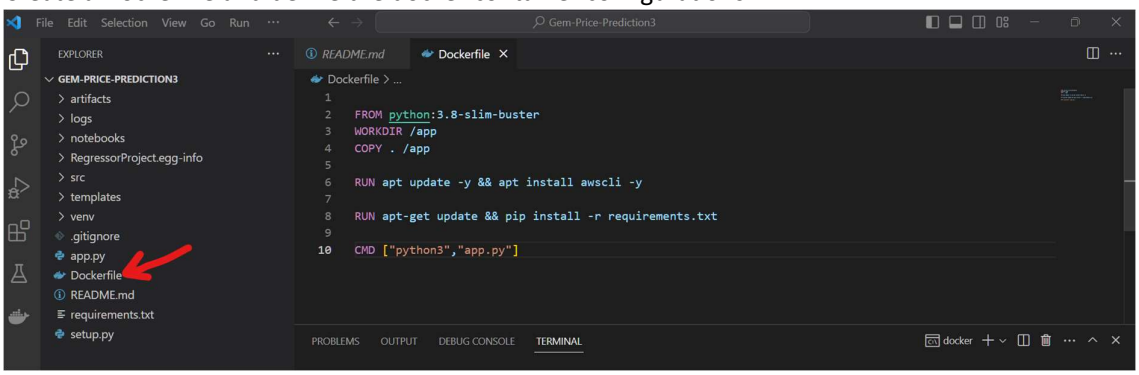
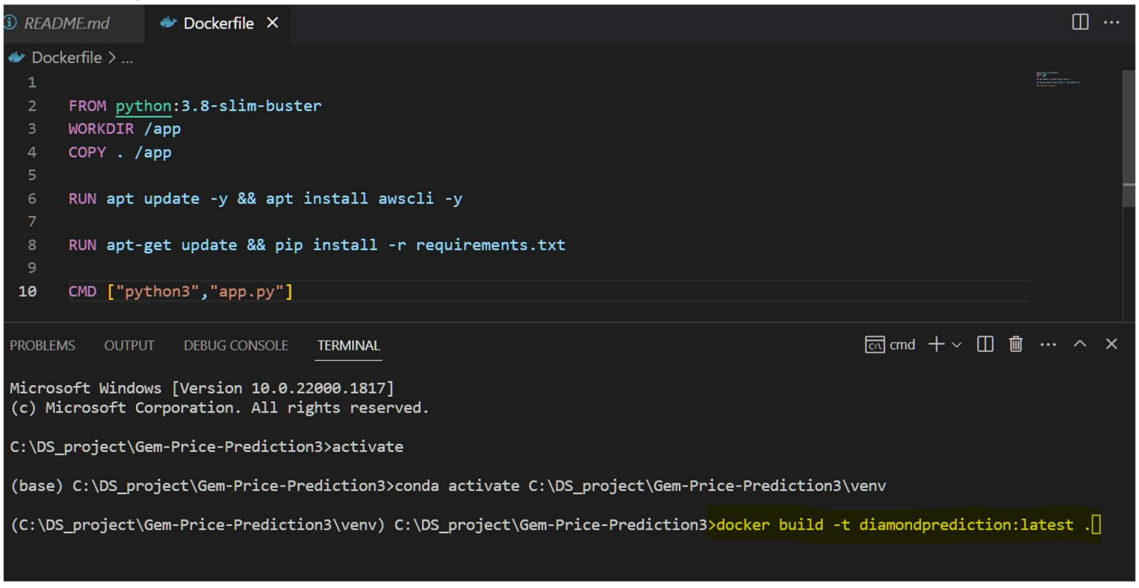
AWS Deployment using GitHub actions, ECR and EC2

1. Create a Dockerfile and define the docker container configurations:



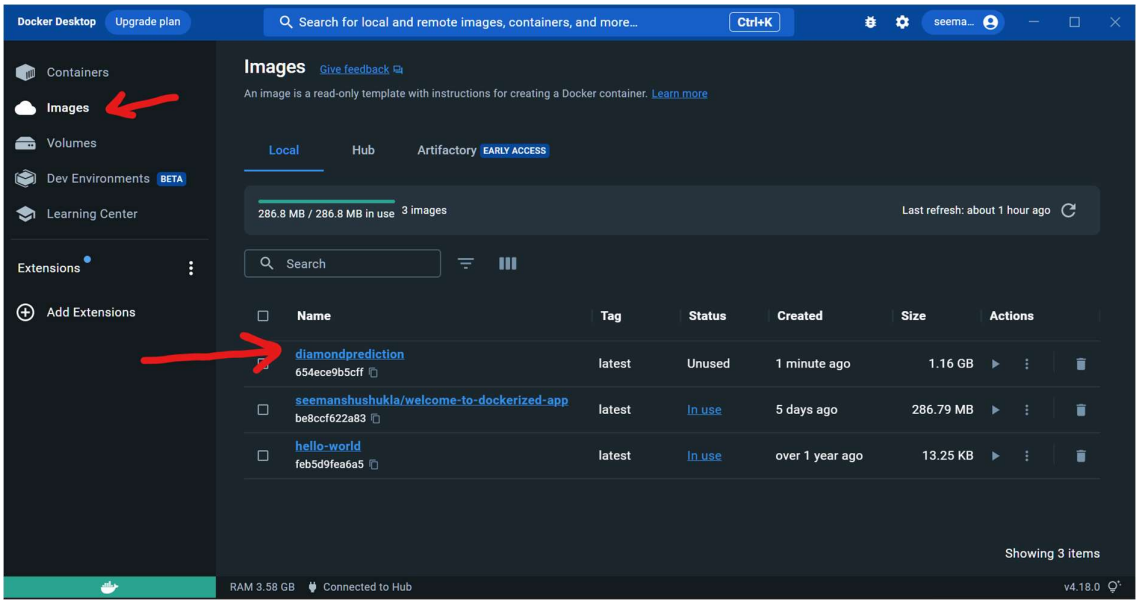
2. **Please note**: Step 2 and 3 are optional steps and can be skipped. This is because later we will be again building docker image and will push it to ECR repository and then this image will be pulled inside EC2 instance and will run as a containerized application or docker container. This entire job is mentioned inside our **workflow file** that is **.yaml file**.

   Build Docker image using "docker build -t diamondprediction:latest ." command where *"diamondprediction"* is user defined image name and *"latest"* represents the tag or version of built image:



   Please note for working with docker one needs to install docker desktop application on their system. Please follow docker installation guide available on https://www.docker.com/products/docker-desktop/

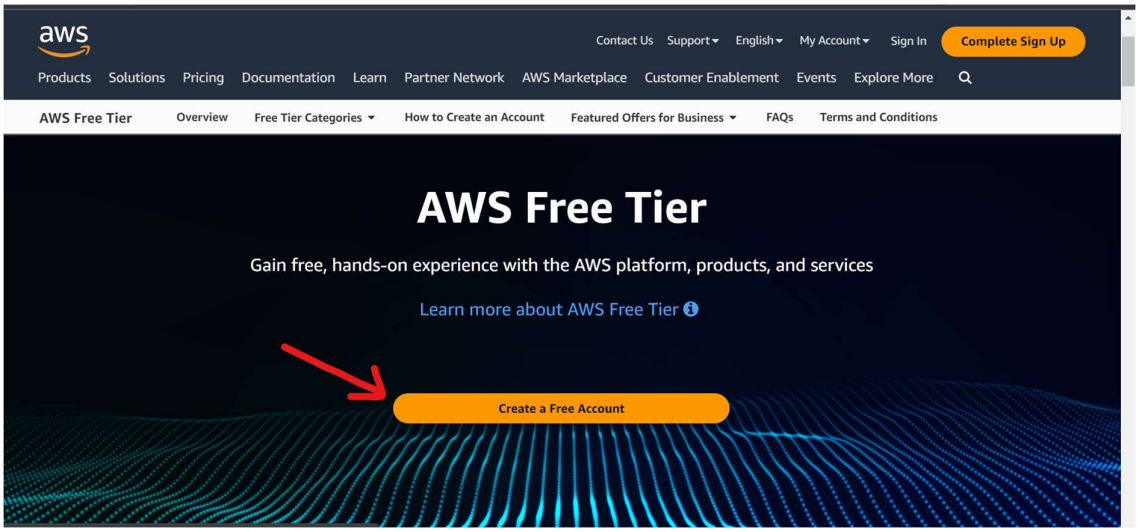
3. For verifying whether Docker image get created or not go to Docker desktop application and check for newly created docker image:



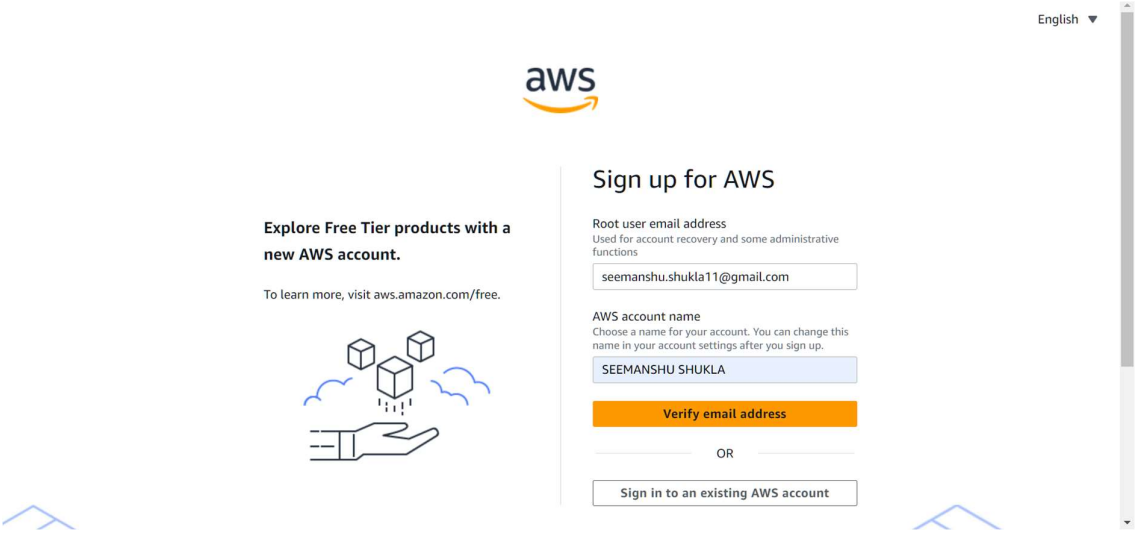
In above snip we can see that *"diamondprediction"* Docker image get created which is of size **1.16 GB.**

4. **<u>AWS account setup</u>:**

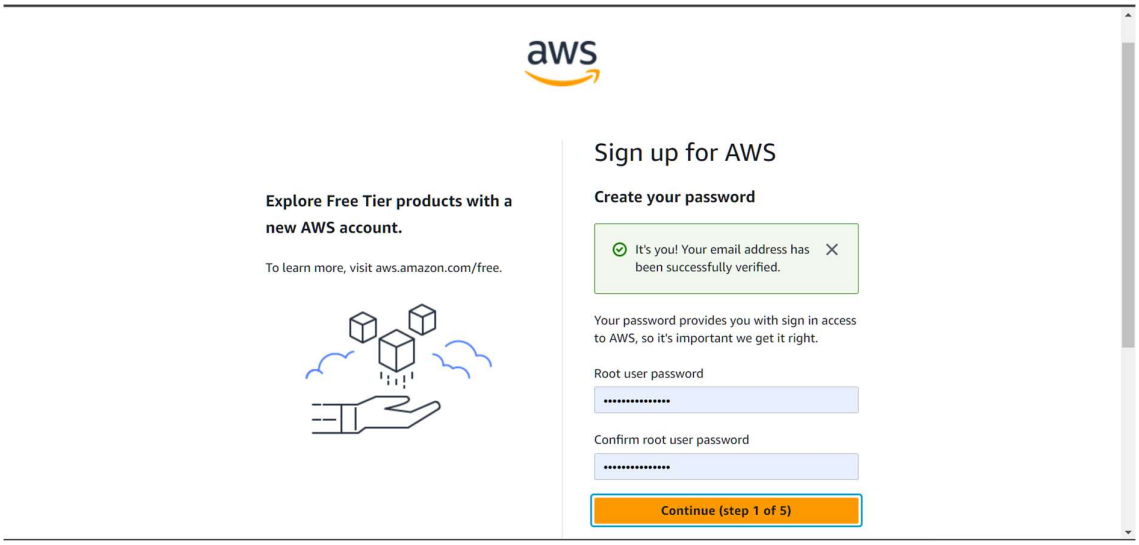
I. Go to AWS website and create a free account:



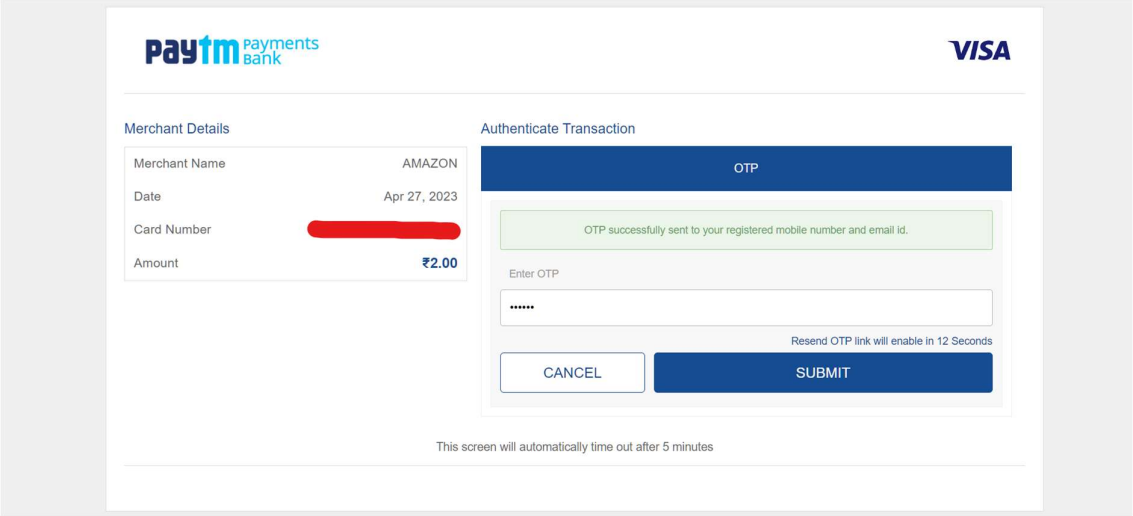
II. Enter details and click **Verify email address** and then verify the email using OTP sent to email address:



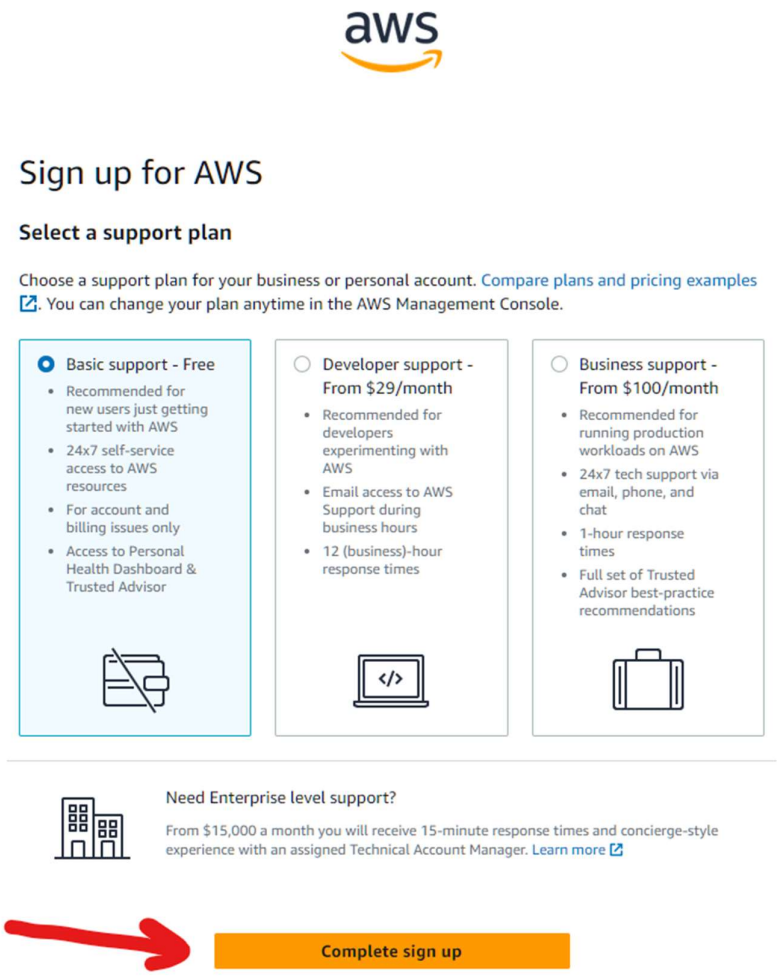
III. Complete all the prompted 5 steps post which we will get redirected to Authentication page:



IV. To verify our personal identity AWS will ask for debit or credit card. AWS will charge INR 2 for the same:

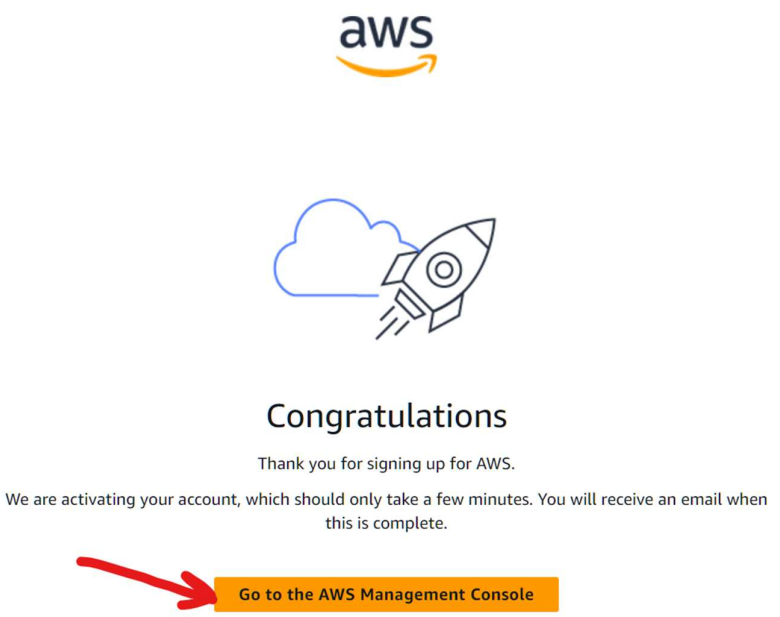


V. For learning purpose we will be selecting **Basic support – Free** plan . After this click **Complete sign up**:

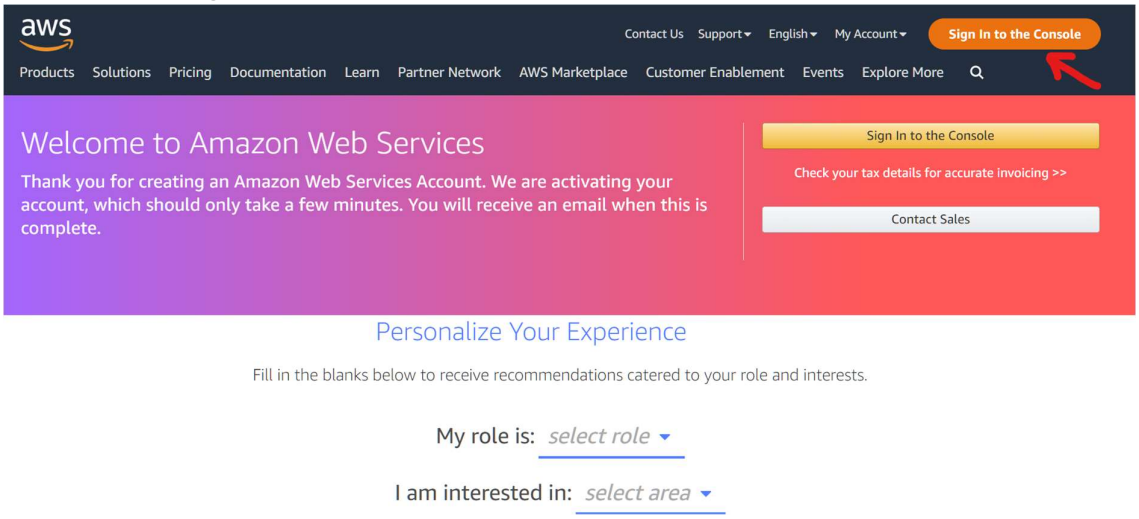


VI. Please note that for the next 12 months, we will be provided with free access to all AWS services within the limits of the Free Tier. Follow the free tier link to know more

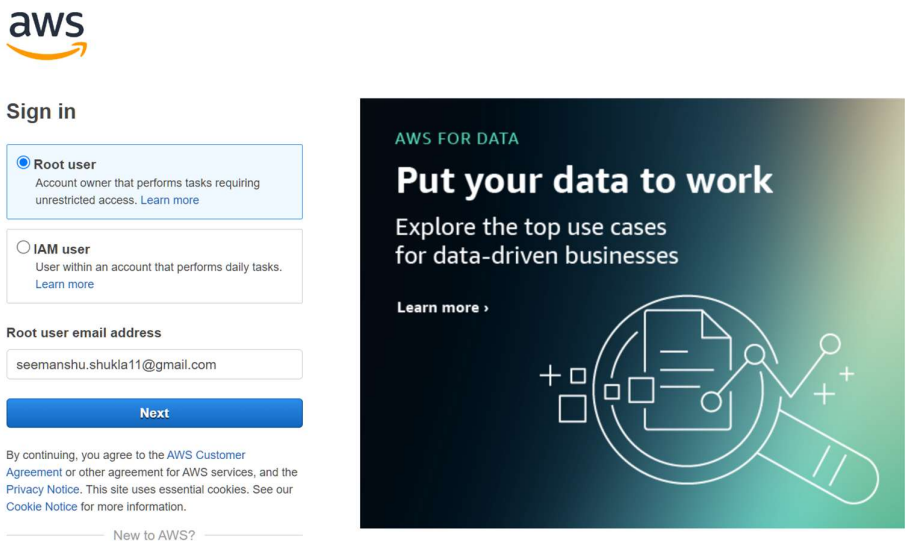
Click on **GO to the AWS Management Console**



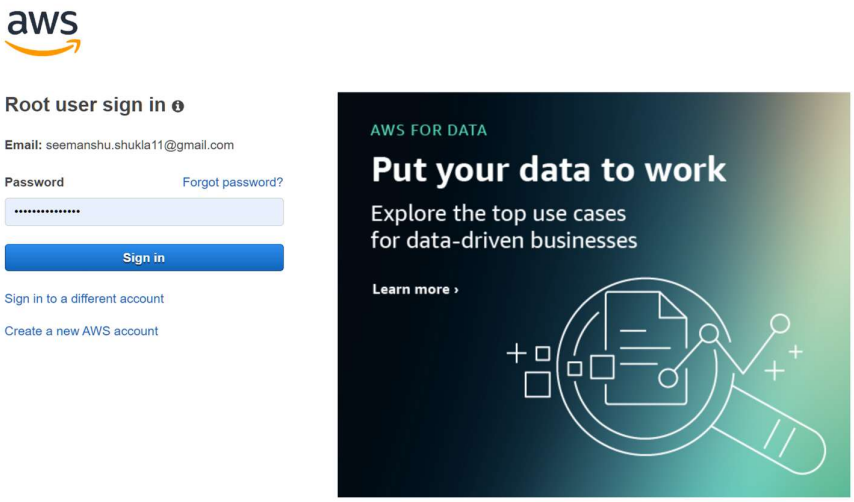
VII. Click on **Sign in to the Console:**



VIII. Select **Root user** and enter the email ID used while setting up the AWS account:



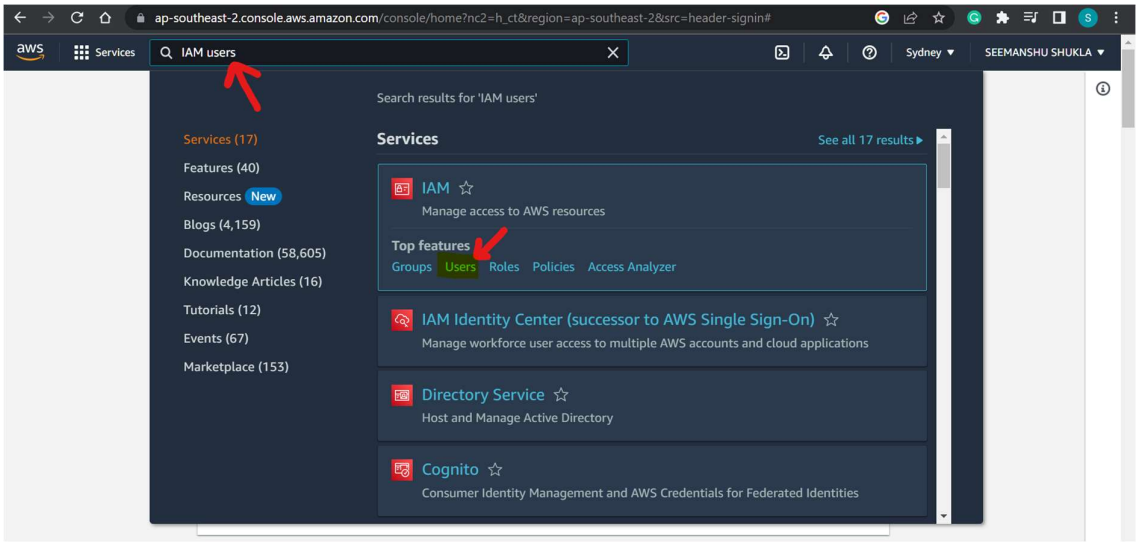
IX. Enter Password and click **Sign in** button:



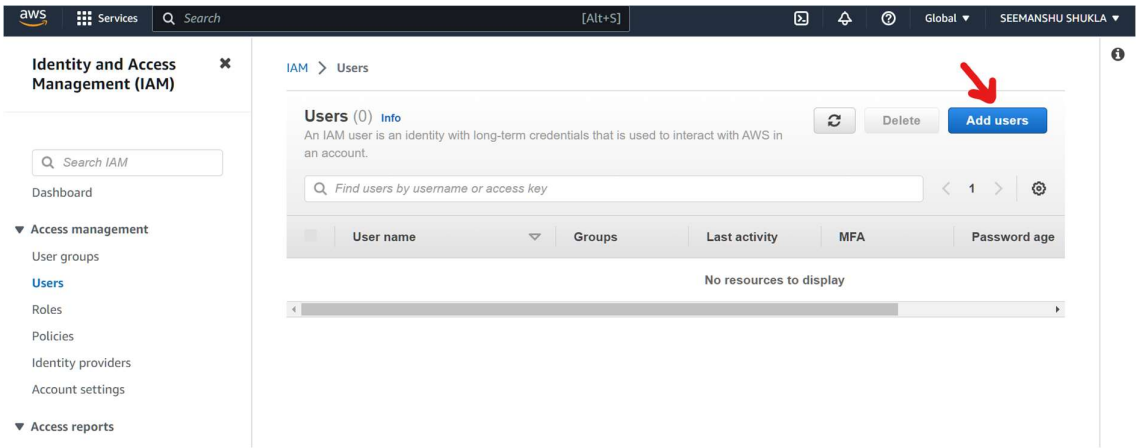
5. <u>IAM user Setup</u>:

Once we are inside our AWS console we will start with creating an IAM user. IAM stands for **Identity Access Management**. Under this we as an IAM admin (since we are using a root account) will be creating a user who will be given some set of privileges using which this user can carry out the deployment job in AWS. This user will be set with some credentials that later will be used to configure it(user) while initiating the deployment to server.

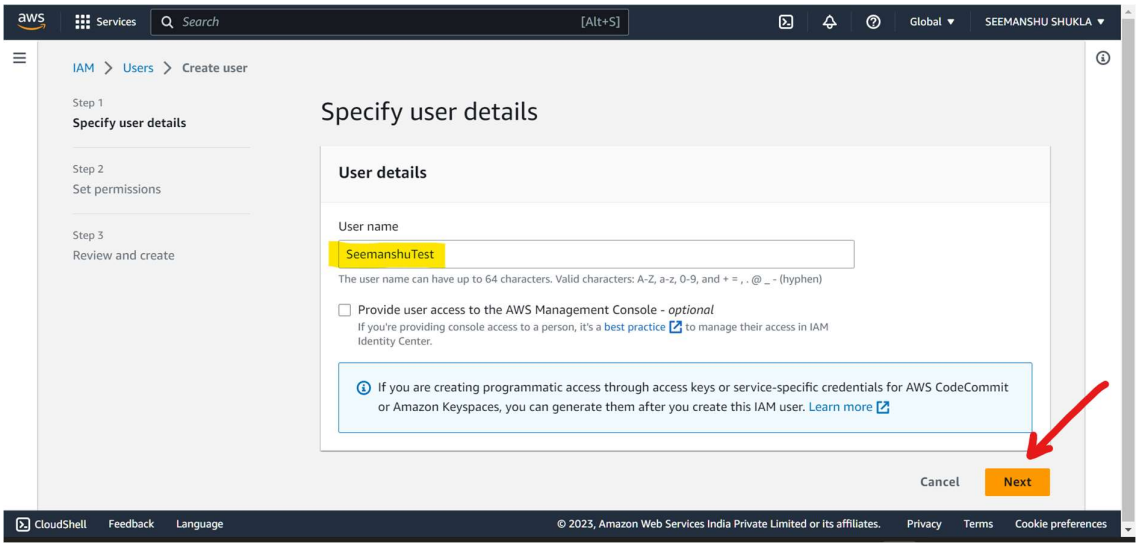
I. Search for **IAM** service in the AWS console:



II. Add the user to whom we want to give access to ECR2 instance and ECR repository to carry out the deployment:

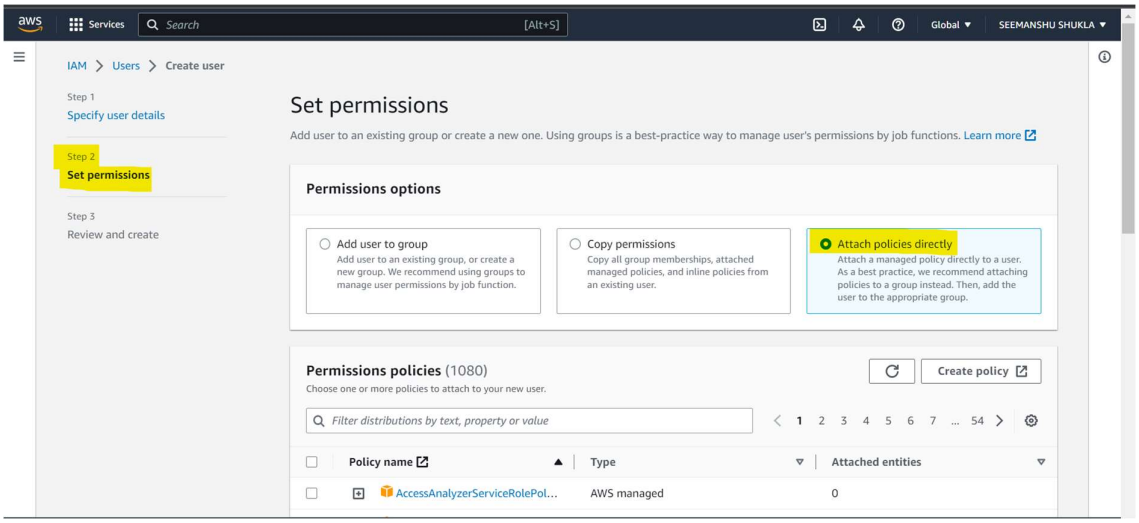


III. Enter the **User name** (this is user defined) and click **Next**:



IV. Now we will be prompted with **Set permissions** page where we will defining the set of privileges to be given to the user for carrying out the deployment:

→Select **Attach policies directly**:

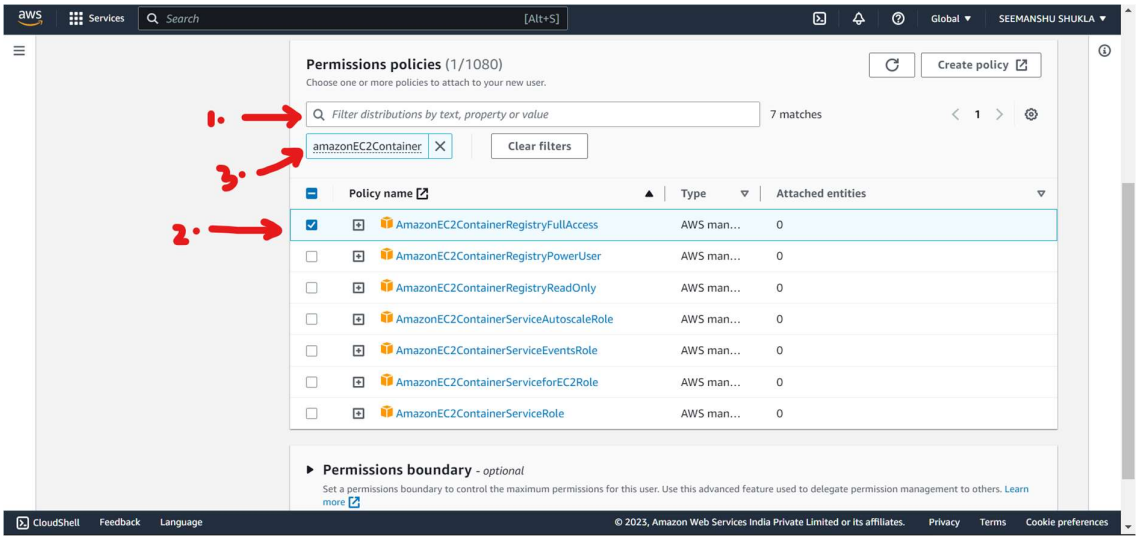


→Follow the following steps to add necessary roles:

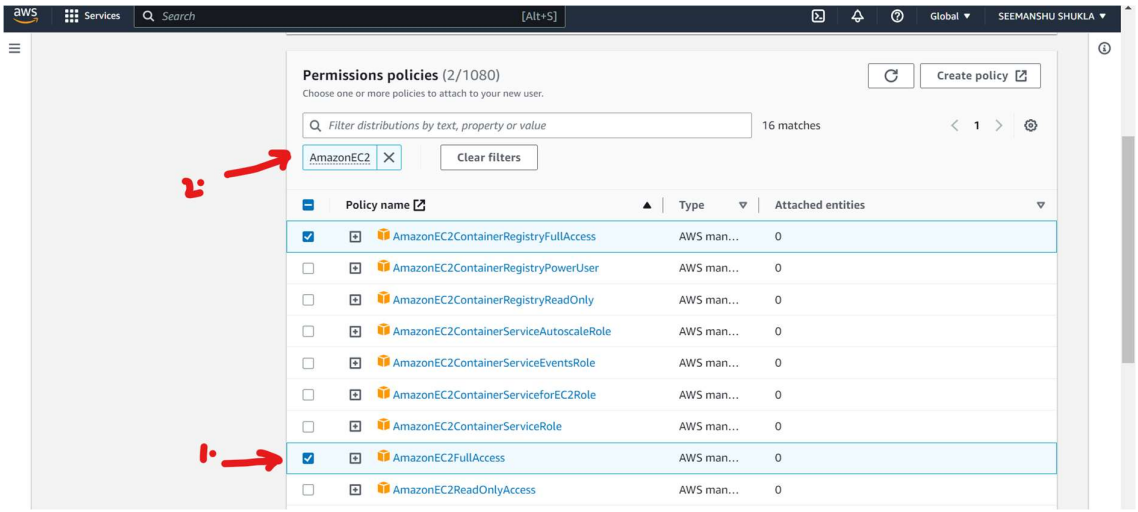
1)Search **amazonEC2Conatainer** and hit enter

2)Select the check box that says full access

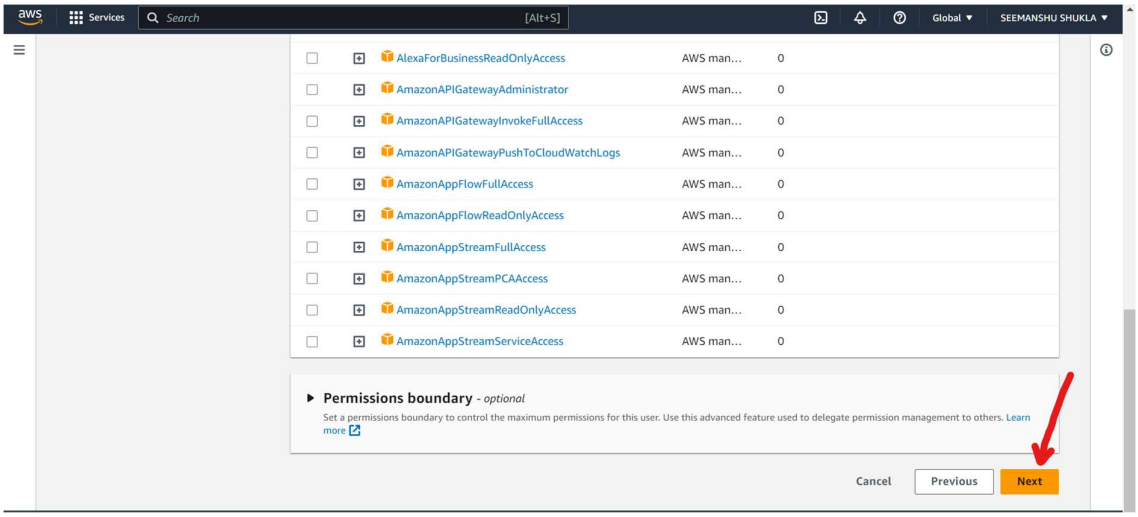
3)Use the cross(X) icon to remove this filter and search for **AmazonEC2**



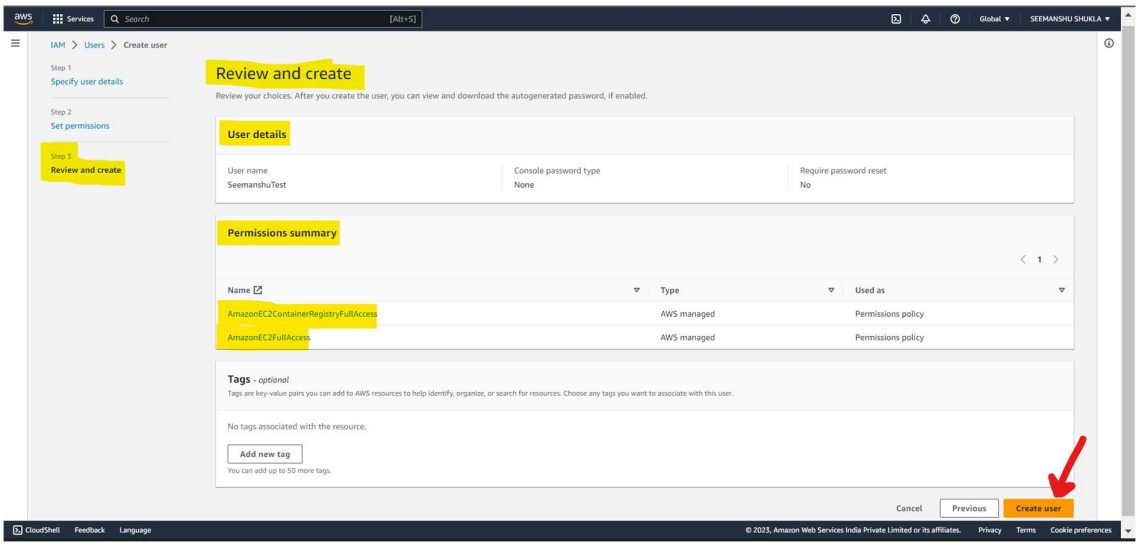
→ Select the **amazonEC2FullAccess** and then remove the filter using cross(X) icon



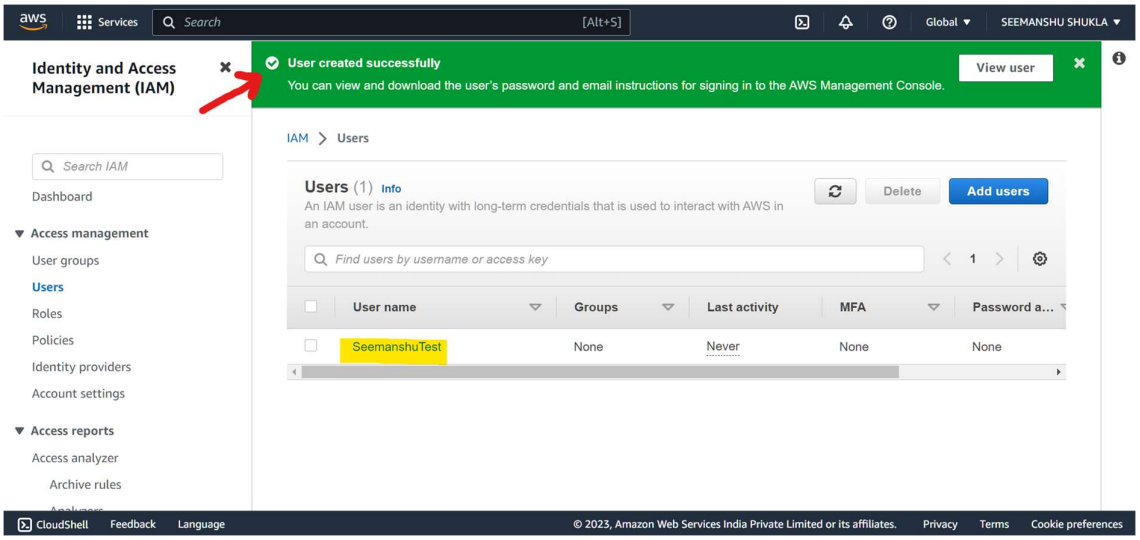
→Scroll down the page and click **Next**:



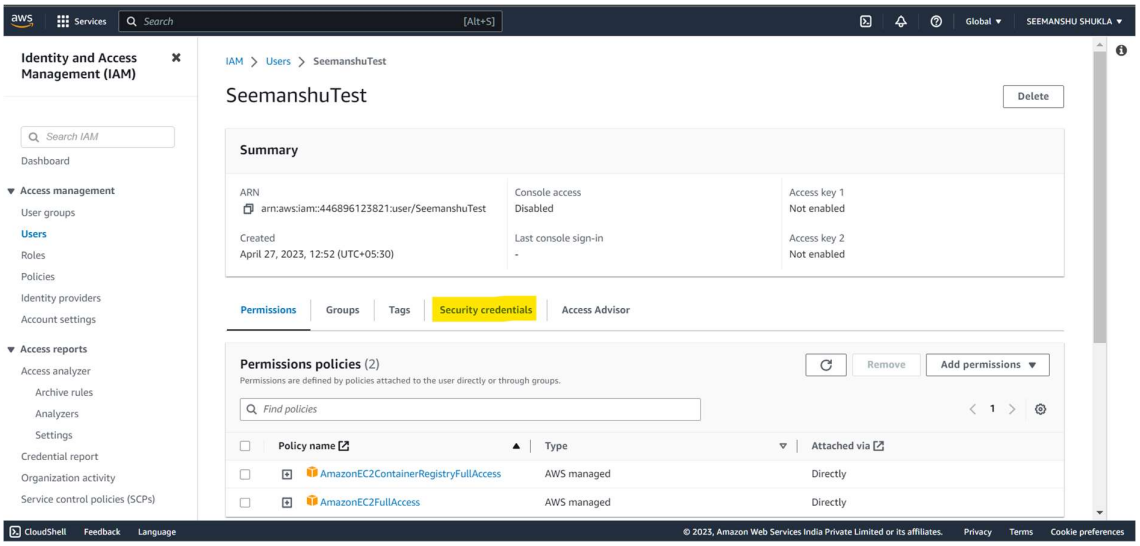
V. Review the details and click **Create user**. Here, we can clearly see the roles that were assigned to our user: SeemanshuTest:



VI. User is now created successfully. Open the highlighted newly created user:

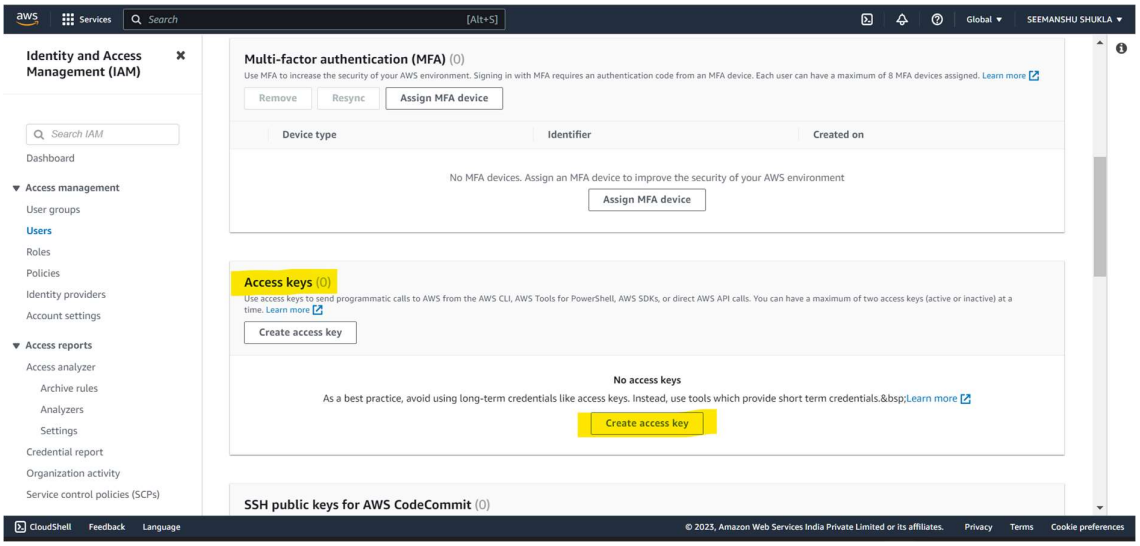


VII. Go to **Security Credentials**:



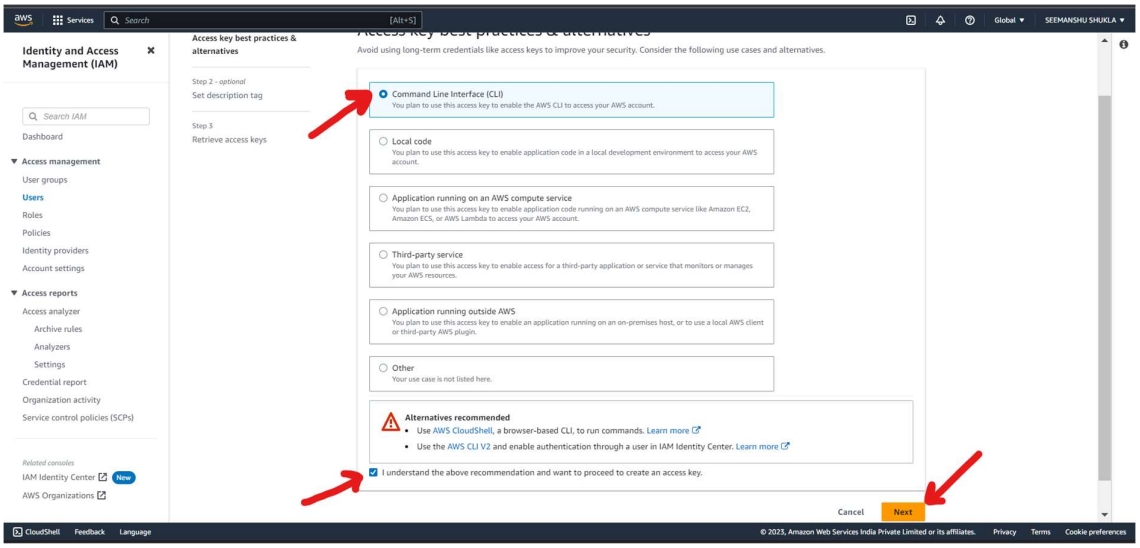
VIII. Now here, we will be creating an access key. This will be privileged key that will be basically used for carrying out our deployment by the newly created user.

Go to **Access keys** and click **Create access key** which will be having 3 sub parts:

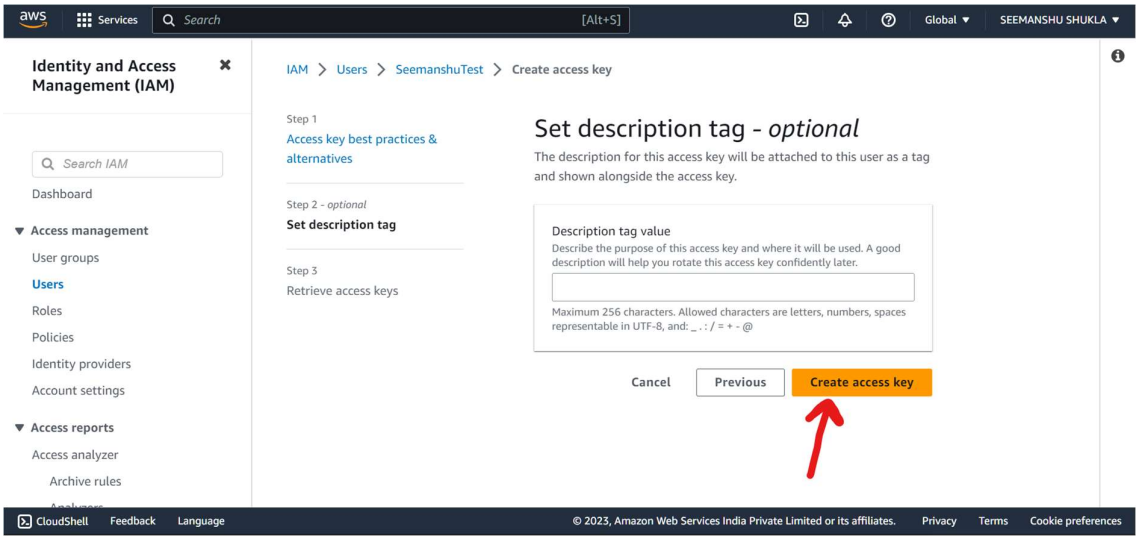


IX. Select **CLI**, radio box. Also, select the disclaimer check box at the bottom and click **Next**.

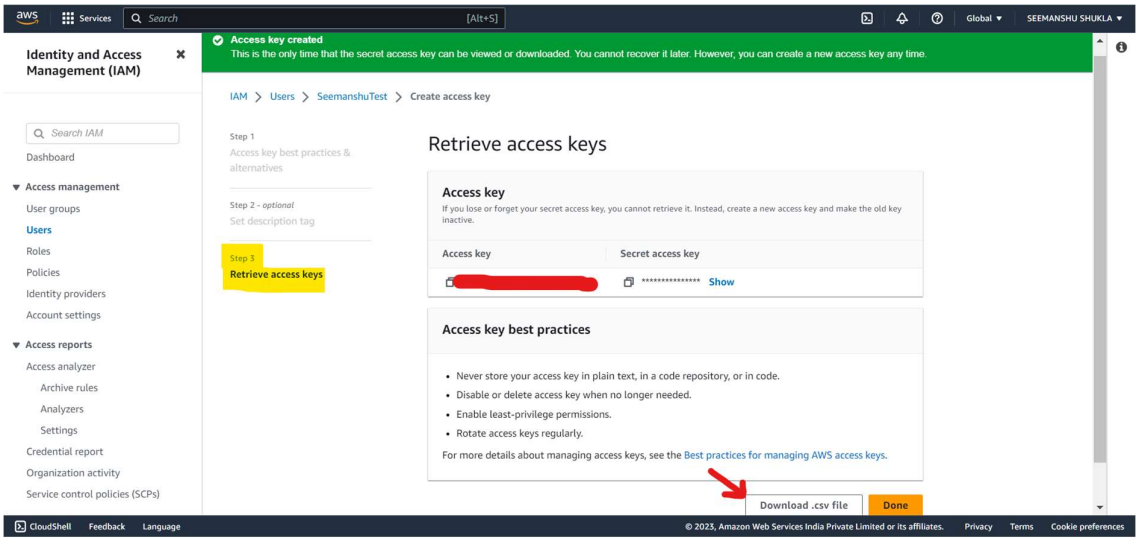
Recall that while creating Dockerfile we installed AWS CLI. It was installed for this purpose so that we can carry our deployment using access key in AWS CLI:



X. We can skip description part and click **Create access key**:

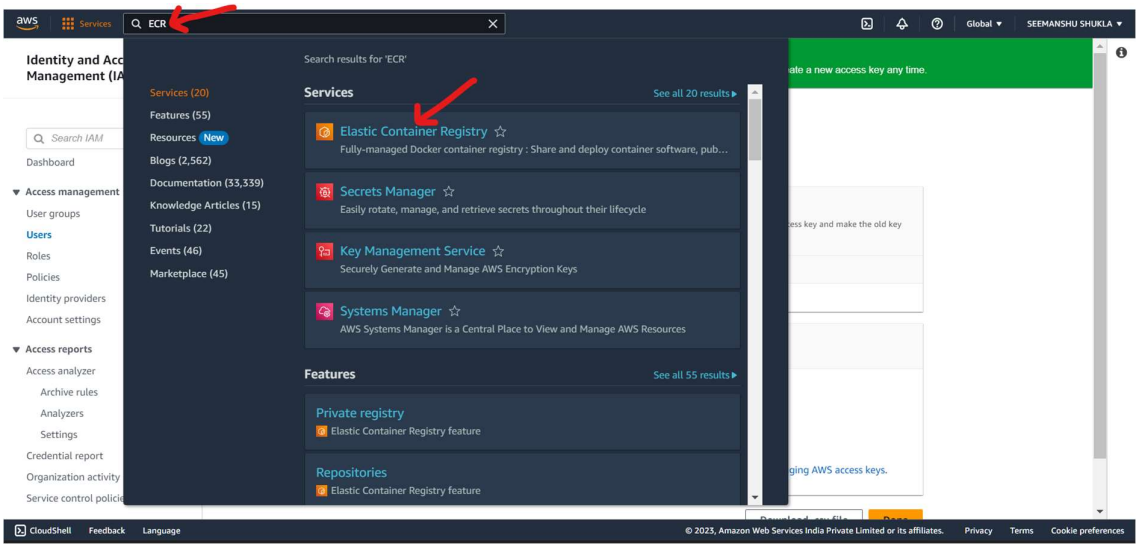


XI. Download access key as .csv file which later will be used to interact with EC2 instance and EC2 Repository via AWS CLI:

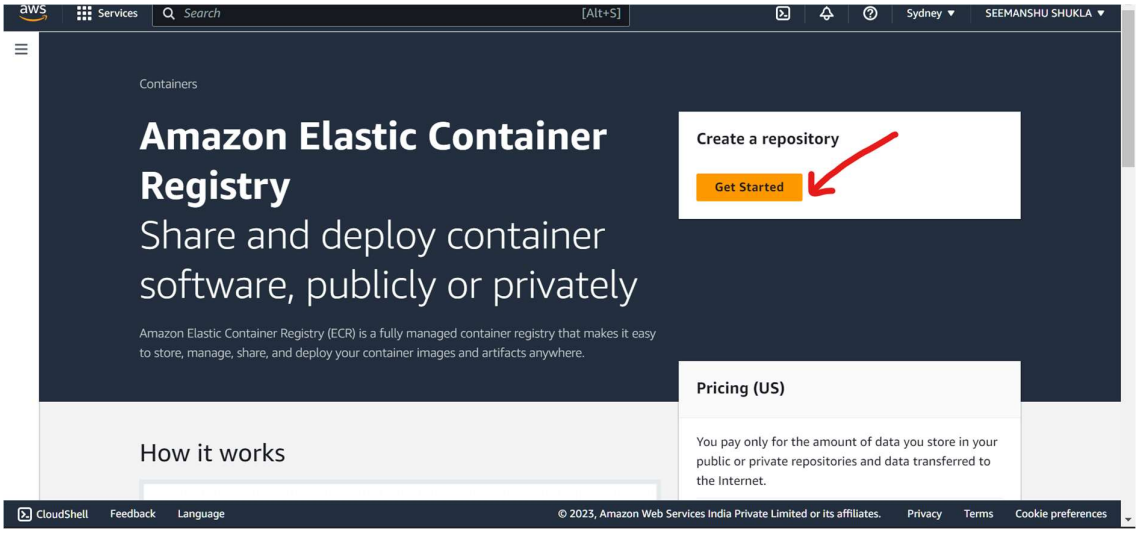


6. ECR (Elastic Container Repository) setup:

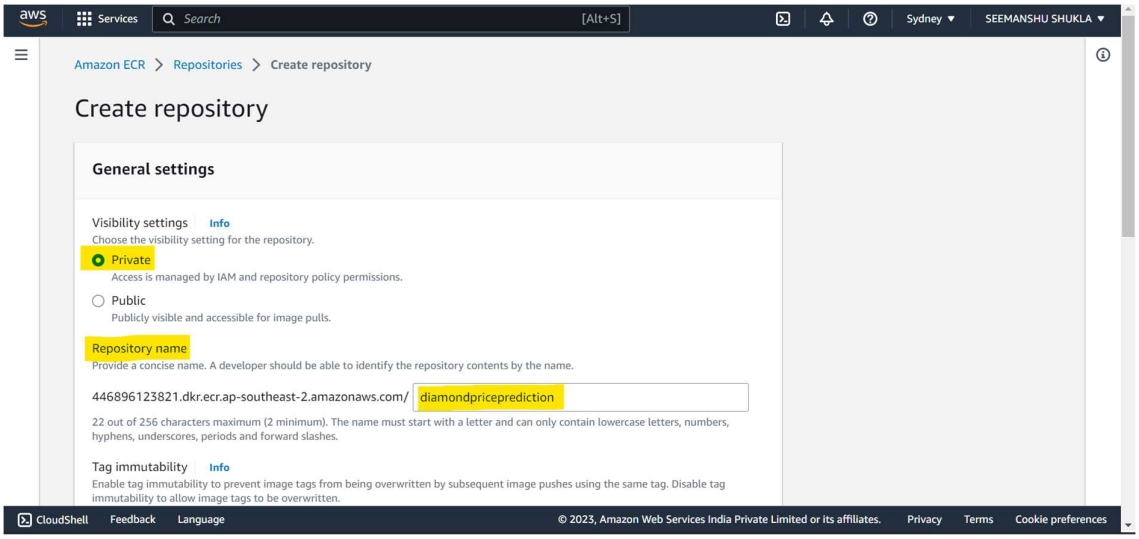
I. Search for **ECR** and open it. In ECR one can Share and deploy container software, publicly or privately:



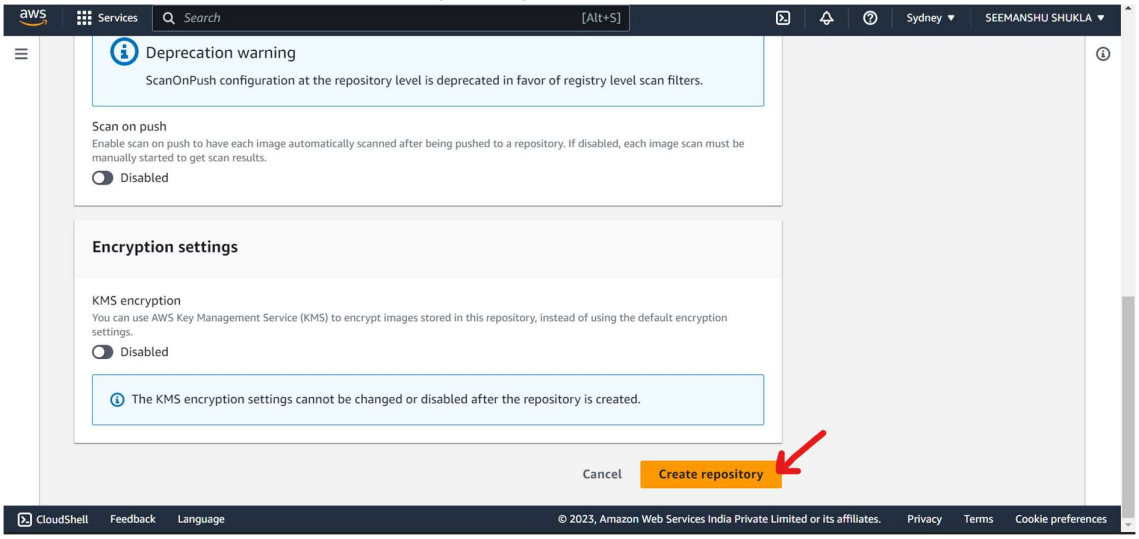
II. Click **Get Started**:



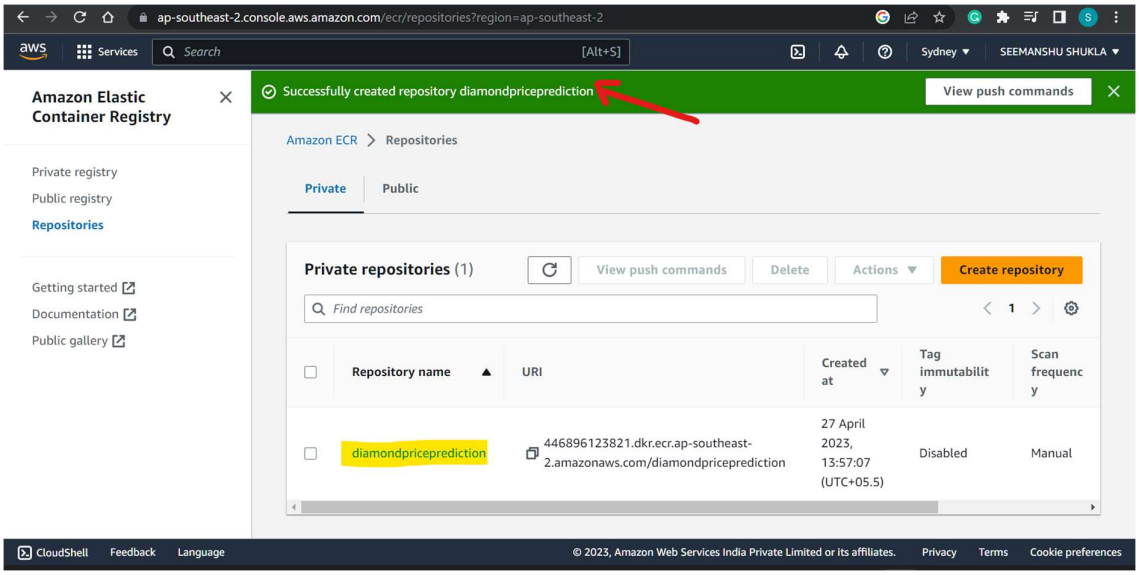
III. Set Visibility as **Private** to make a private repository and set a repository name which is a user defined name:

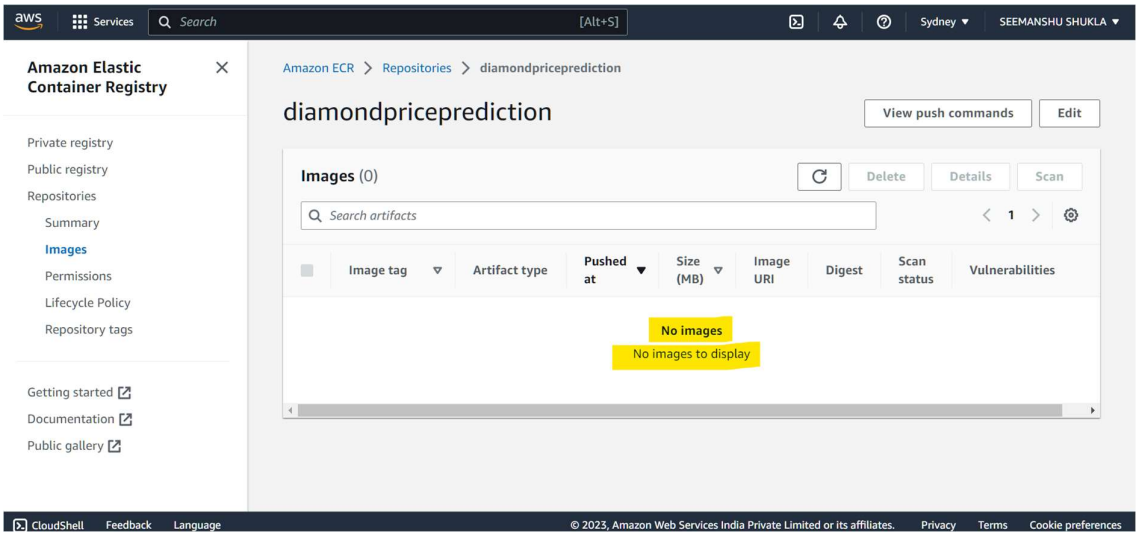


IV. Scroll down and click **Create repository**:



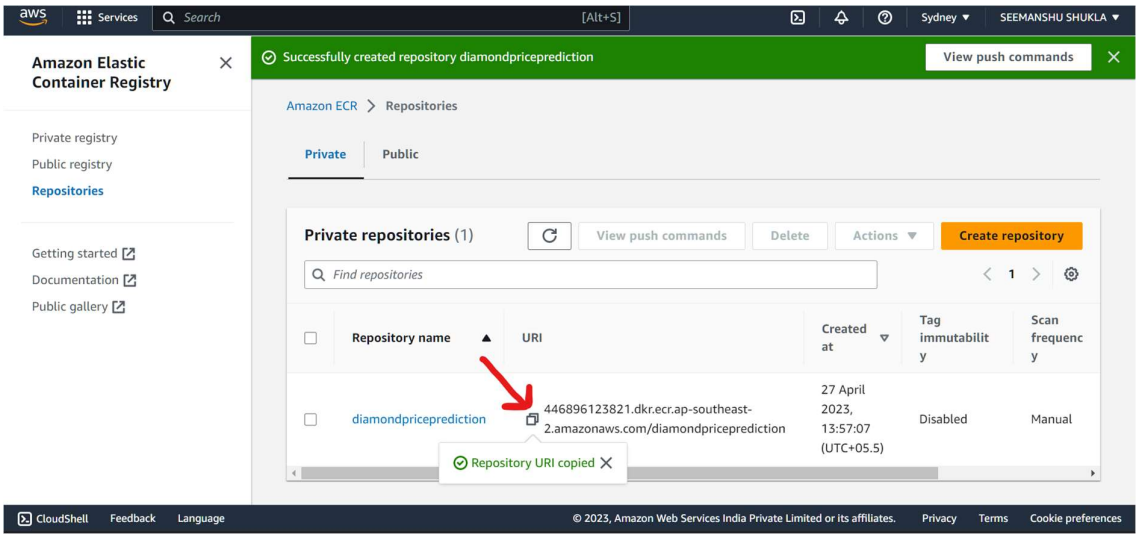
V. Repository will get created. If we open this repository we will not find any image (docker image) since we are yet to build our docker image. The job for building docker image and pushing it to ECR is defined under .yaml workflow file. This file will be executed when we initiate our deployment.





VI. Go back and copy the URL for Private ECR repo.
Copied URL:
446896123821.dkr.ecr.ap-southeