps behind, we gonna copy the third command and paste it, without forgot to prefix the docker command with ‘sudo’:

```
rhyme@ip-172-31-138-134:/home/web-container-application$ sudo docker tag website:latest 827905500843.dkr.ecr.u
s-east-1.amazonaws.com/website:latest
rhyme@ip-172-31-138-134:/home/web-container-application$
```

We can see we have our images in the new tech for our image.

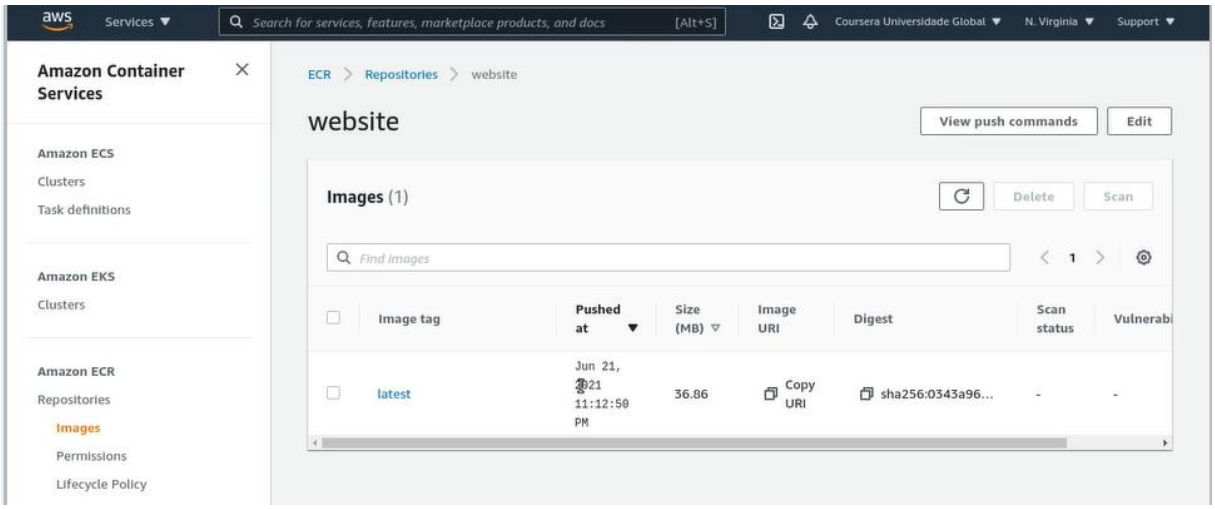
```
rhyme@ip-172-31-138-134:/home/web-container-application$ sudo docker images
REPOSITORY                                                TAG       IMAGE ID       CREATED          SIZE
827905500843.dkr.ec[illegible].us-east-1.amazonaws.com/website   latest    873935ebf736   5 minutes ago    90.3MB
website                                                   latest    873935ebf736   5 minutes ago    90.3MB
alpine                                                    3.13      6dbb9cc54074   2 months ago     5.61MB
```

And finally we gonna a copy the latest command:



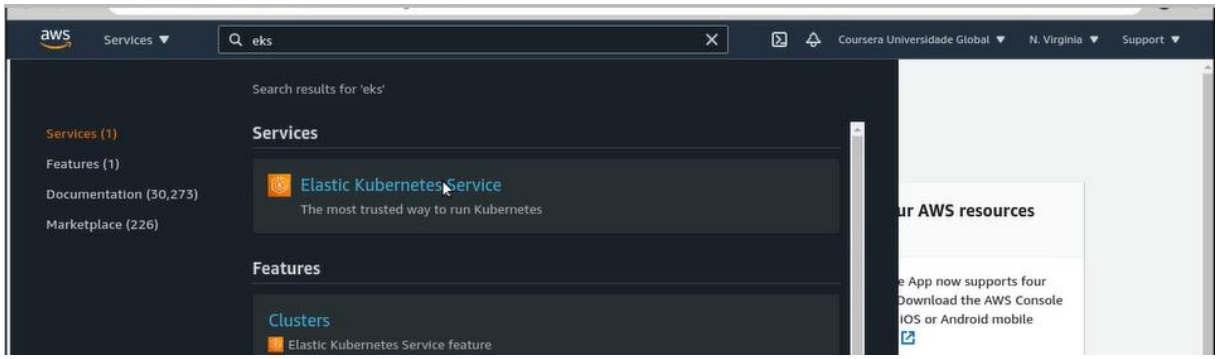
```
rhyme@ip-172-31-138-134:/home/web-container-application$ sudo docker push 827905500843.dkr.ecr.us-east-1.amazo
naws.com/website:latest
The push refers to repository [827905500843.dkr.ecr.us-east-1.amazonaws.com/website]
73237bfd4927: Pushed
d3b1a41dc061: Pushed
8b0de1cb6102: Pushed
d788eef9a8c7: Pushed
fb525caa4887: Pushed
219d41de9e7e: Pushed
7a94da318d8a: Pushed
b2d5eeeaba3a: Pushed
latest: digest: sha256:0343a96f66b05864a6149016c61e4031d3118dbcfe1c82e941b7f2b135c03a0c size: 2001
```

Then our image will be pushed to the ECR repository. And like we see our image is here:

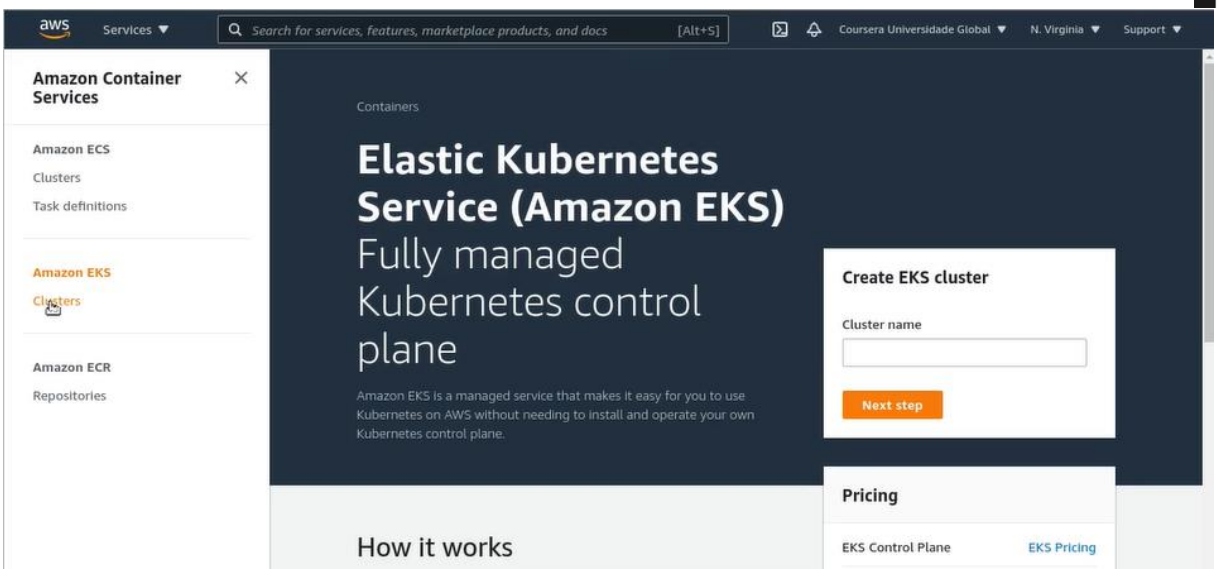


Task 5: Create an Elastic Kubernetes Cluster and set all configuration needed

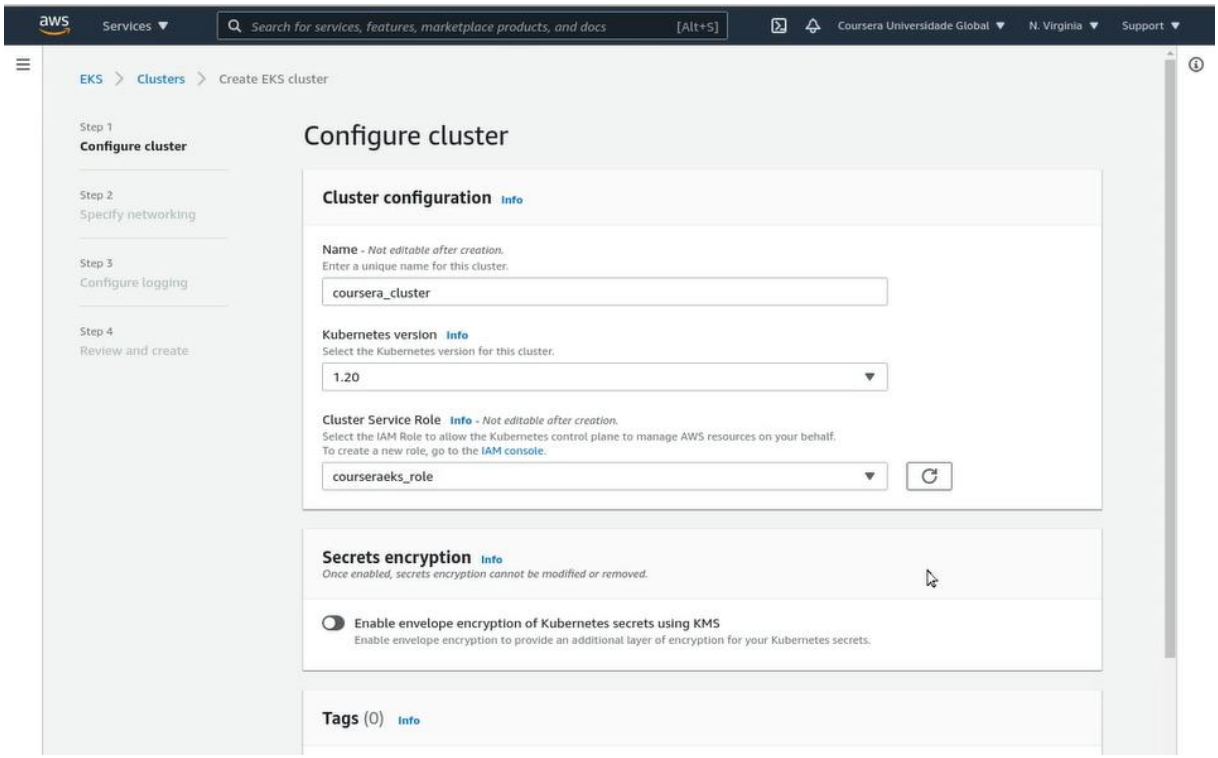
We have the AWS console opened, we will to create our Kubernetes cluster on AWS. So, firstly we will to search for EKS and we will choose Elastic Kubernetes Service:

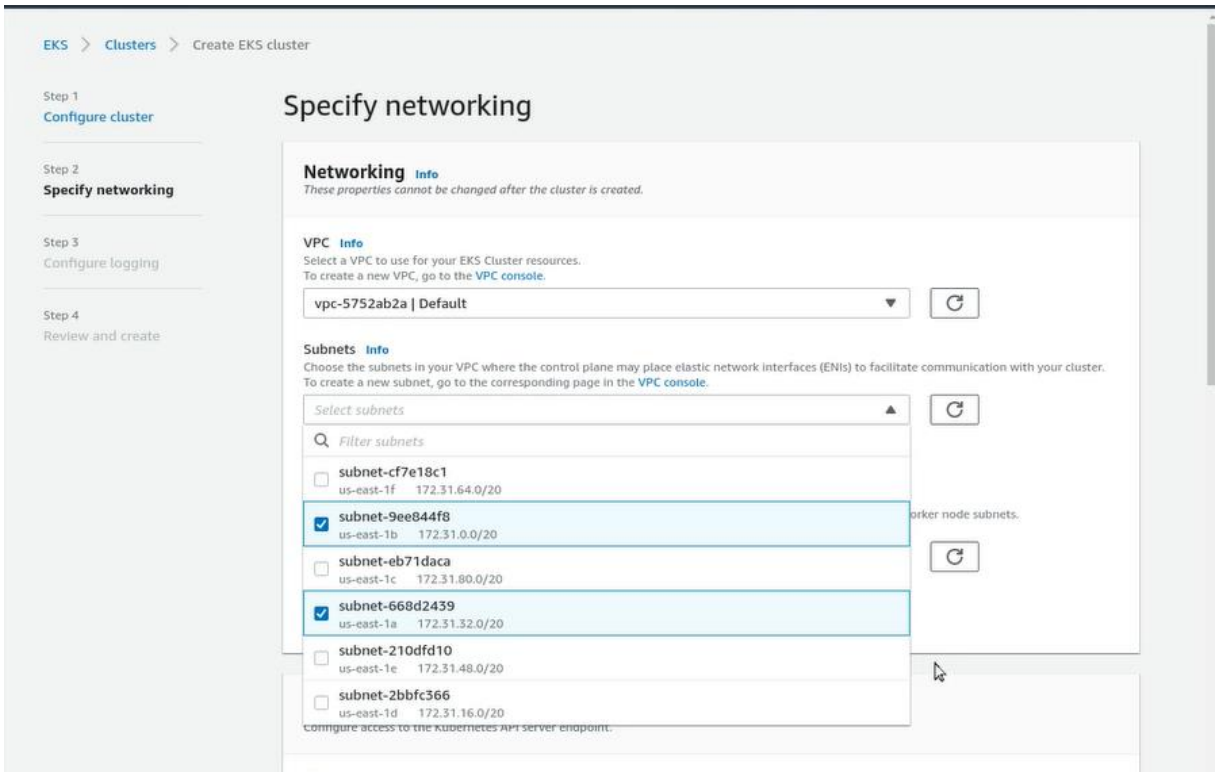


Later, in the left menu below Amazon EKS we click on **Clusters**:

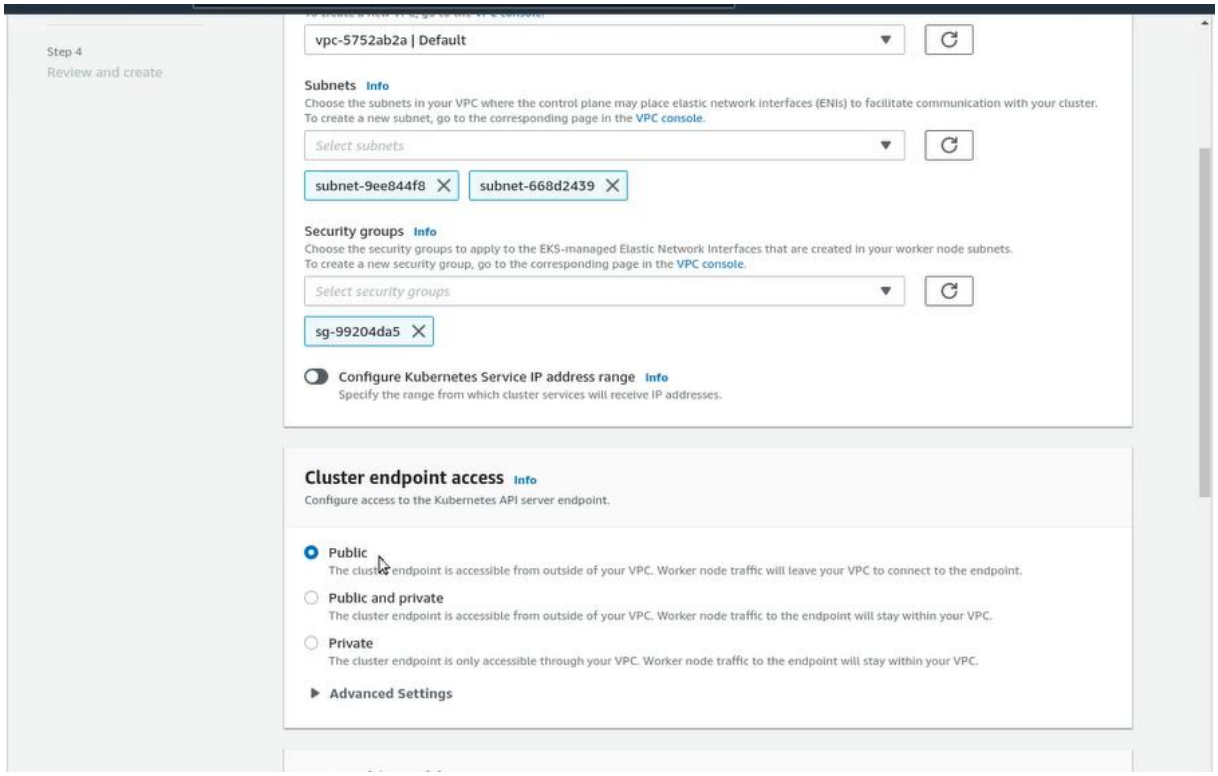


To create a cluster, we should define a name for the cluster, the Kubernetes version and the cluster role:

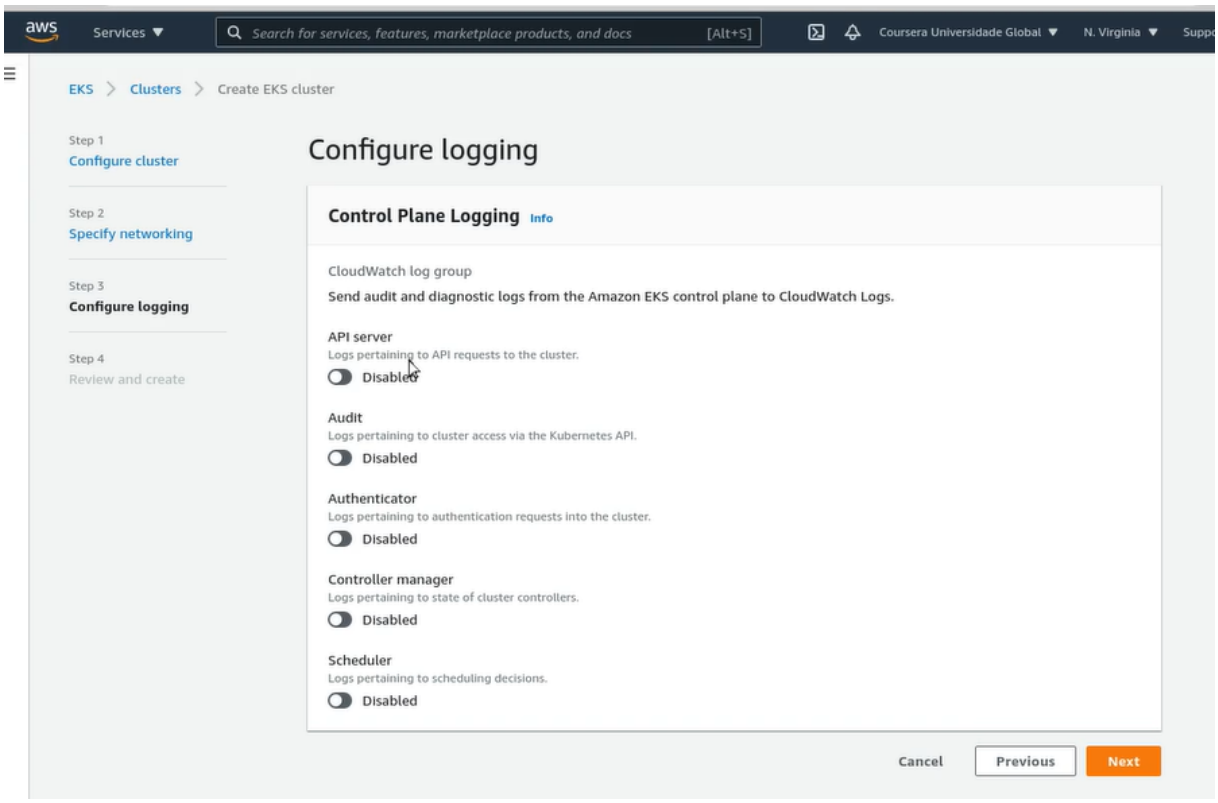




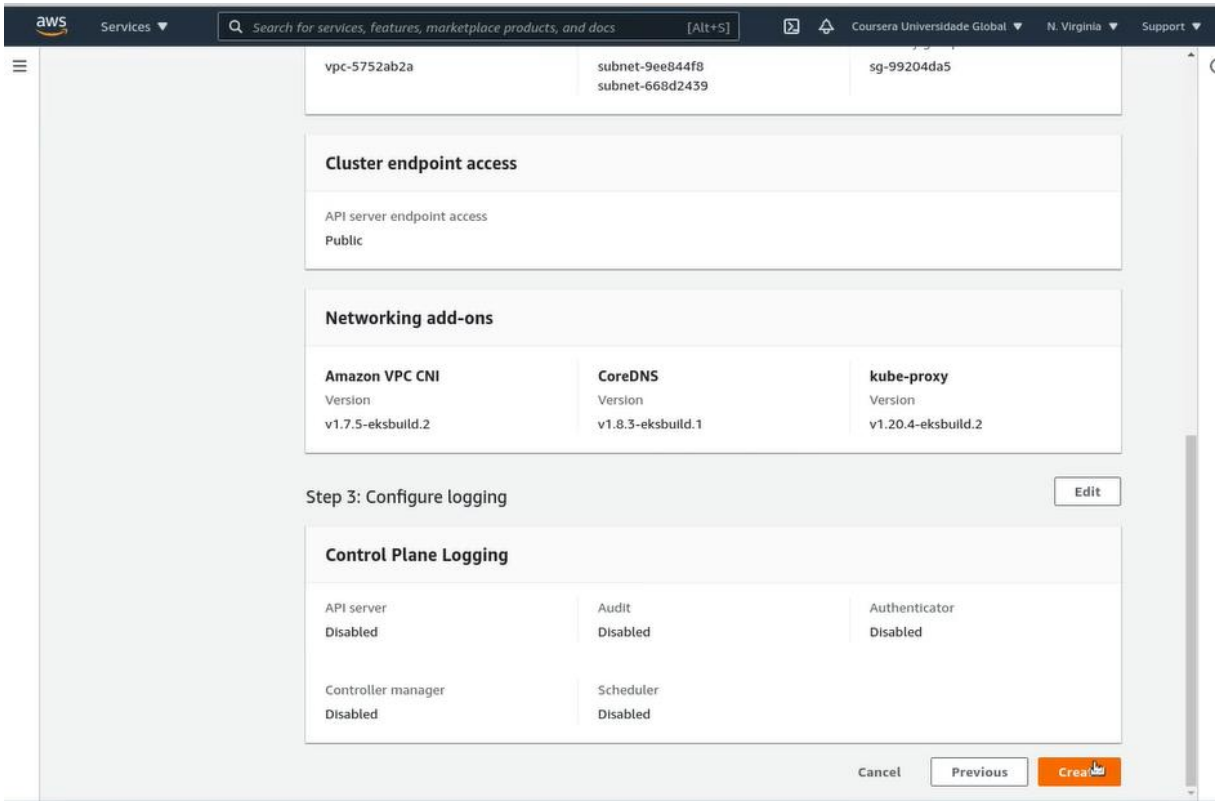
And we select the public cluster endpoint, and we keep the other configurations as is:



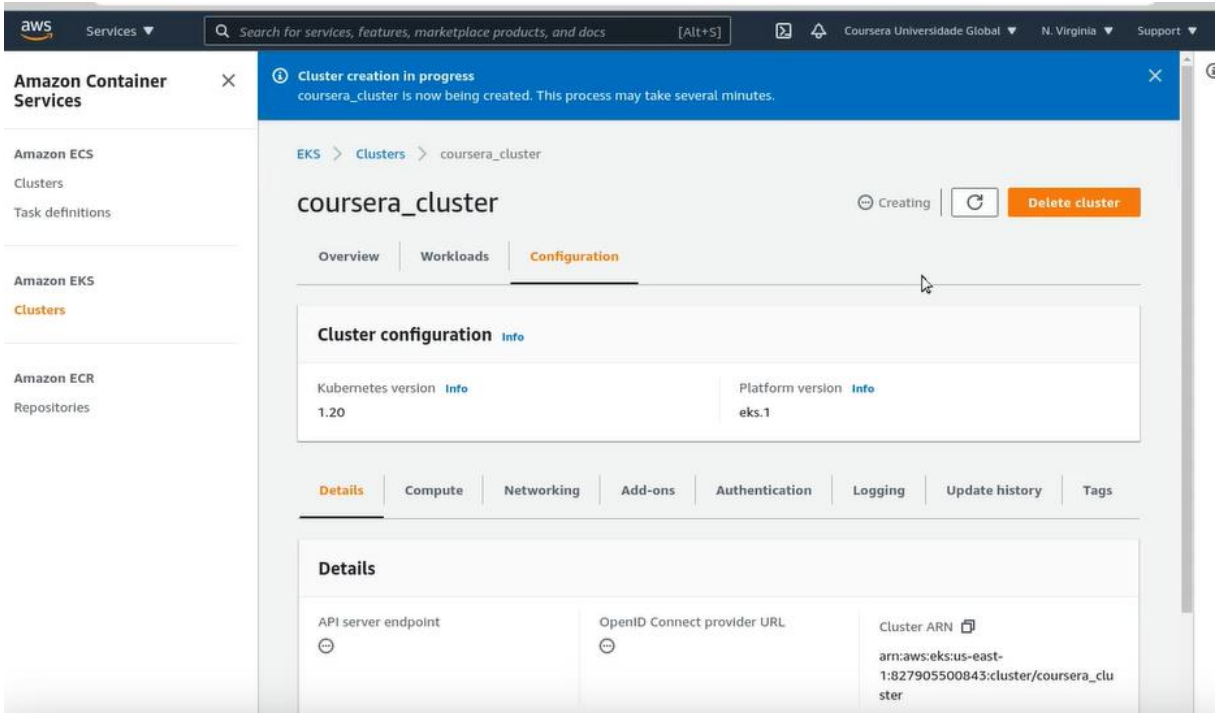
, and we click -> **Next:**



And -> **Next**



And -> **Create**

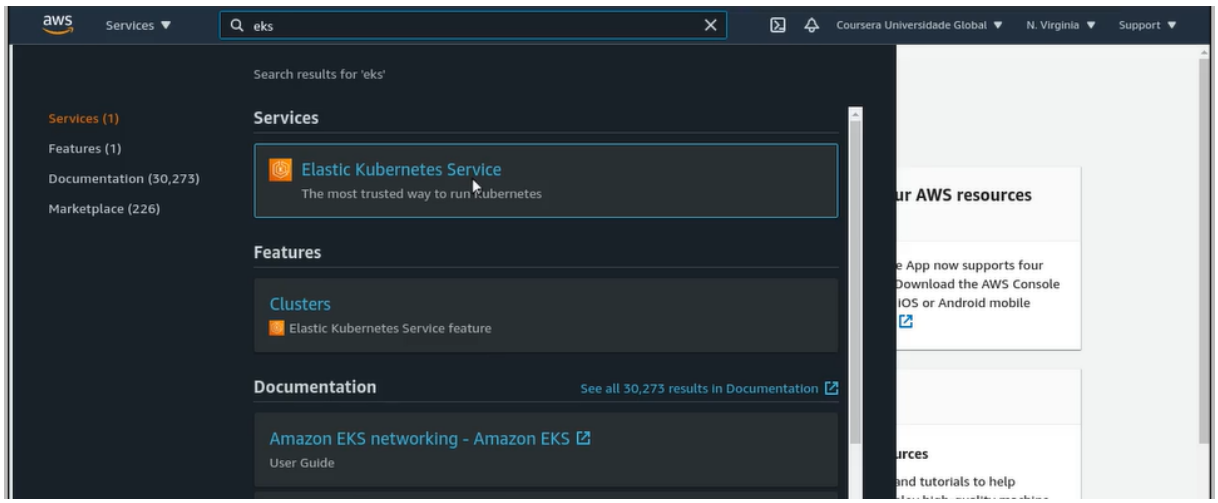


The cluster will be created and it will take several minutes, generally 15-20minutes

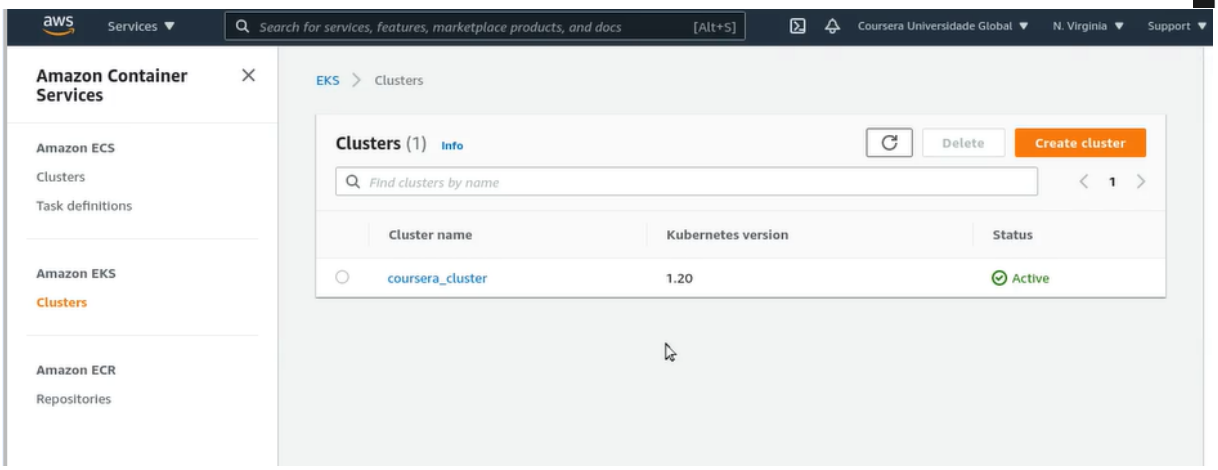
Task 6: Create and set a Node Group to host containers in a POD architecture

In this task we will create our load group which is the group of virtual machines that will be part of our cluster and where our containers will run.

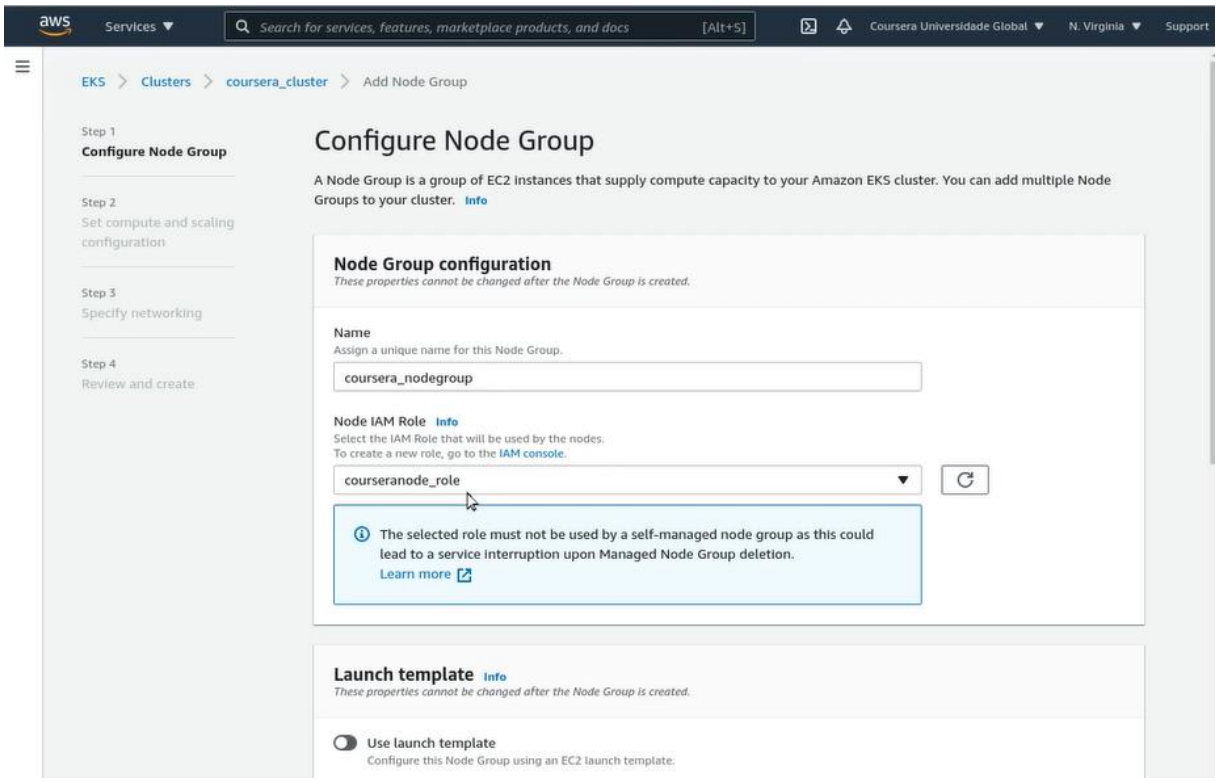
Firstly, we gonna check if our clusters was created and if is active. In the AWS console, we go to the search bar on to top and we type EKS and we click to Elastic Kubernetes Service:



And click on **-> Clusters**

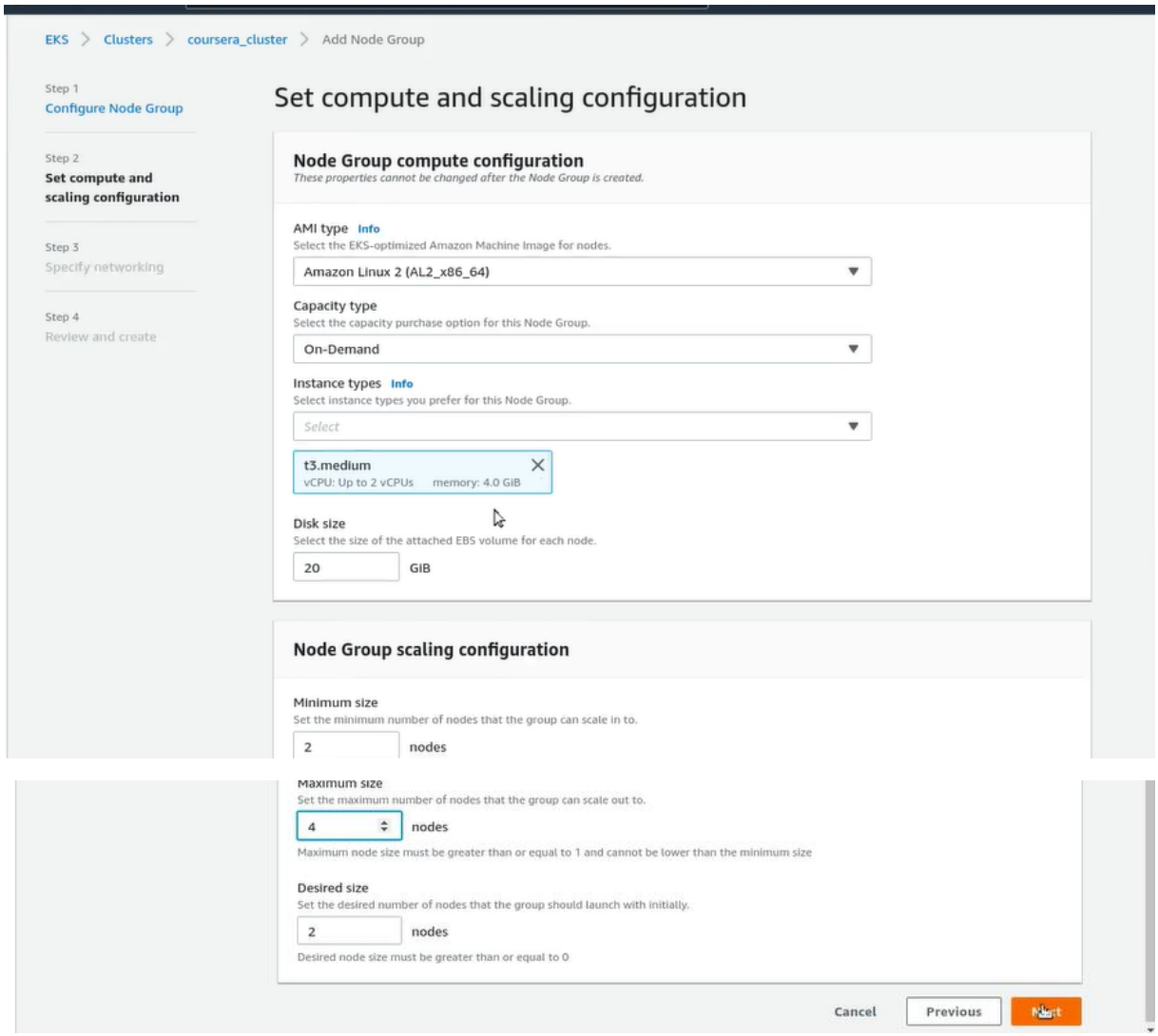


In this page above, we see the status of the cluster which we created in the previous task is active, it means our cluster is running. So, we click on the cluster name to open the cluster:

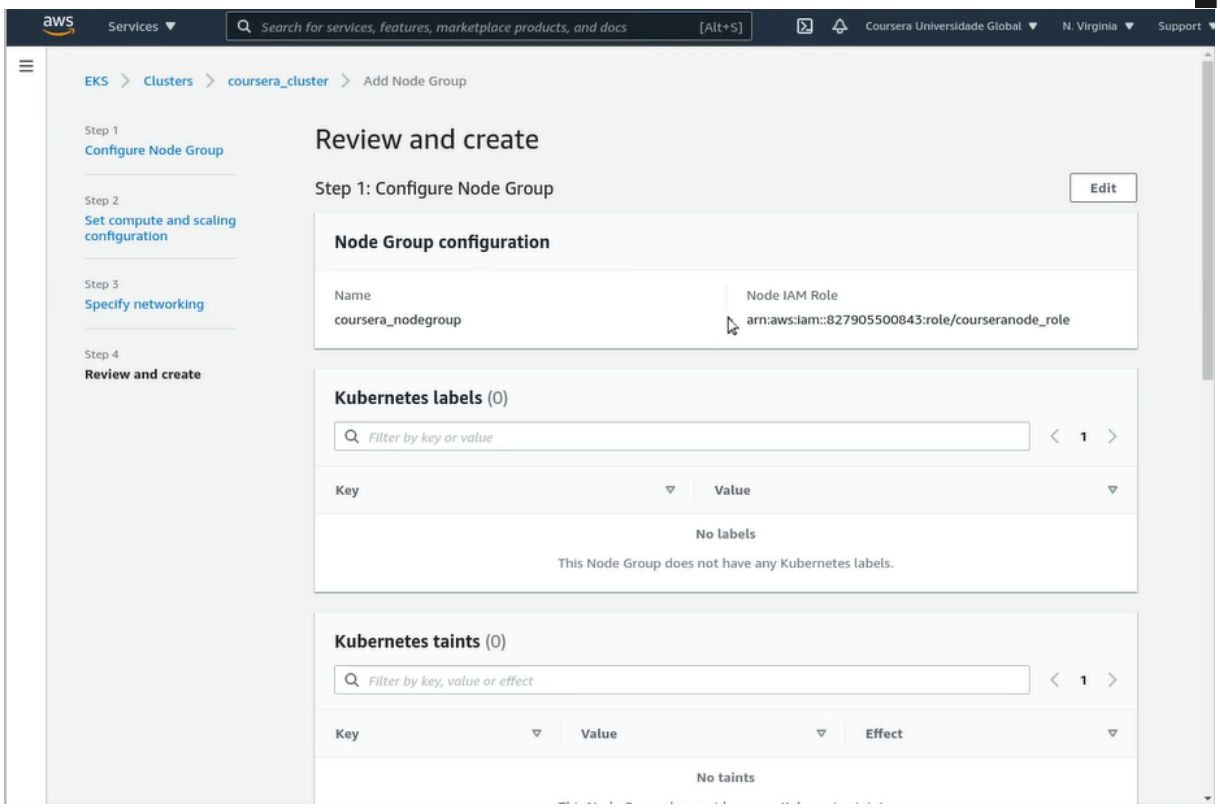


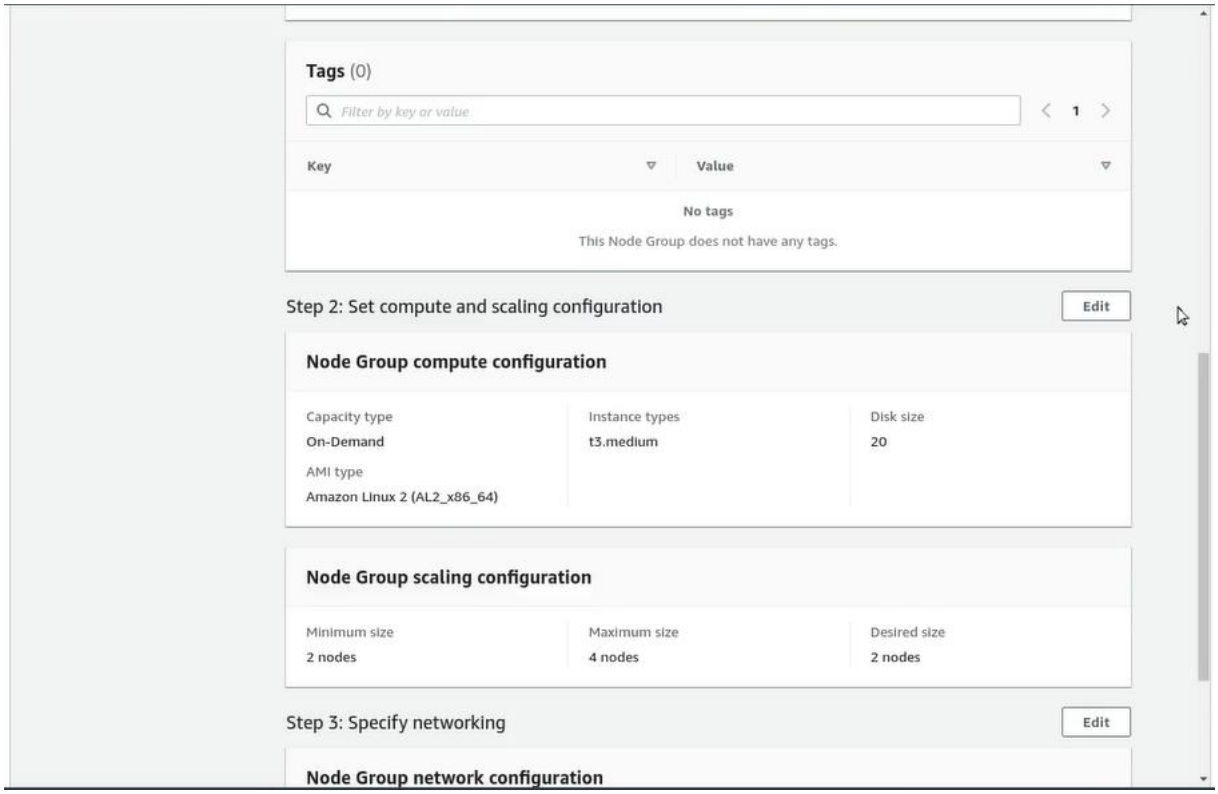
And we click on -> **Next**

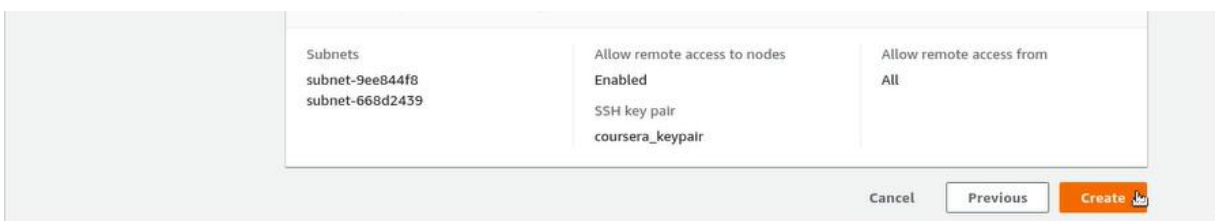
And we fill the configuration:



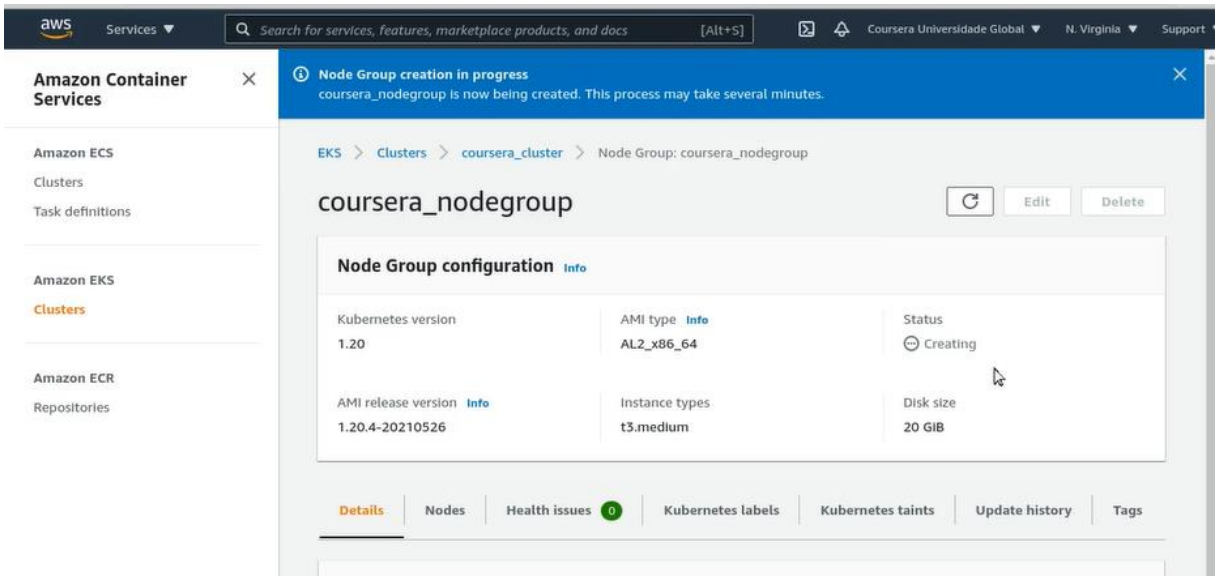
And we click on -> **Next**


aws
Services
Search for services, features, marketplace products, and docs [Alt+S]
Coursera Universidade Global
N. Virginia
Support
EKS > Clusters > coursera_cluster > Add Node Group
Step 1
Configure Node Group
Step 2
Set compute and scaling configuration
Step 3
Specify networking
Step 4
Review and create
Review and create
Step 1: Configure Node Group
Edit
Node Group configuration
Name
coursera_nodegroup
Node IAM Role
arn:aws:iam::827905500843:role/courseranode_role
Kubernetes labels (0)
Filter by key or value
1
Key
Value
No labels
This Node Group does not have any Kubernetes labels.
Kubernetes taints (0)
Filter by key, value or effect
1
Key
Value
Effect
No taints


Tags (0)
Filter by key or value
1
Key
Value
No tags
This Node Group does not have any tags.
Step 2: Set compute and scaling configuration
Edit
Node Group compute configuration
Capacity type
On-Demand
Instance types
t3.medium
Disk size
20
AMI type
Amazon Linux 2 (AL2_x86_64)
Node Group scaling configuration
Minimum size
2 nodes
Maximum size
4 nodes
Desired size
2 nodes
Step 3: Specify networking
Edit
Node Group network configuration


Subnets
subnet-9ee844f8
subnet-668d2439
Allow remote access to nodes
Enabled
SSH key pair
coursera_keypair
Allow remote access from
All
Cancel
Previous
Create

And later, we click on **-> Create**



Then, the virtual machines and his sources will be created.

Task 7: Deploy an application in Elastic Kubernetes Cluster using an Image hosted in Elastic Container Registry

In this task we will deploy the web application we have built in task four and push it to Elastic Container Registry into our Elastic Kubernetes Cluster. To do that we will use a deployment

Let's first set the cube conflict credentials to connect in our cluster by **kubectl**

```
rhyme@ip-172-31-138-134:/$ aws eks --region us-east-1 update-kubeconfig --name c
oursera_cluster
Added new context arn:aws:eks:us-east-1:827905500843:cluster/coursera_cluster to
 /home/rhyme/.kube/config
rhyme@ip-172-31-138-134:/$
```

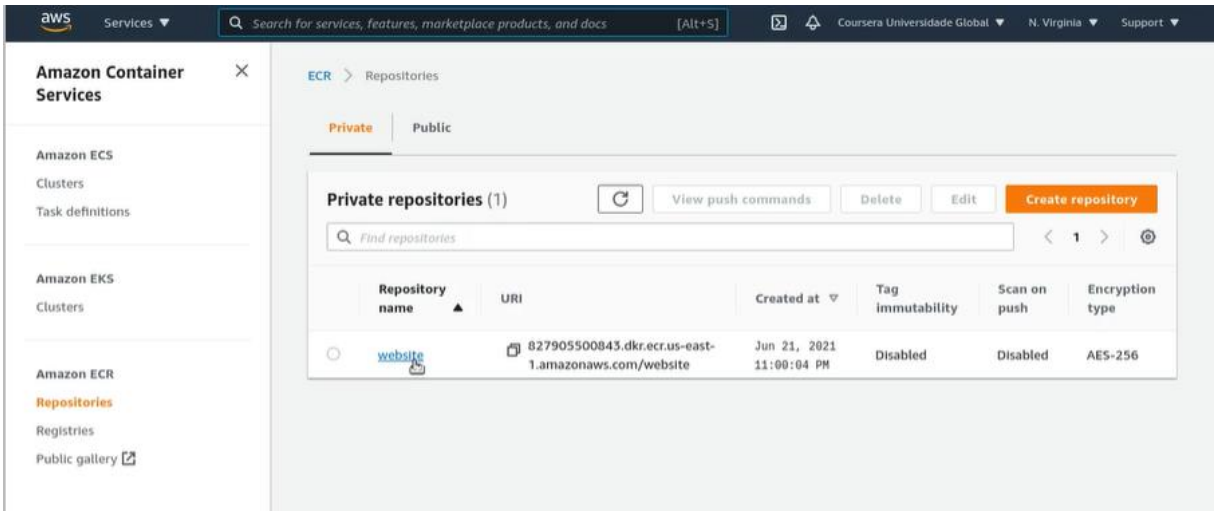
We can see if the credentials are working by typing **kubectl:**

```
rhyme@ip-172-31-138-134:/$ kubectl get nodes
NAME                           STATUS   ROLES    AGE     VERSION
ip-172-31-13-155.ec2.internal   Ready    <none>   7m26s   v1.20.4-eks-6b7464
ip-172-31-38-179.ec2.internal   Ready    <none>   7m20s   v1.20.4-eks-6b7464
```

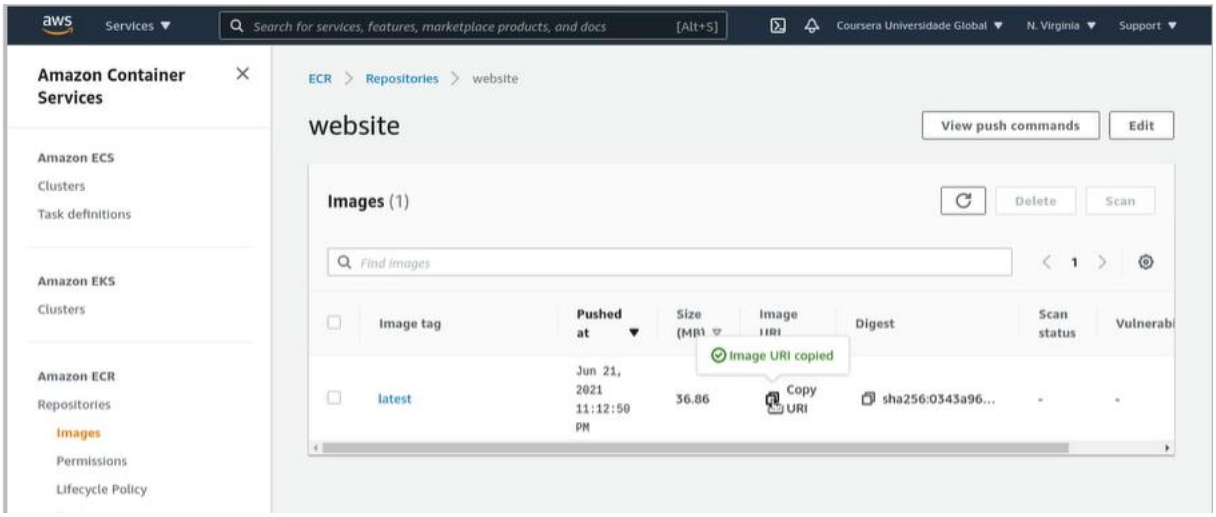
We can see our two notes are listed here.

This deployment file has two parts, the first one is deployments. Deployments will deploy our pots, our containers. So, we need to define how many replicas, it means how many containers we want to deploy. In this example we gonna define 5 replicas. And the name of our application will be ‘website’.

Now, we gone to the Elastic Container Registry on the AWS console, and we click on our website repository:

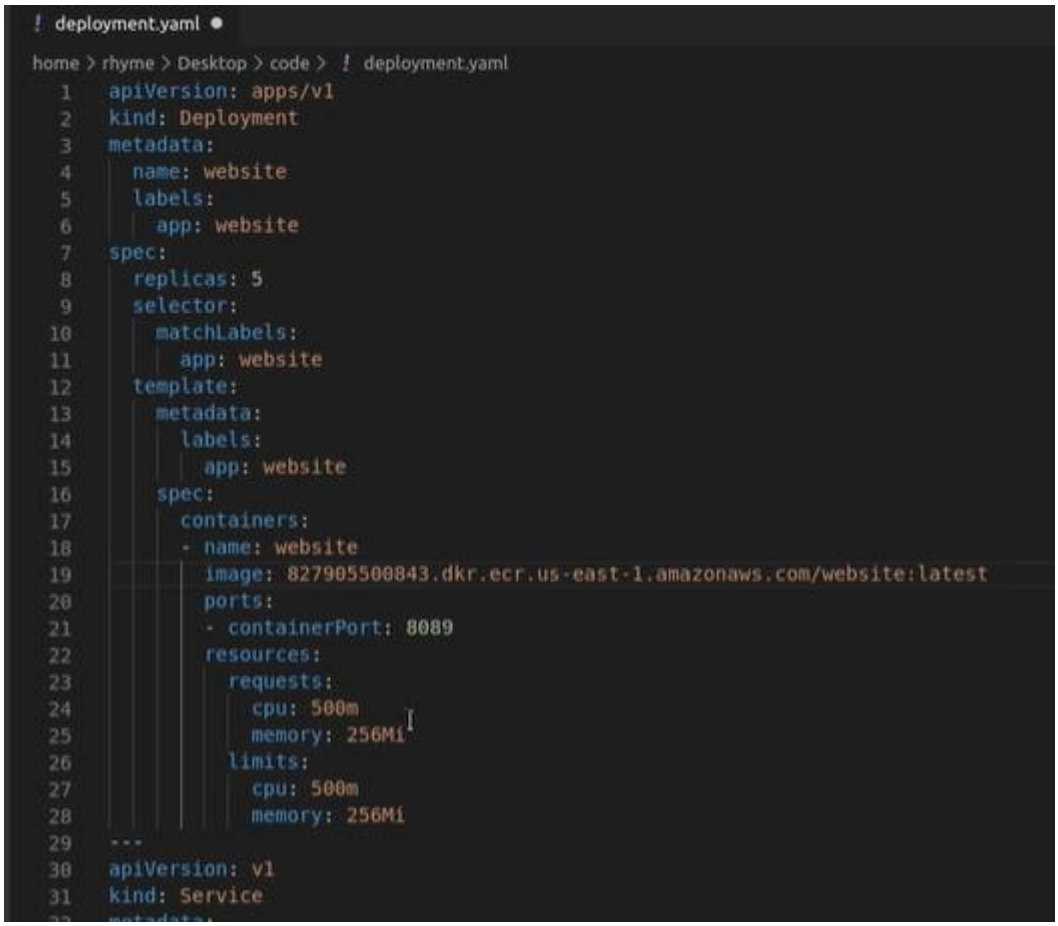


And, we copy the URI for the latest version:

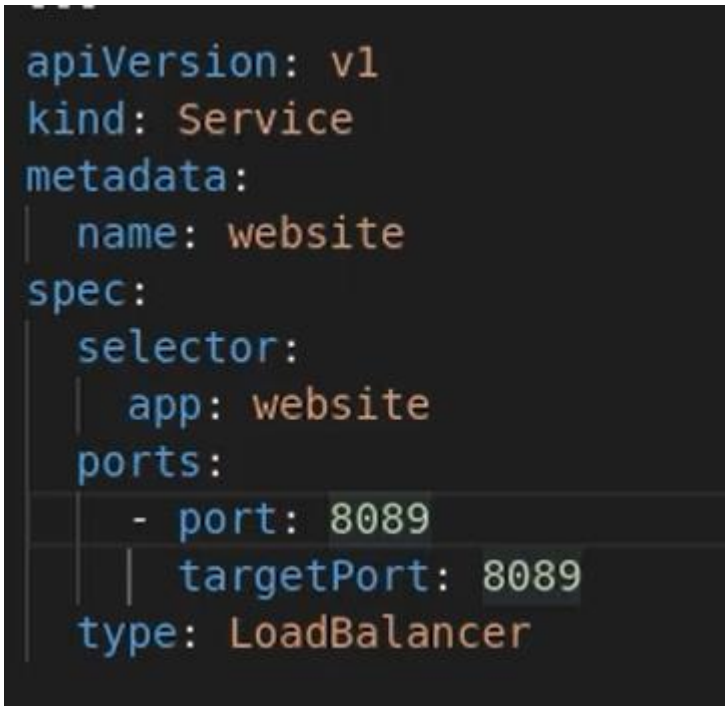


And we paste the URI in the image session of the deployment file:


deployment.yaml
home > rhyme > Desktop > code > deployment.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: website
labels:
app: website
spec:
replicas: 5
selector:
matchLabels:
app: website
template:
metadata:
labels:
app: website
spec:
containers:
- name: website
image:
ports:
resources:
requests:
cpu: 500
memory:
limits:
cpu: 500
memory: 256Mi
---
apiVersion: v1
kind: Service
metadata:
name: website
Change All Occurrences Ctrl+F2
Cut Ctrl+X
Copy Ctrl+C
Paste Ctrl+V
Command Palette... Ctrl+Shift+P



The next part of the deployment file is servicing:



Where we define services for our application like: load balancer

Now, we will save the file, and go to the terminal in our IDE, and type the commands:

```
rhyme@ip-172-31-138-134:~$ cd Desktop/code
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl apply -f deployment.yaml
deployment.apps/website created
service/website created
```

Like we see our deployment and service will be created in our cluster.

To verify our elements, we can continue using the **kubectl** to list the deployments:

```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl get deployments
NAME      READY   UP-TO-DATE   AVAILABLE   AGE
website   5/5     5            5           23s
```

Then, we can see our deployment to list the pods we can use:

```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl get pods
NAME                       READY   STATUS    RESTARTS   AGE
website-77f9f8c9cc-598b4   1/1     Running   0          37s
website-77f9f8c9cc-8q8cx   1/1     Running   0          37s
website-77f9f8c9cc-9sjgm   1/1     Running   0          37s
website-77f9f8c9cc-cmk99   1/1     Running   0          37s
website-77f9f8c9cc-slfrm   1/1     Running   0          37s
```

And like we see this is the pods are being deploying and now is running.

Now, if we want to see more details about or pods, we can use this command: **#kubectl get pods -o wide**

```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl get pods -o wide
NAME                       READY   STATUS    RESTARTS   AGE   IP              NODE                             NOMINATED NODE   READINESS GATE
S
website-77f9f8c9cc-598b4   1/1     Running   0          55s   172.31.11.147   ip-172-31-13-155.ec2.internal    <none>           <none>
website-77f9f8c9cc-8q8cx   1/1     Running   0          55s   172.31.12.4     ip-172-31-13-155.ec2.internal    <none>           <none>
website-77f9f8c9cc-9sjgm   1/1     Running   0          55s   172.31.14.41    ip-172-31-13-155.ec2.internal    <none>           <none>
website-77f9f8c9cc-cmk99   1/1     Running   0          55s   172.31.38.241   ip-172-31-38-179.ec2.internal    <none>           <none>
website-77f9f8c9cc-slfrm   1/1     Running   0          55s   172.31.47.170   ip-172-31-38-179.ec2.internal    <none>           <none>
```

So, we can see for example, which node, each pod was deployed in the IP for each pod.

We can also see our nodes by: **#kubectl get nodes**

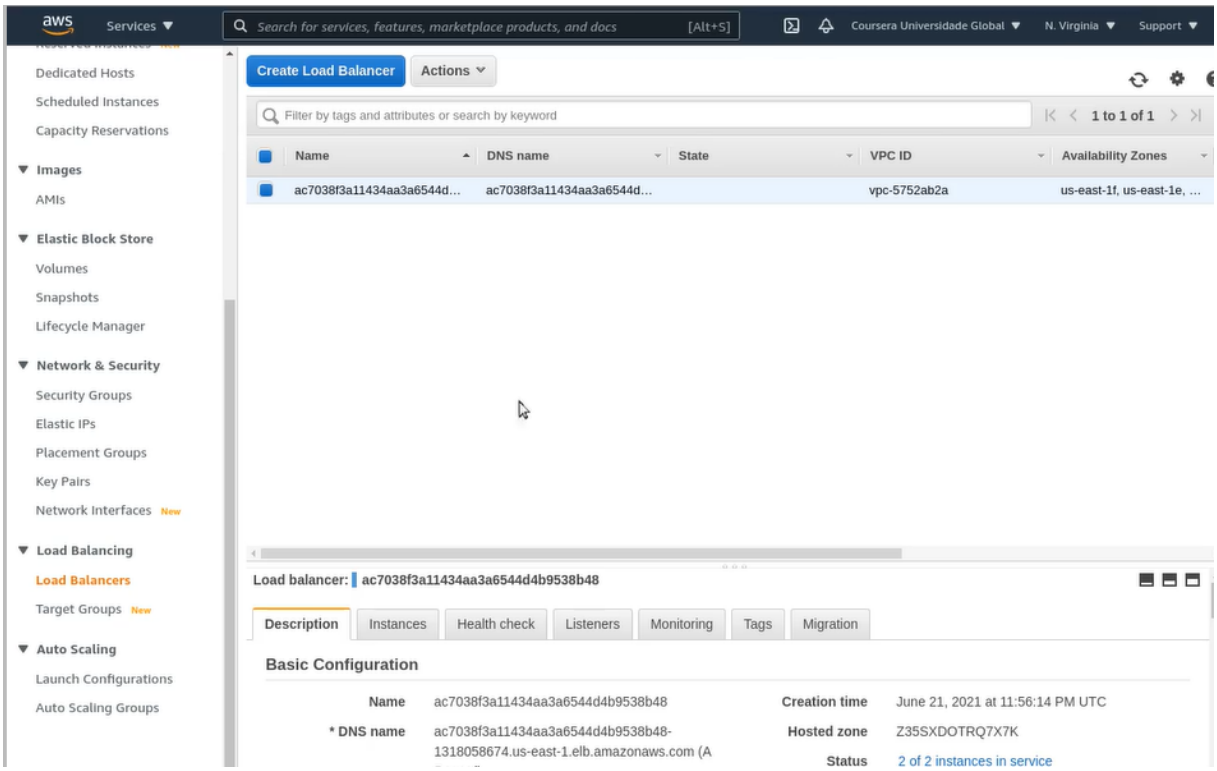
```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl get nodes
NAME                             STATUS   ROLES    AGE   VERSION
ip-172-31-13-155.ec2.internal    Ready    <none>   17m   v1.20.4-eks-6b7464
ip-172-31-38-179.ec2.internal    Ready    <none>   17m   v1.20.4-eks-6b7464
```

Similarly, we can have a list of our services:

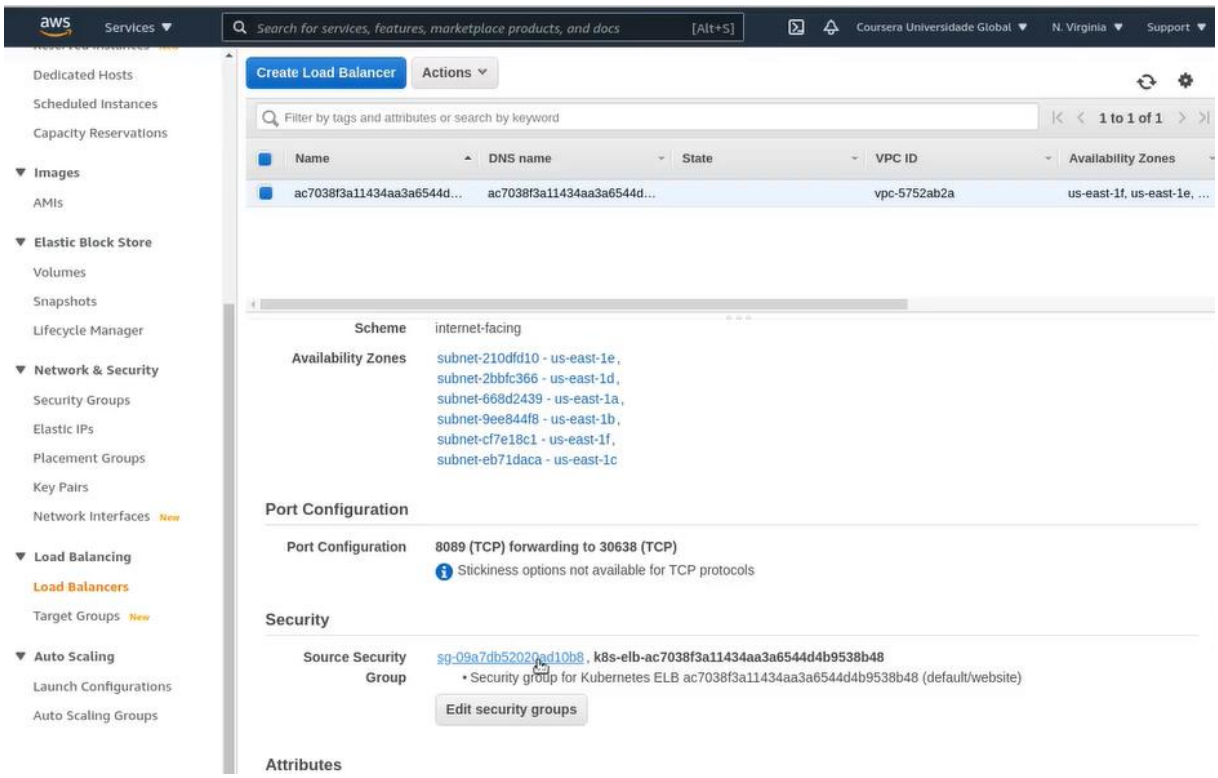
```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl get svc
NAME         TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)          AGE
kubernetes   ClusterIP      10.100.0.1       <none>                                                                     443/TCP          28m
website      LoadBalancer   10.100.239.188   ac7038f3a11434aa3a6544d4b9538b48-1318058674.us-east-1.elb.amazonaws.com   8089:30638/TCP   84s
```

Now, before access, let's go to our load balancer security group configuration and make our application available for any source.

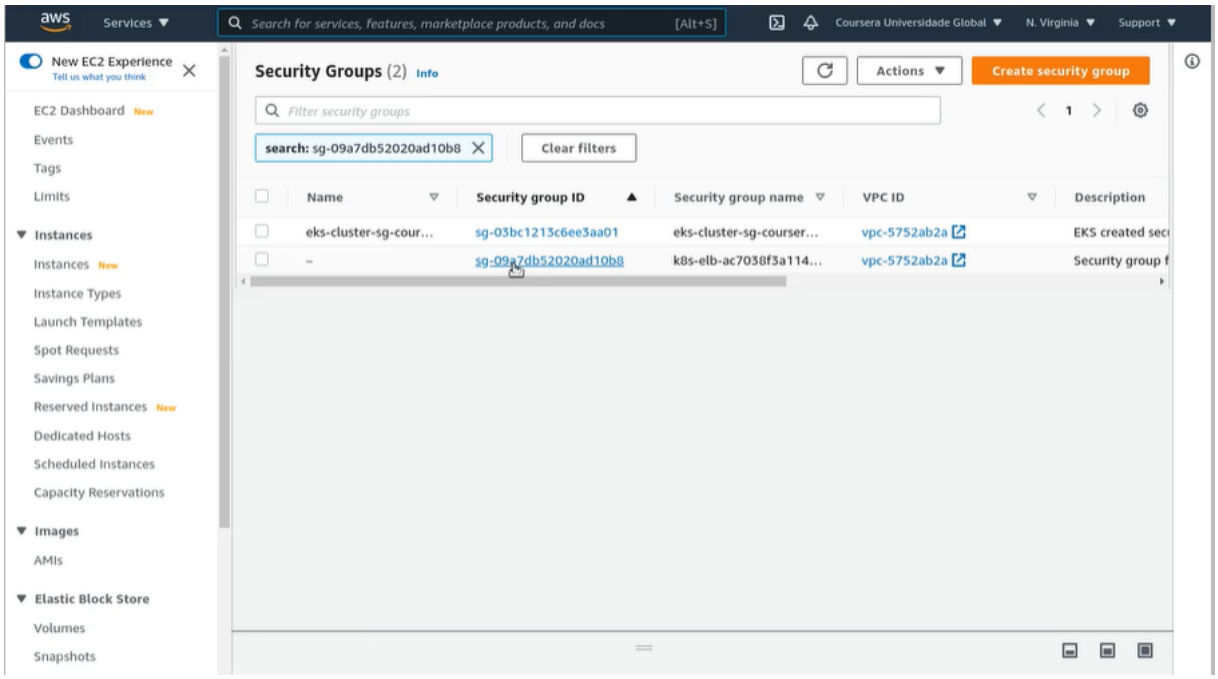
So, on going to the AWS console, in the left menu, let's click on **'Load balancer':**



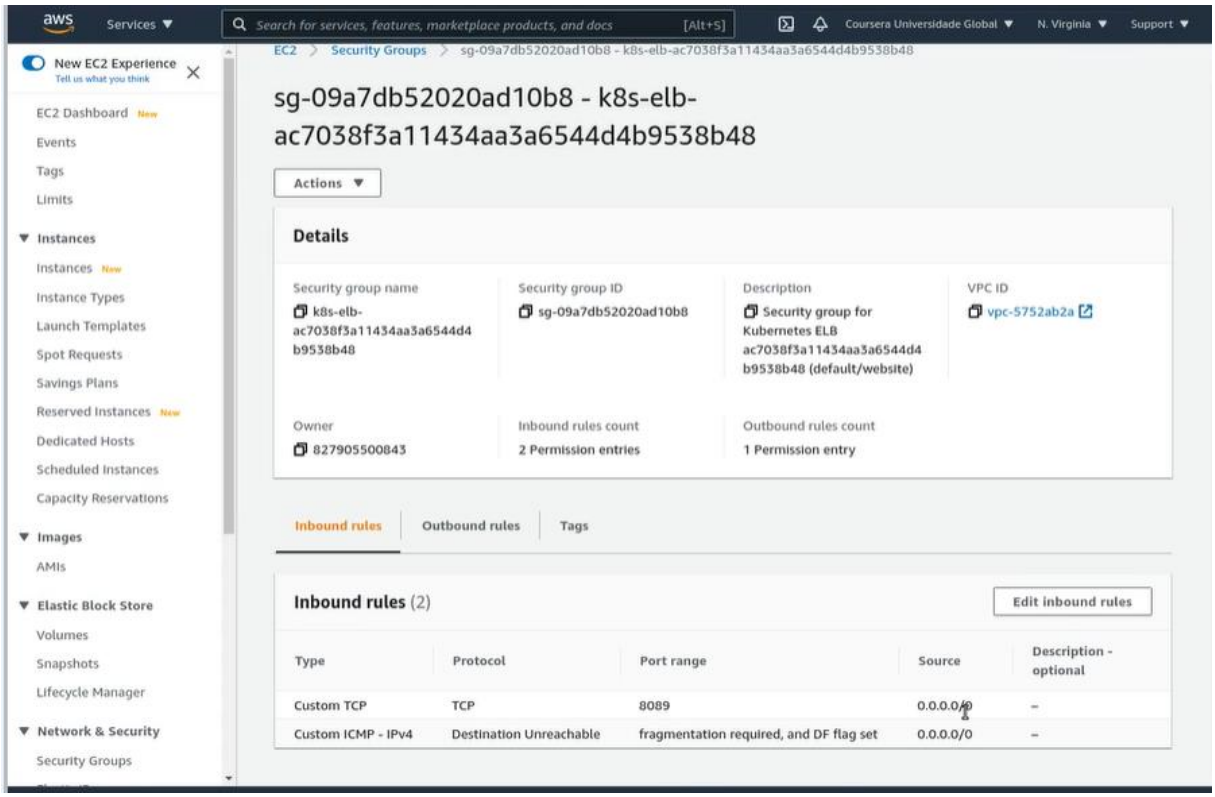
Later, we scroll down, and click on this link:



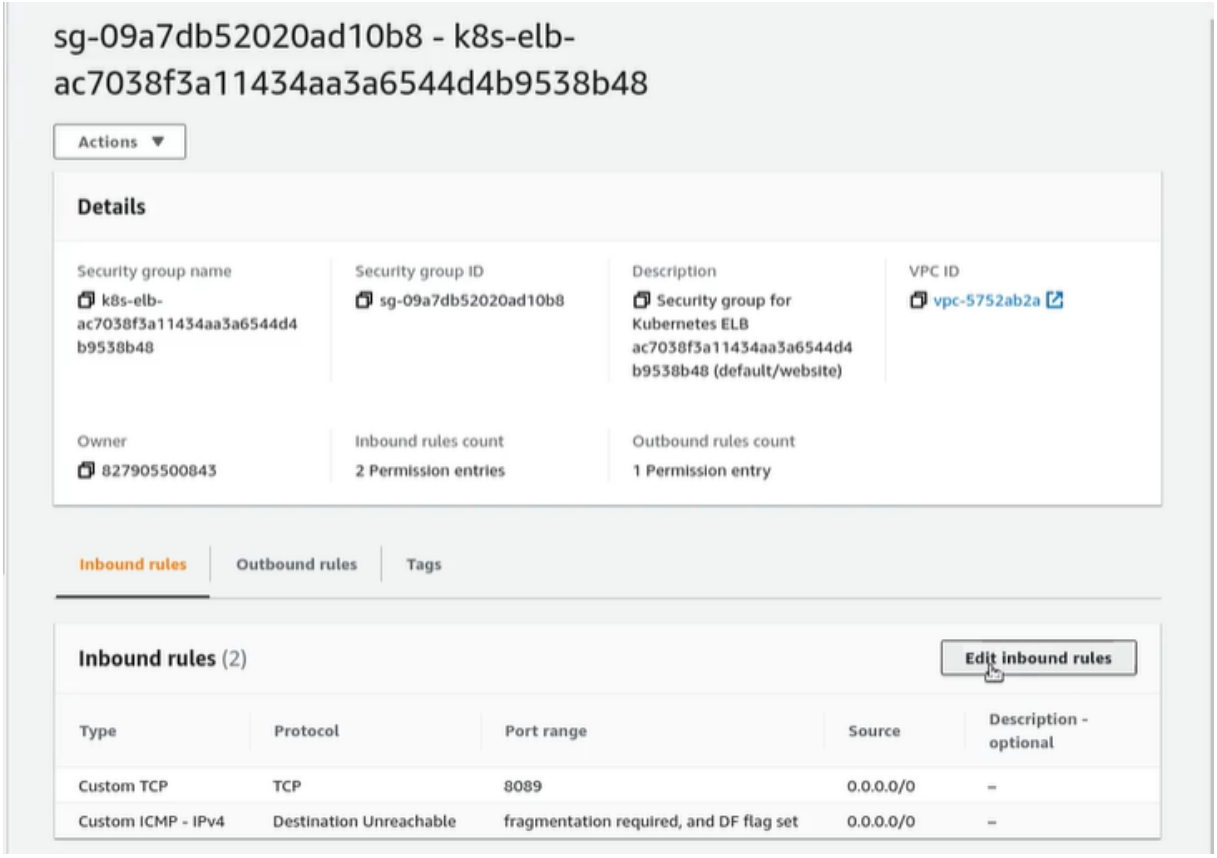
Security Groups acts like a firewall to control the inbound and outbound in our load balancer. We check if we have in the inbound rule open for our peace.



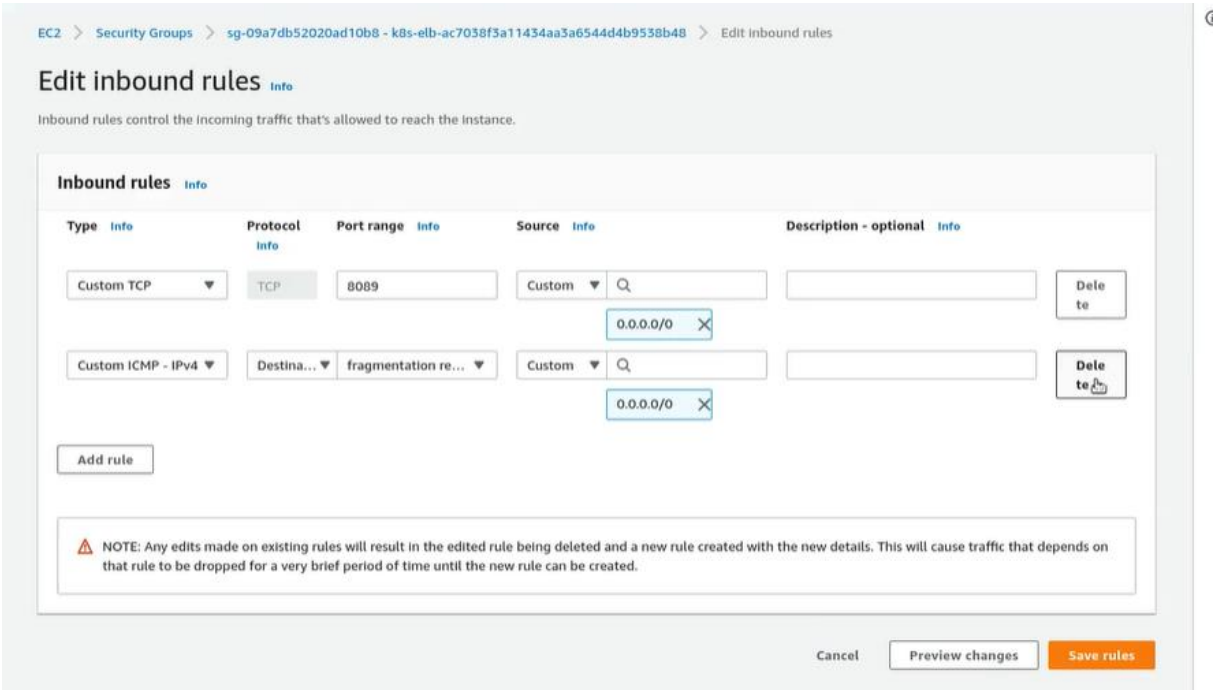
Like we see, we have 0.000/0.

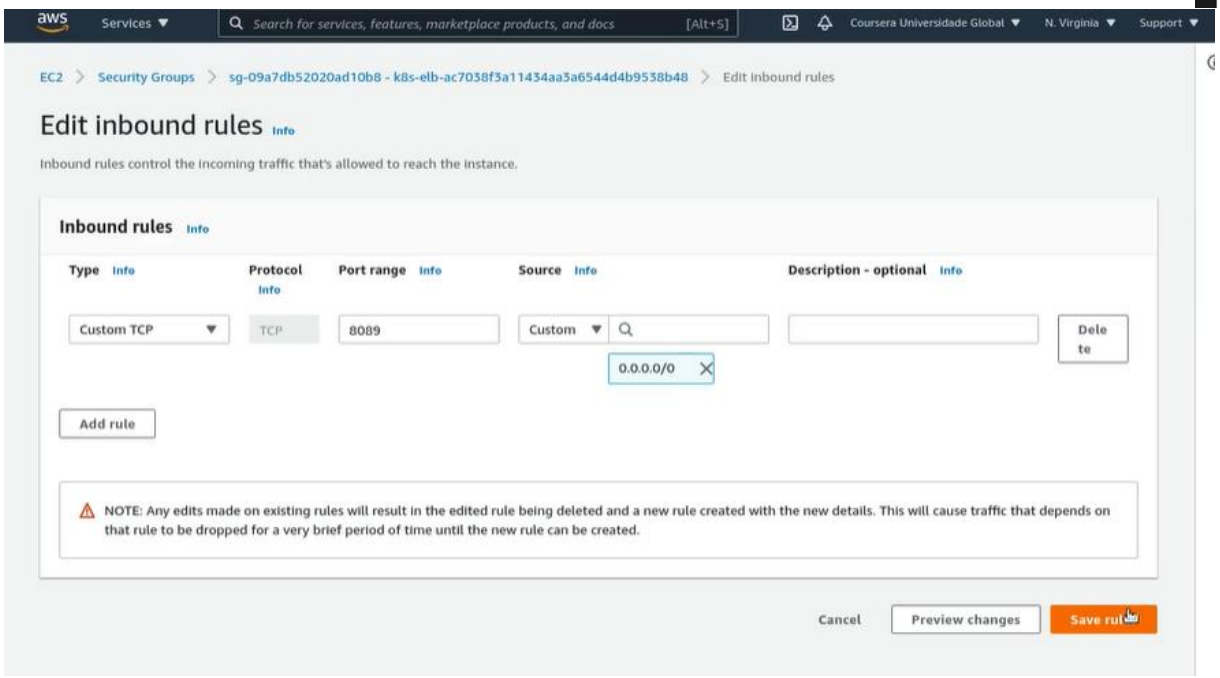


Let’s delete any other rule:



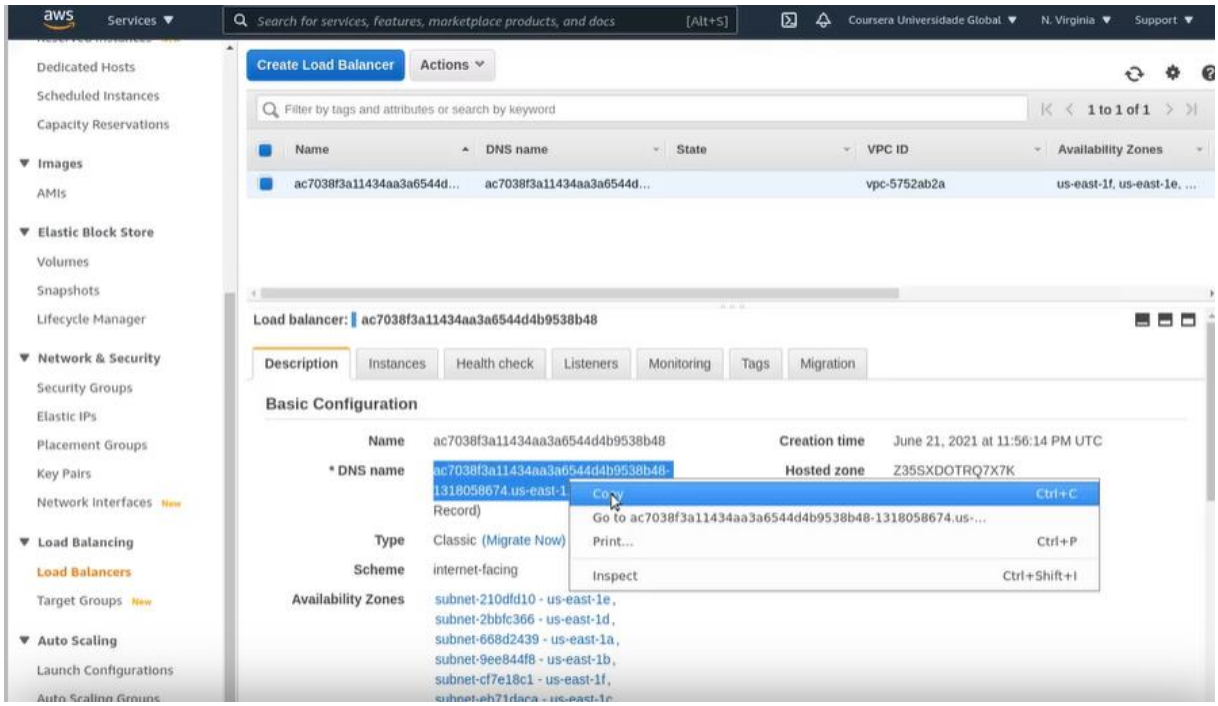
Click on **-> Edit inbound rules**





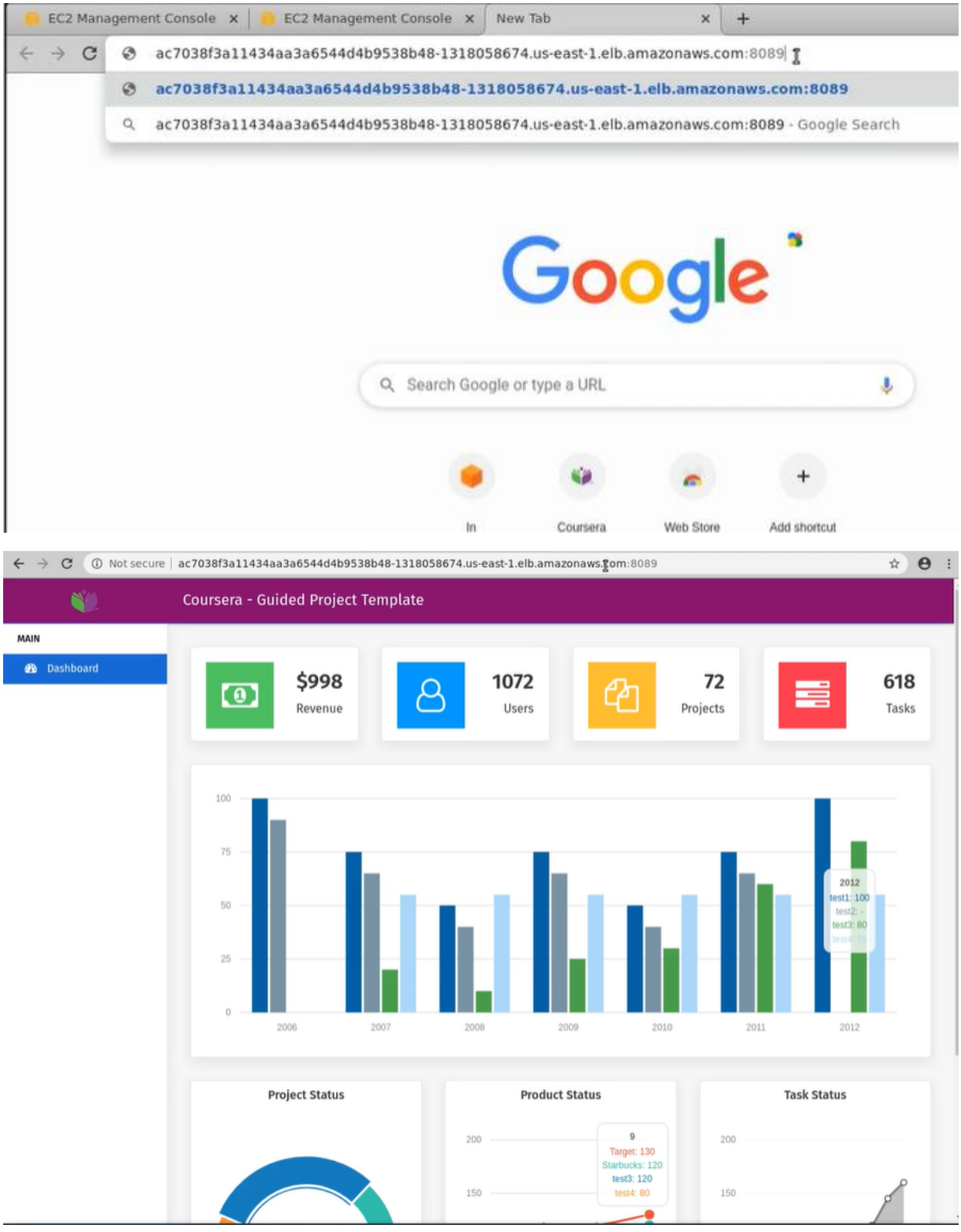
We Clicked on -**> Delete -> Save rules**

Later, we go back on the load balancer configuration page and we copy the DNS name:



Now, we pass to a new tab and complete the URL with the port 8089, which is the port our container application is using:

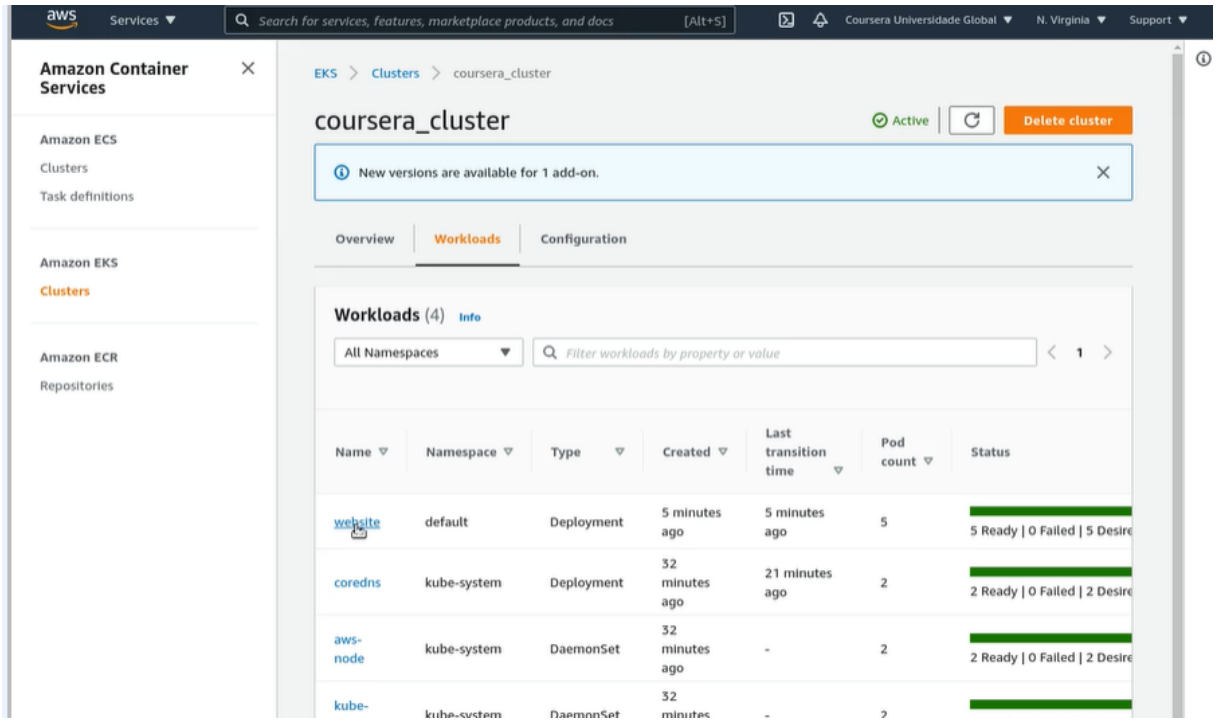
And we paste it:



Like we see, our population is running in our EKS cluster on AWS.

We can also open our EKS cluster dashboard to see the workload steps, to see our application deployment

Elastic Kubernetes Service -> Clusters -> Coursera_cluster



Then like we see in the picture above, our deployment is running in our cluster.

Task 8: Scale the Node Group to support the increase of new pods in the Kubernetes cluster

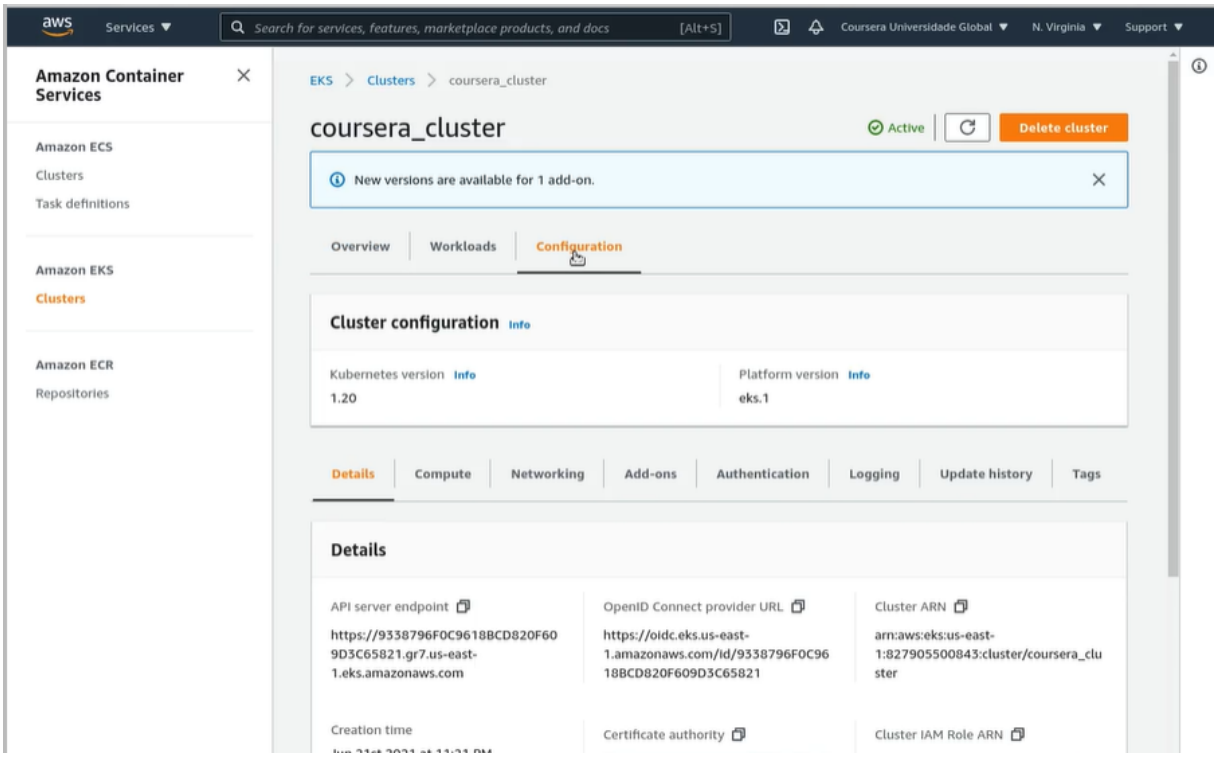
In this task, we gonna see how we can scale our cluster and deploy more containers for our application based on the environment we have deployed. Our instance can support around 17 pods in total.

Considering we have application pods and Kubernetes system pods, it means if we need more pods, more containers for our application, we need to scale our load group at the more virtual machines.

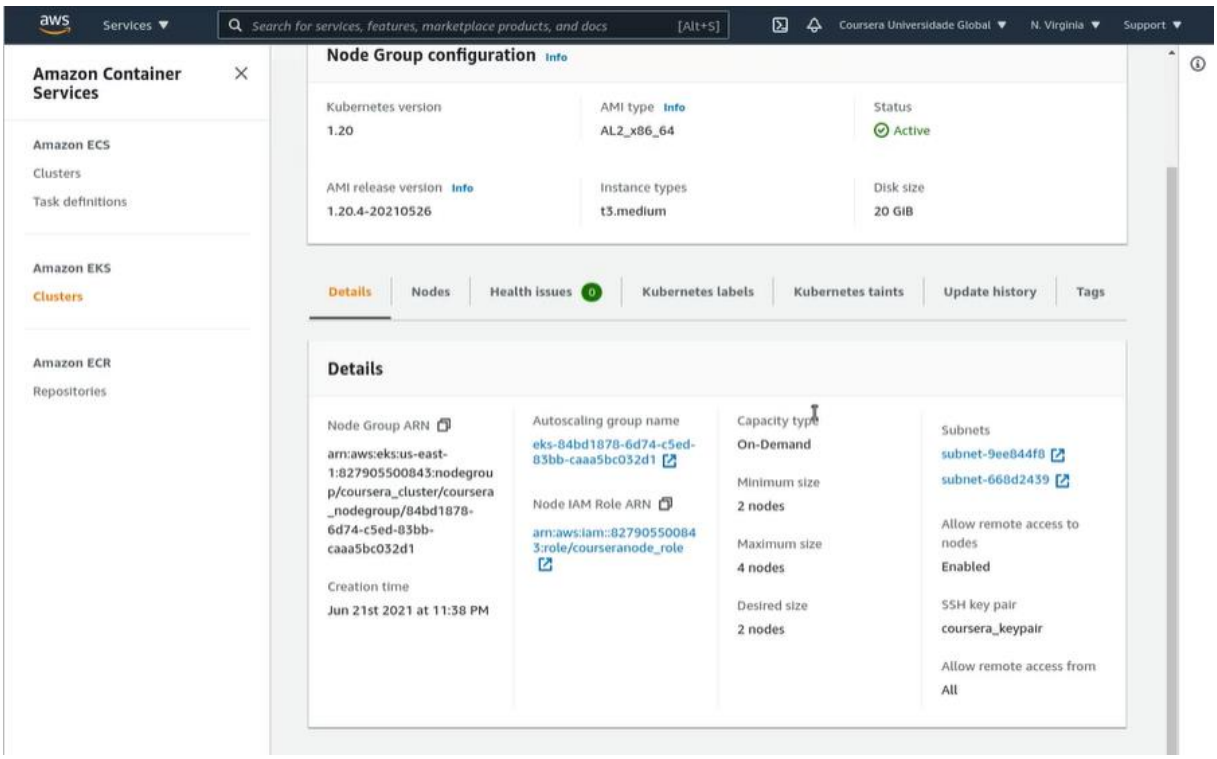
We can do that manually, but the big advantage of using a management services like Elastic Kubernetes services is to make it automatically.

Firstly, we gone to AWS console and see our cluster and node group configuration:

Elastic Kubernetes Service -> Clusters -> Coursera_cluster -> configurations

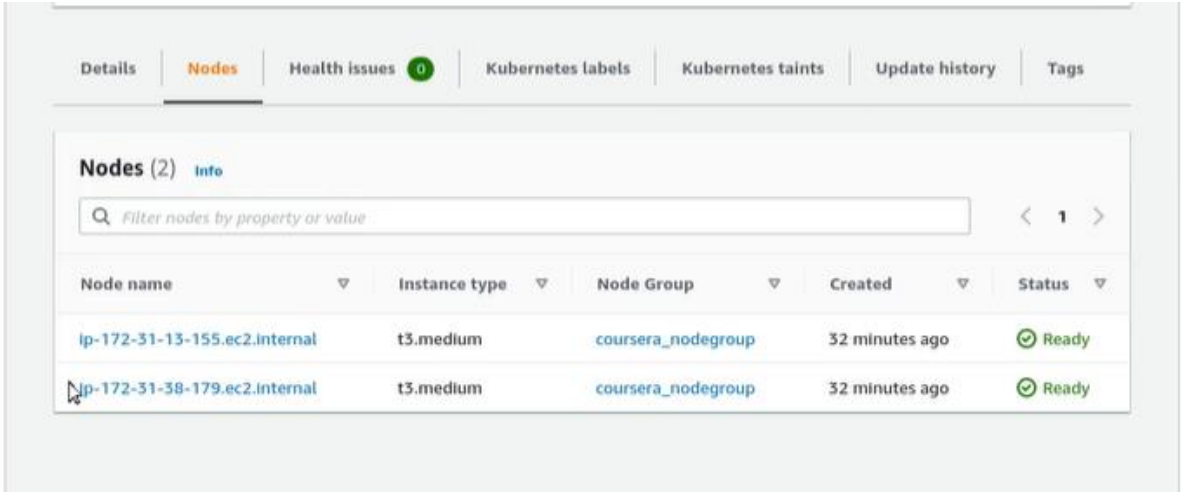


-> compute



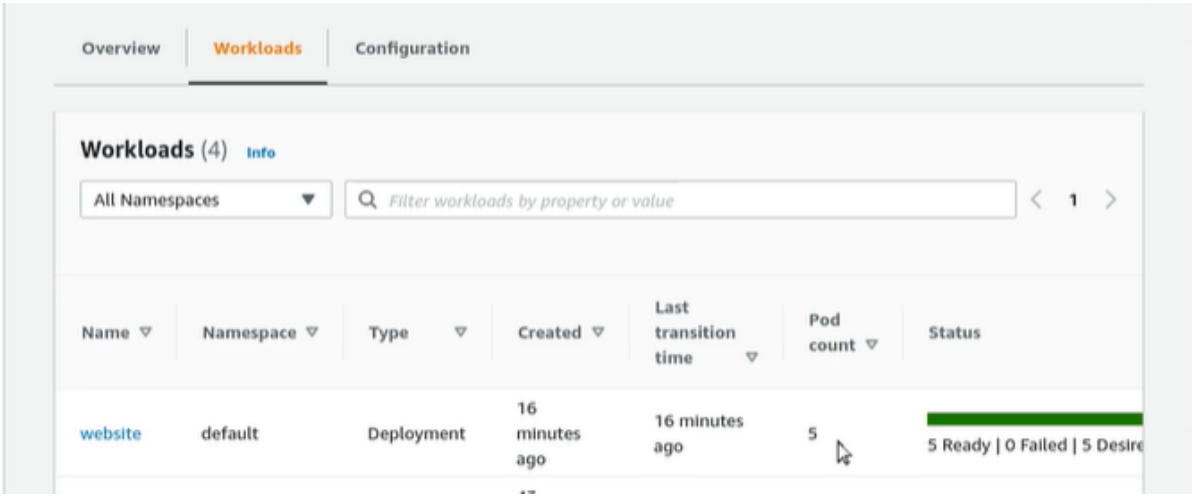
We see in capacity type we have two nodes minimal for nodes maximum and two nodes desire it.

Now, we click on nodes Notes step



And like we see, we have two nodes up and running.

And if we go to our closer workload information, we see we have our application deployment running five boards.

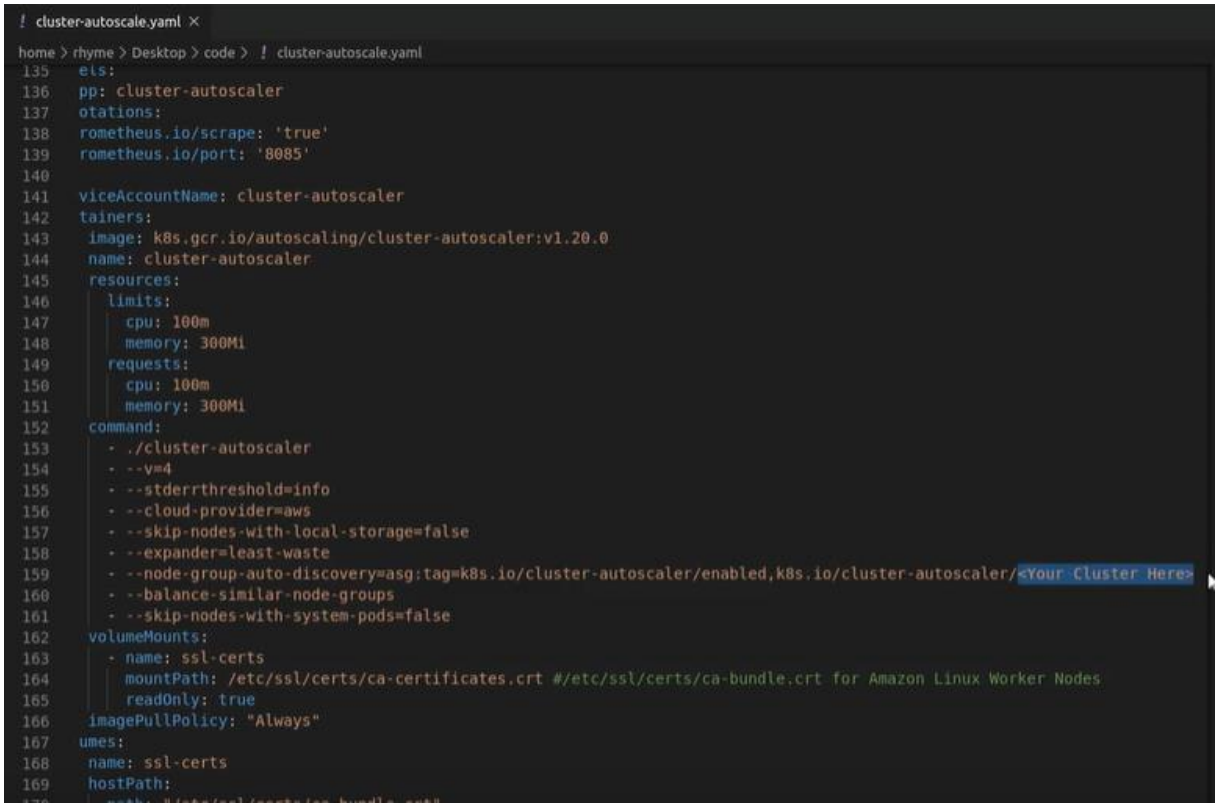


It means five containers.

Kubernetes can be installed in our cluster using a deployment like we did for our application.

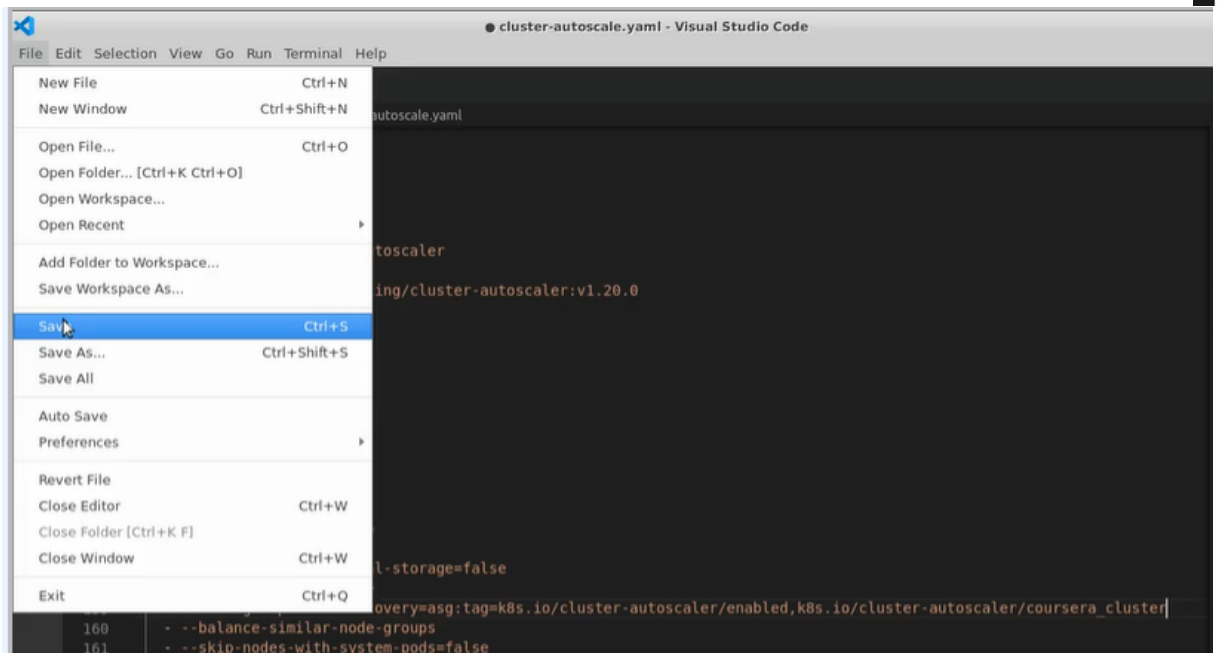
We have an E M O file prepared for that with our information needed for Kubernetes version 1.20. We just need to inform our cluster name inside this file for that.

You scroll down and change this placeholder here to include our cluster name:



In our case is **Coursera_cluster:**

Later, we save the file:

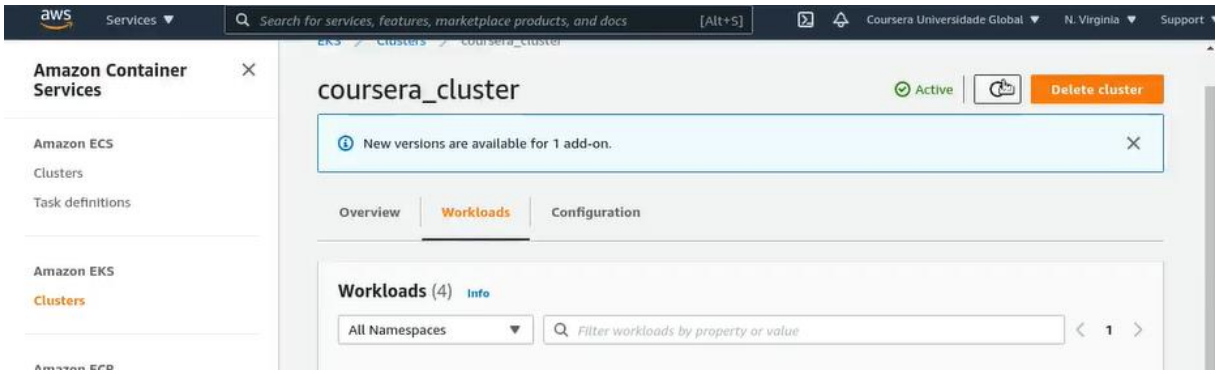


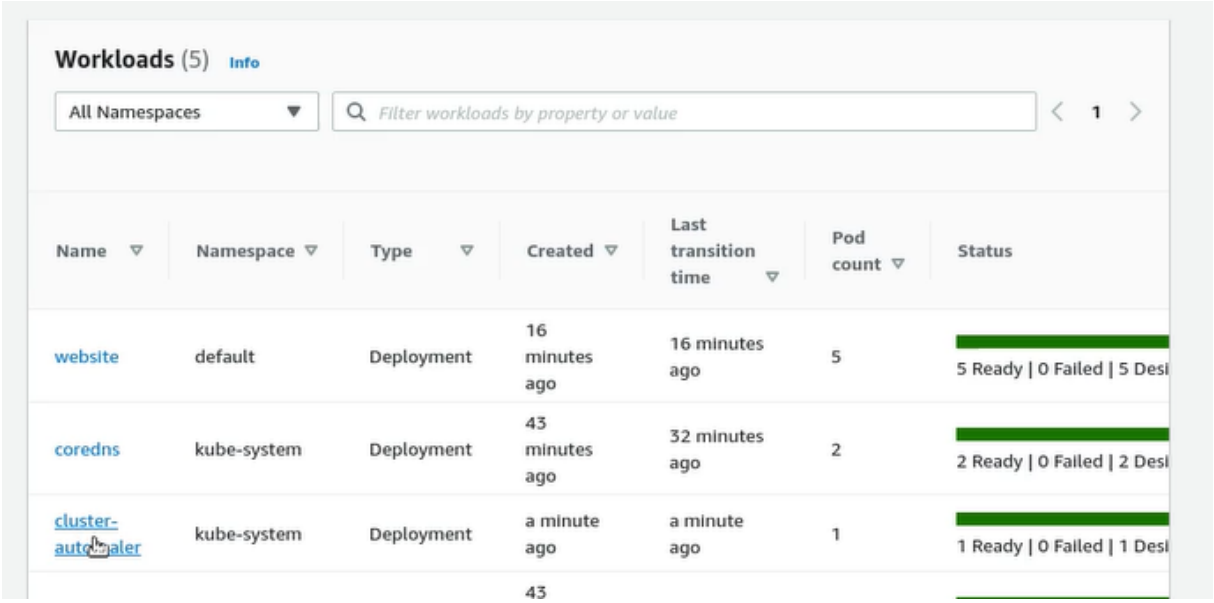
Now, let's go to the terminal:

```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl apply -f cluster-autoscale.yaml
serviceaccount/cluster-autoscaler created
clusterrole.rbac.authorization.k8s.io/cluster-autoscaler created
role.rbac.authorization.k8s.io/cluster-autoscaler created
clusterrolebinding.rbac.authorization.k8s.io/cluster-autoscaler created
rolebinding.rbac.authorization.k8s.io/cluster-autoscaler created
deployment.apps/cluster-autoscaler created
```

Our autoscaler will be deployed.

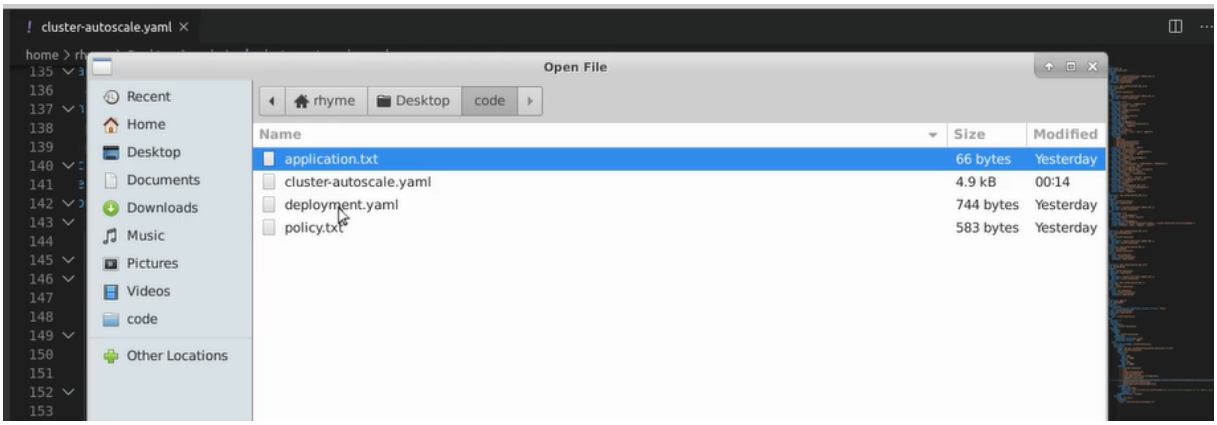
Now, let's take a look in closer workloads, on refresh it in the first:



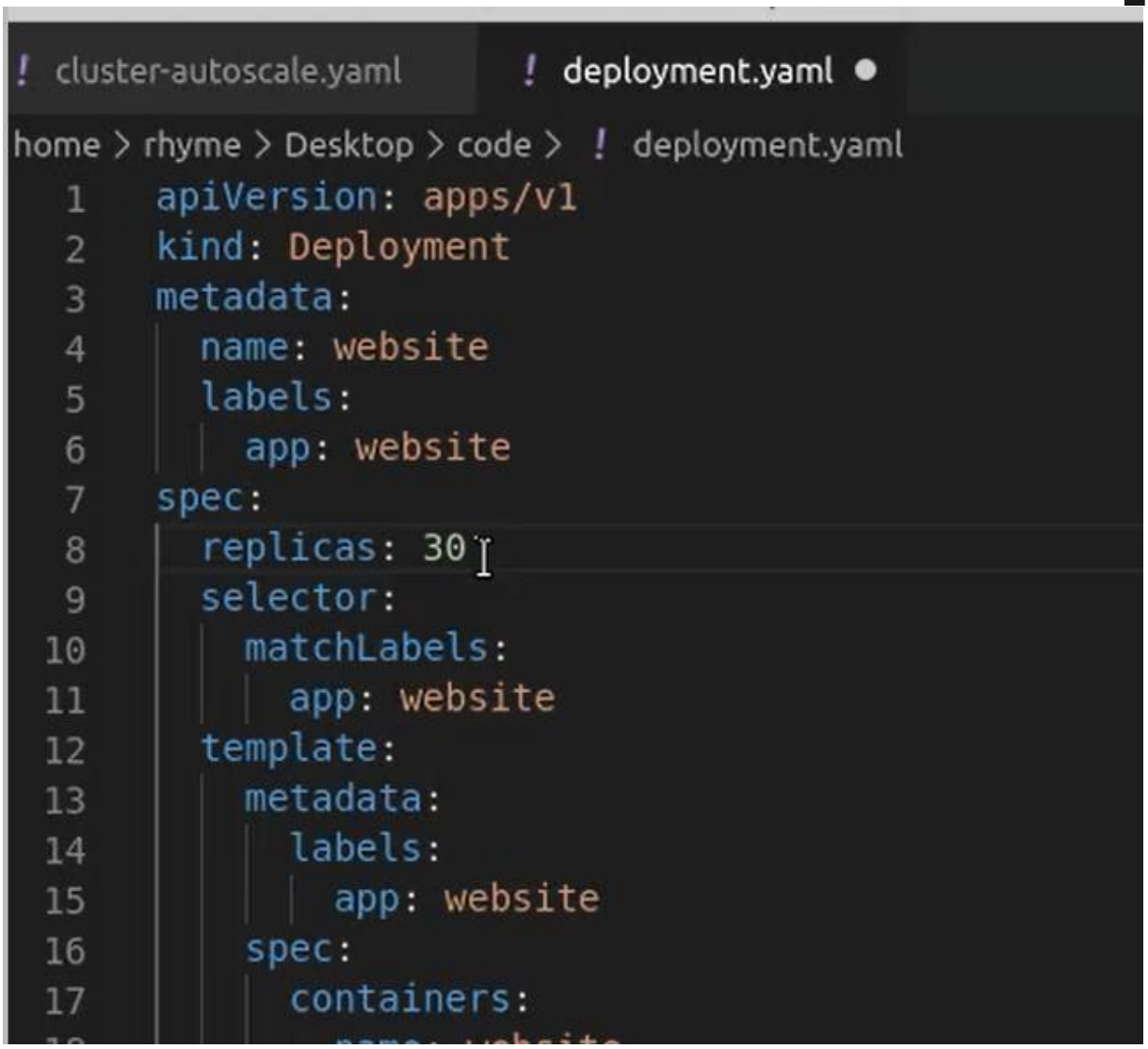


We see we have a cluster-autoscaler deployment.

To validate if autoscaler is working, we can change the replicas in our application deployment and deploy again. Back to the visual studio and open the deployment yaml file inside our code folder:

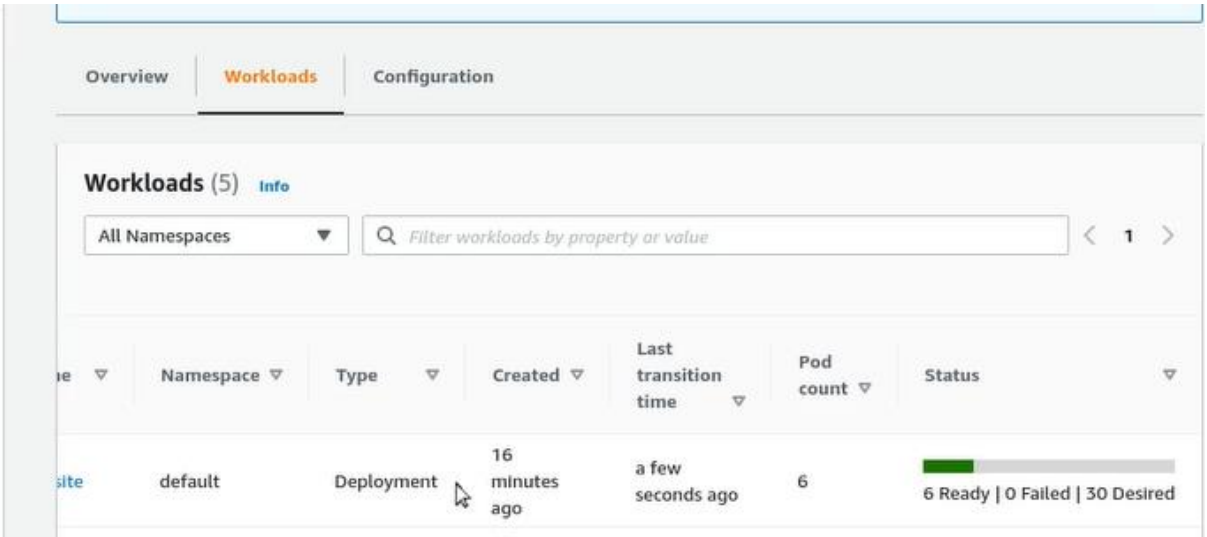


We change their replicas for 30 points:



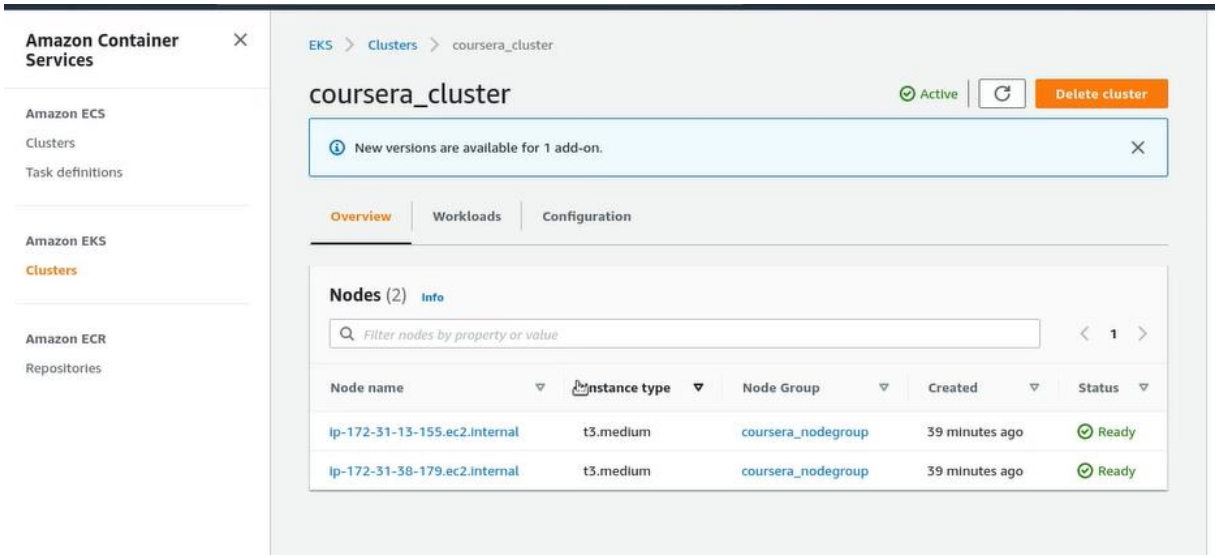
And we save the file.

```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl apply -f deployment.yaml
deployment.apps/website configured
service/website unchanged
```

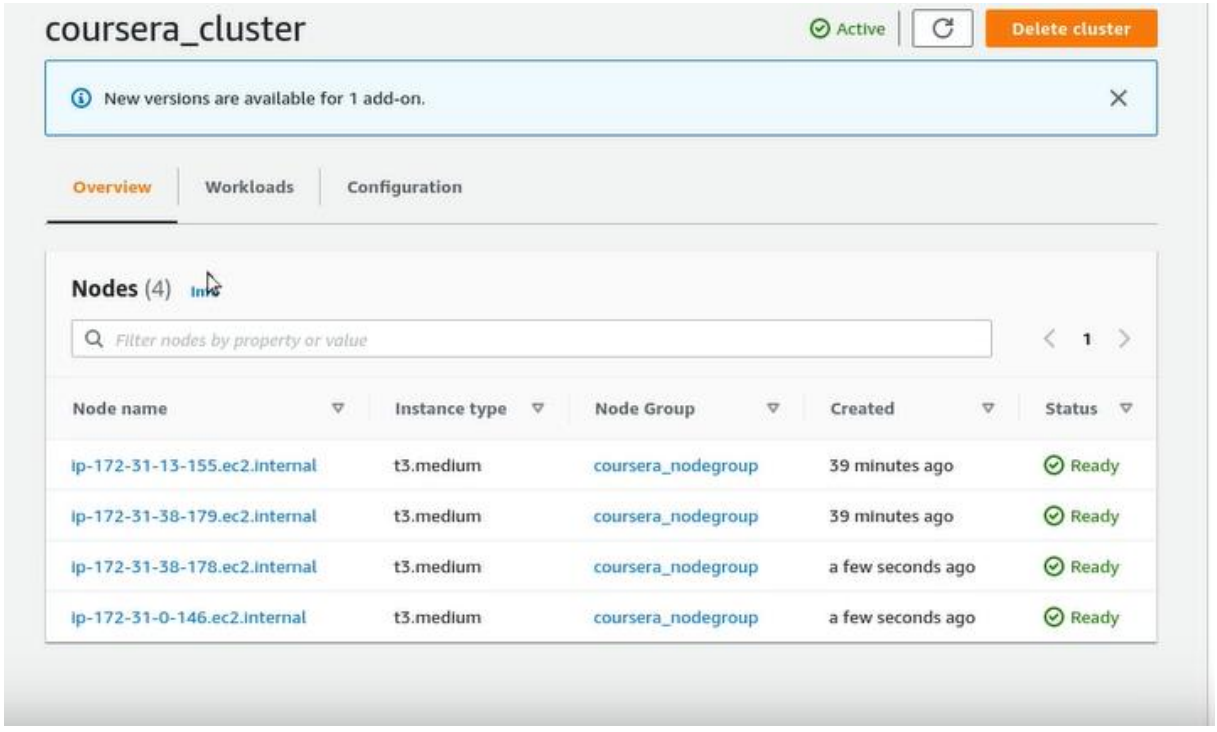


We have more pods being applied.

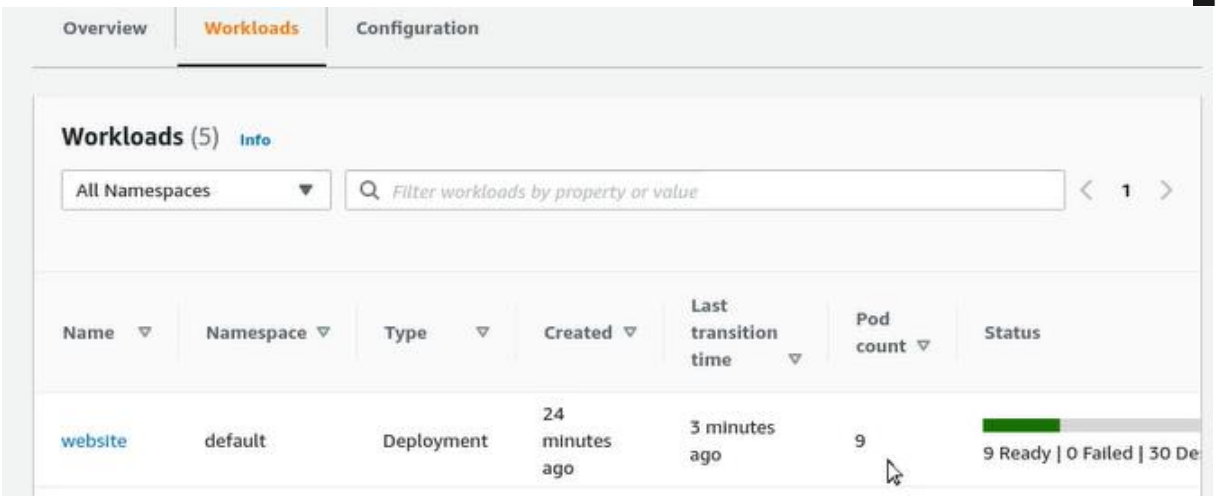
On clicking on **'Overview'** page:



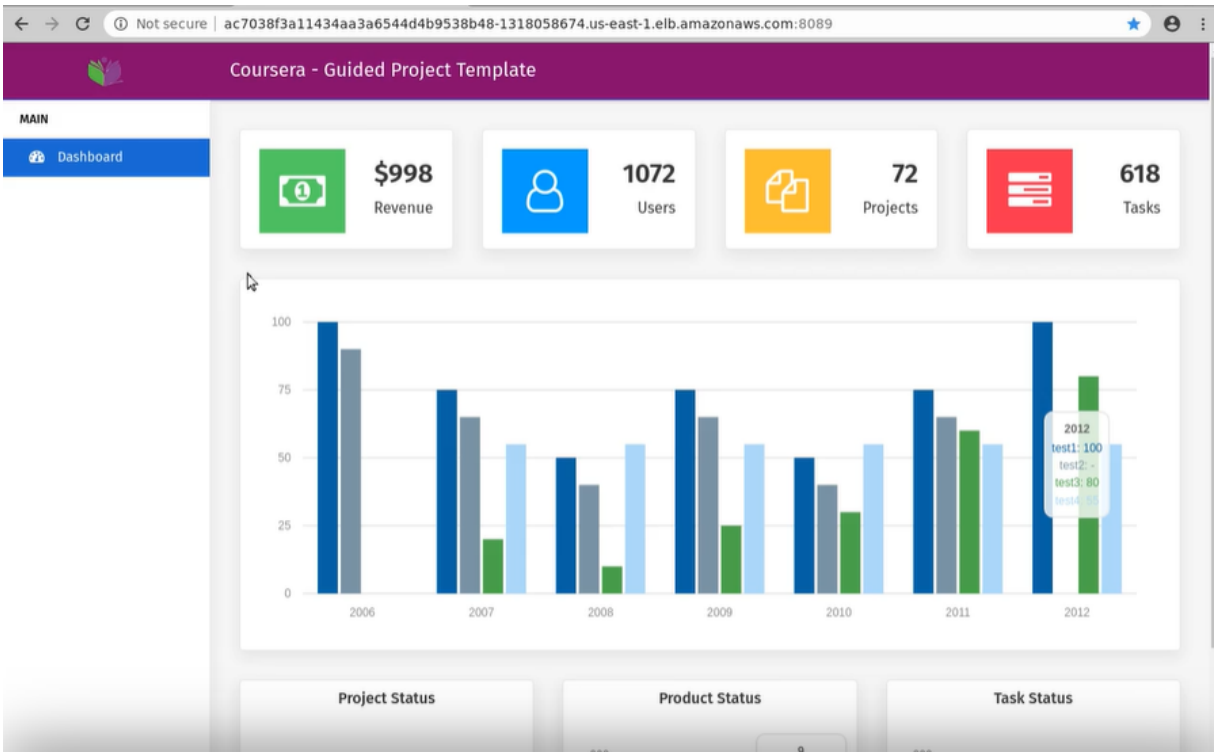
The Kubernetes cluster, we'll ask our node groups to create more virtual machines instance to deploy our pods. It can take several minutes if needed.



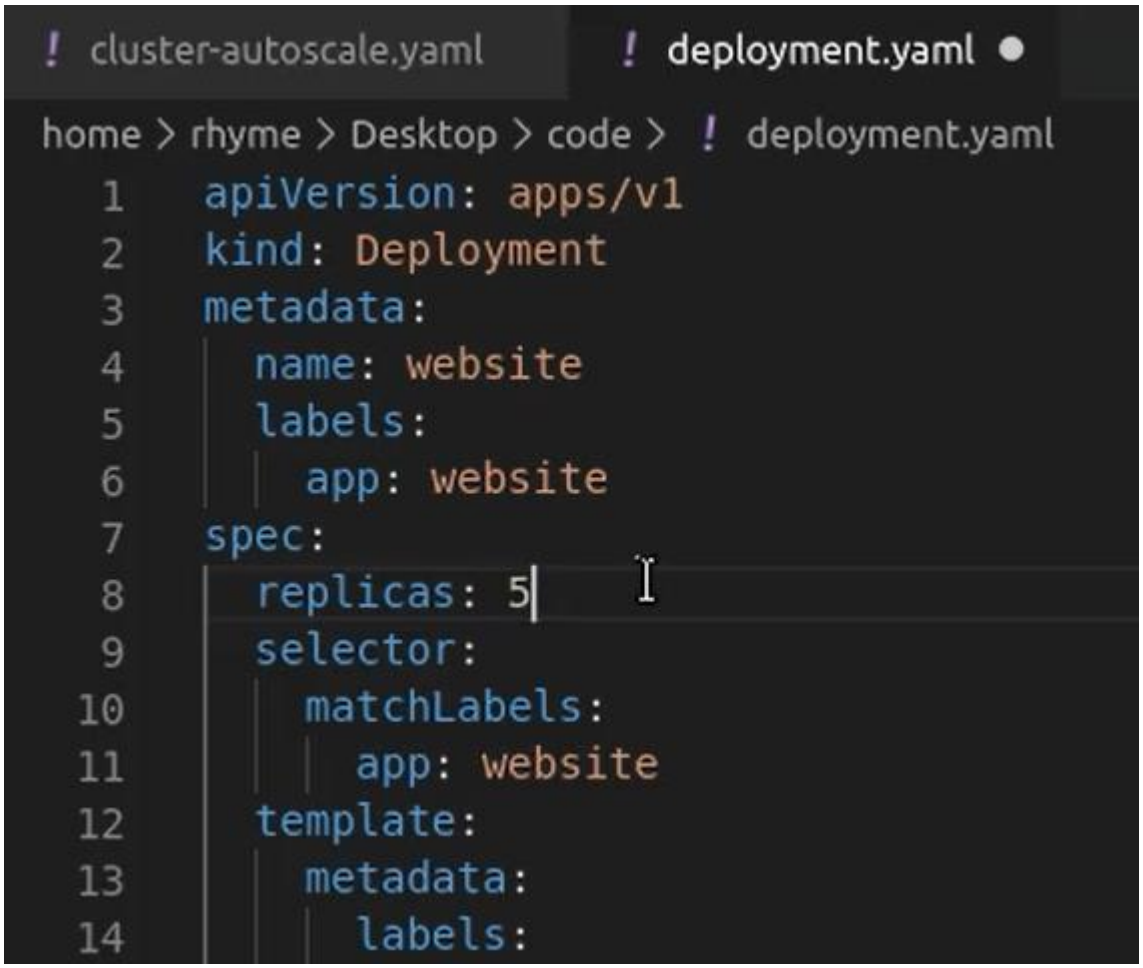
We can see now we have far Nodes, which is the maximum nodes. We define it for our hottest killer in those group configuration and in the workloads we can see we have more pods deploy it:

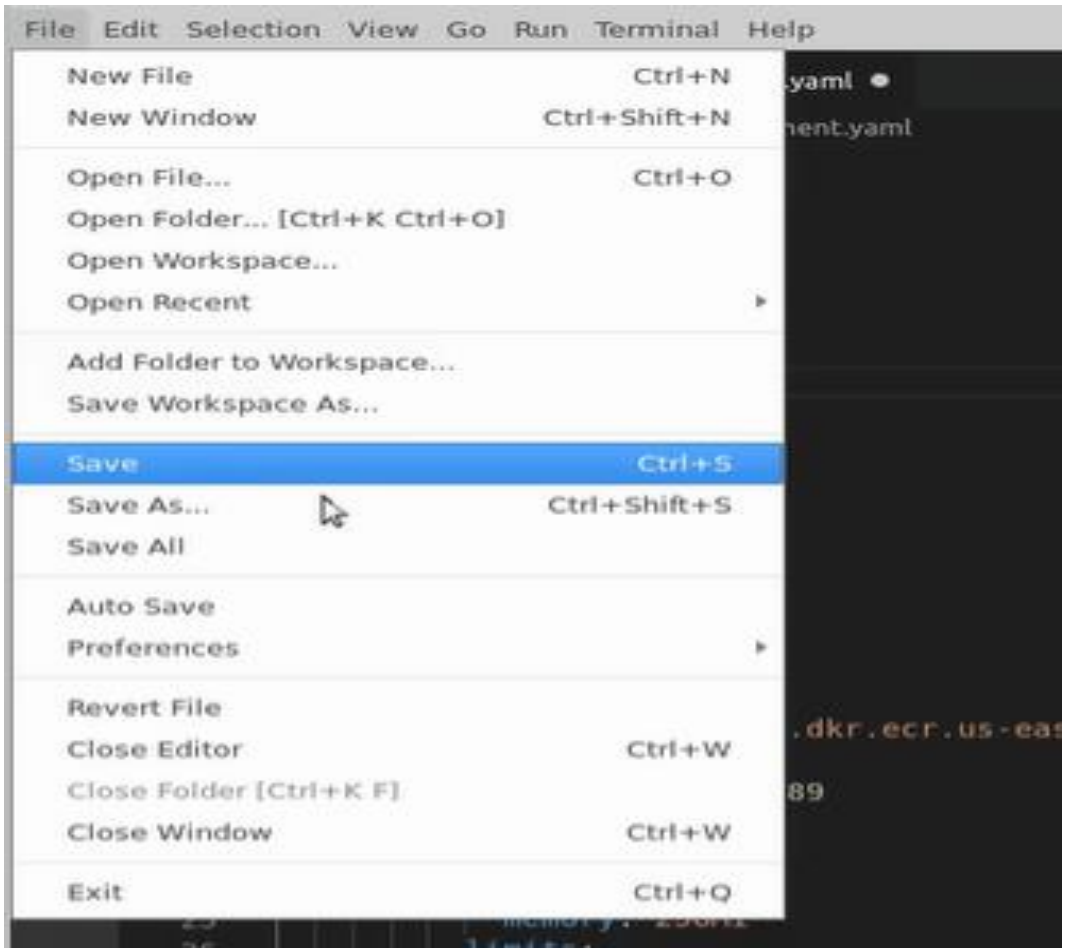


We can open a web browser tab to see if our application is still running:



What we can do now is reduced again the number of replicas for our application and see out scalar reduced the number of notes deploy it in the deployment. We change the replicas number back to five:

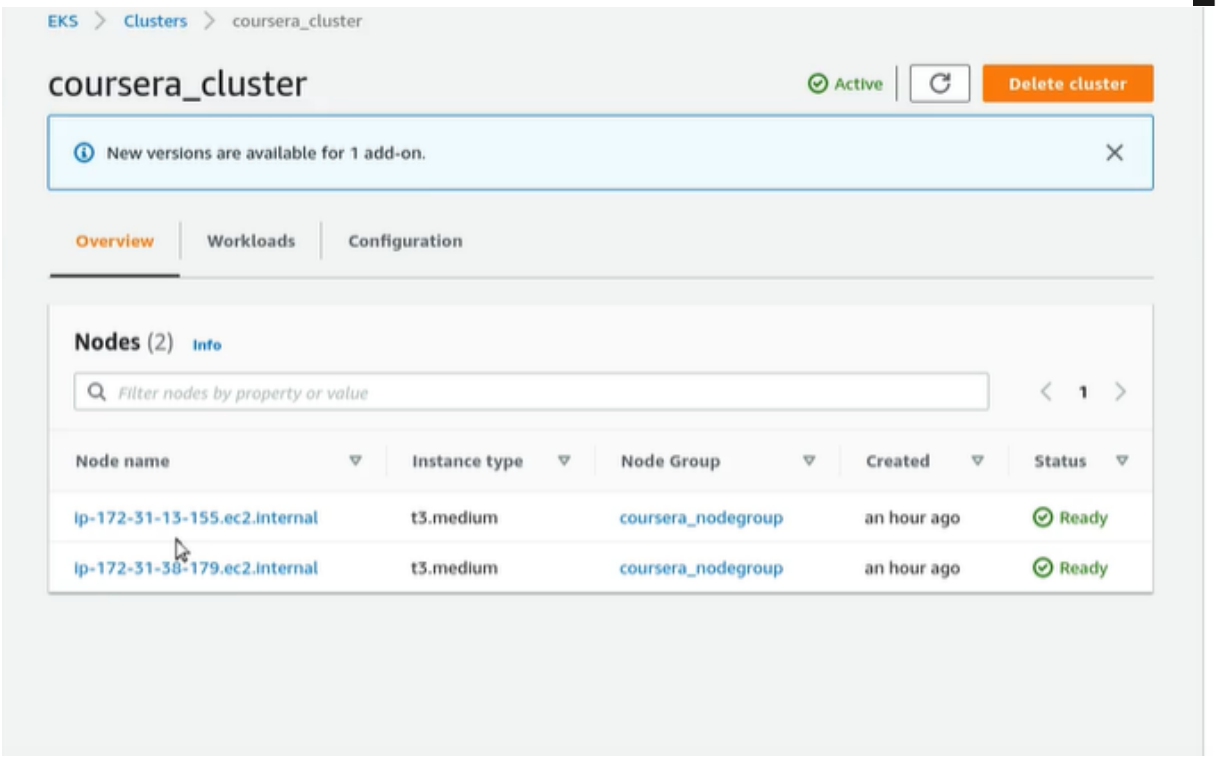

! cluster-autoscale.yaml
! deployment.yaml
home > rhyme > Desktop > code > ! deployment.yaml
1 apiVersion: apps/v1
2 kind: Deployment
3 metadata:
4 name: website
5 labels:
6 app: website
7 spec:
8 replicas: 5
9 selector:
10 matchLabels:
11 app: website
12 template:
13 metadata:
14 labels:



and deploy using **kubectl** too:

```
rhyme@ip-172-31-138-134:~/Desktop/code$ kubectl apply -f deployment.yaml
deployment.apps/website configured
service/website unchanged
```

And now if we refresh the cluster here:

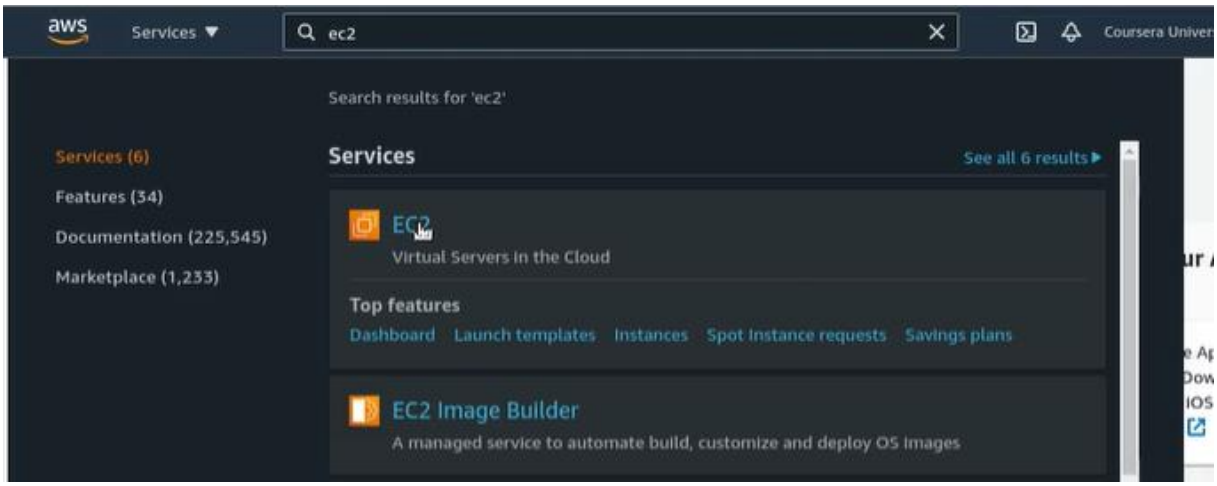


We have again only two nodes deployed in our cluster. Our outer scalar is working as expected.

Task 9: Clean Up the environment

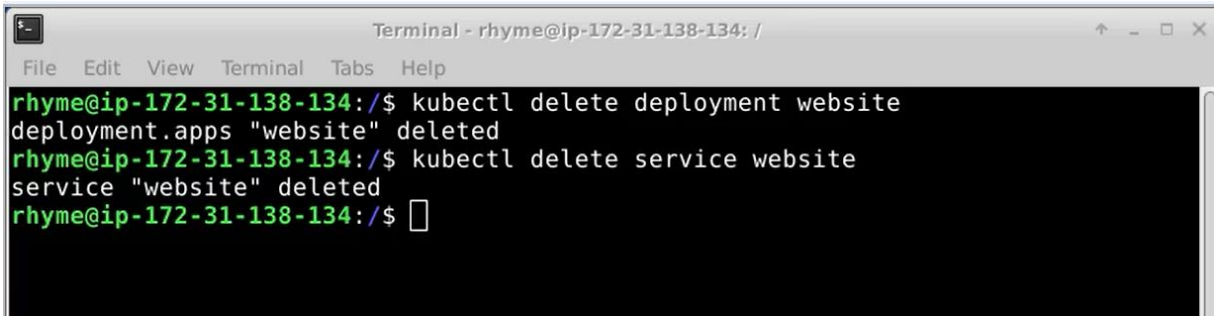
In this task, we will see how to cleanup our environment to avoid further costs.

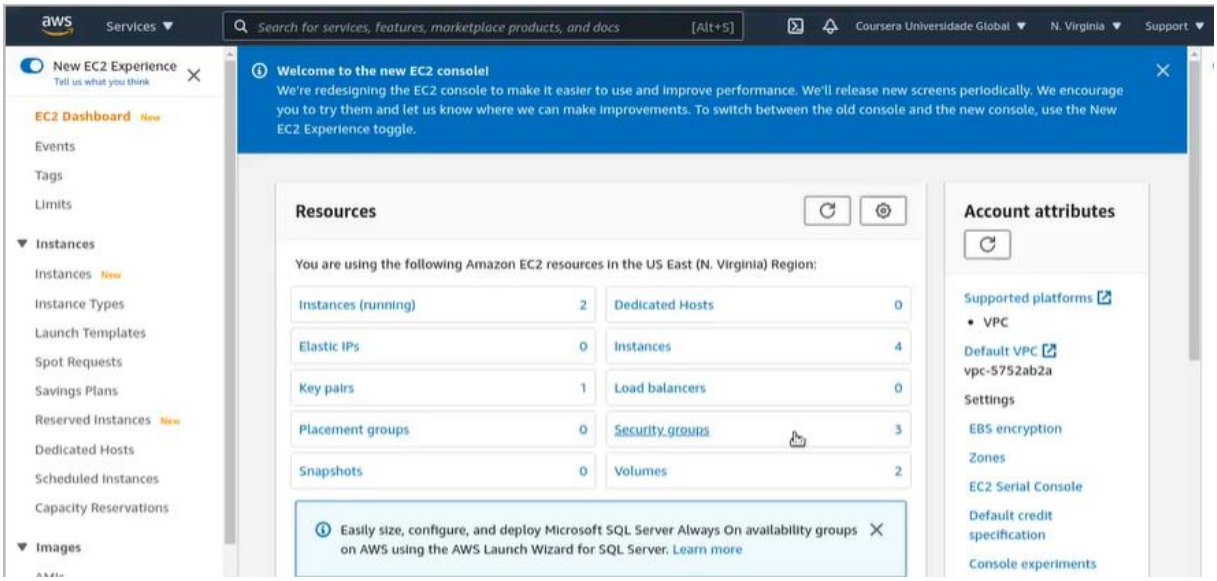
First let's take a look at some of his services.



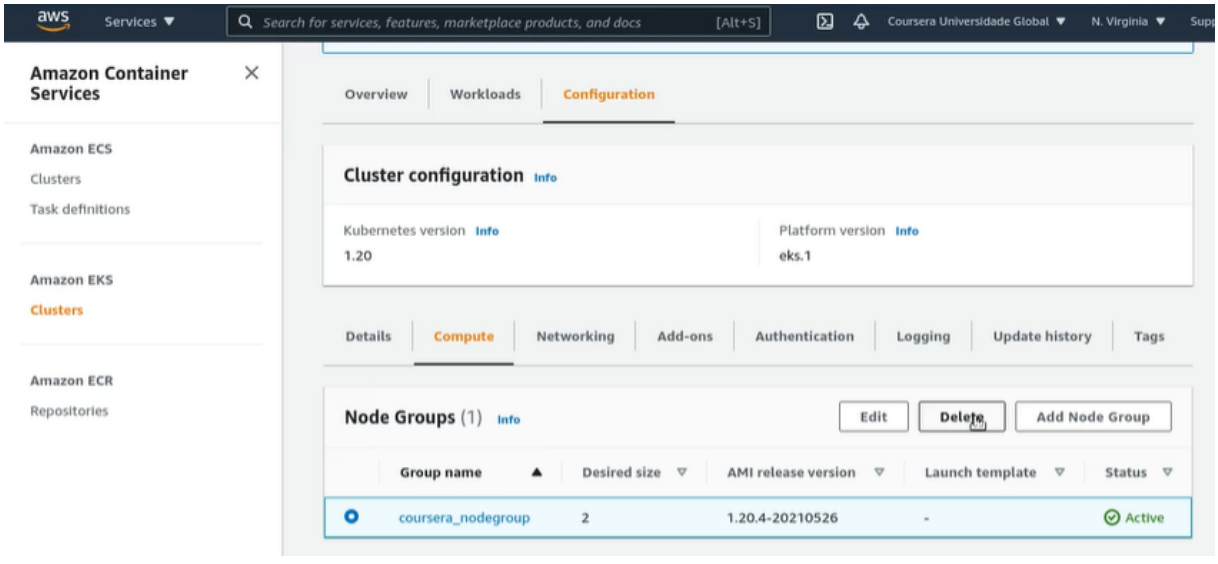
Some items were created by cluster, others by note group and others because we made a deployment in the service the creation.

Let's delete our deployment in our services.





EKS -> Configuration -> Compute



II. QCM

1. **Which AWS Service can be used to store container images ?**

- ○ Elastic Kubernetes Service
- ○ Elastic Docker Registry
- ◉ Elastic Container Registry

Correct
Correct. Elastic Container Registry is the service to store our images repositories.

2. **Which Docker command do we use to build our container images?**

- ○ docker ps
- ◉ docker build
- ○ docker run

Correct
Correct. This is the right command

3. **Can we choose the size of the nodes (virtual machines) we will have in our Node Group?**

- ◉ Yes
- ○ No

Correct
Correct. You can define the virtual machine size based on EC2 predefined types

4. **Which Kubernetes tool do we use to deploy containers in our Elastic Kubernetes Cluster?**

- ◉ kubectl
- ○ docker deploy

Correct
Correct. This tool could be used to deploy and manage the Kubernetes cluster

5. **Which part of Kubernetes is responsible for exposing our application for users?**

- (●) Services
- ( ) Watch
- ( ) Deployments

Correct
Correct. Services define how the application will be exposed and available for users.

6. **What other AWS resources does AWS create when an Elastic Kubernetes cluster is deployed?**

- (●) load balancers, public IPs, and others.
- ( ) No resources are created by AWS in an Elastic Kubernetes Service Deployment
- ( ) WebApps, Lambda, WAF and others.

Correct
Correct. AWS on behalf of the user creates a lot of other resources to support the Kubernetes architecture

7. **When we want to increase or decrease the number of containers running in our Cluster, which feature we need to change in deployment.yaml file?**

- ( ) Change the value in the "pod" feature.
- ( ) Change the value in the "image" feature.
- (●) Change the value in the "replica" feature.

Correct
Correct. Changing the value of the replica feature will create a new ReplicaSet and increase or decrease the number of containers running in the cluster.

8. **Which Kubernetes resource do we need to install to enable the node group scale?**

- (●) Auto Scaler
- ( ) Auto Deployment.
- ( ) Kubernetes Dashboard

Correct
Correct. You can install Kubernetes Auto Scaler to scale your node group automatically

9. **After creating a Load Balancer service in our Kubernetes Clusters, which information do we use to access our application?**

- (●) DNS Name
- ( ) IP Number

Correct
Correct. Load Balancer creates a Public DNS name we can use to access our application.

10. **Node groups with Auto Scaler enabled can take several minutes to scale in or scale out, is that correct?**

- ( ) No.
- (●) Yes

Correct
Correct. Any scale operation can take several minutes.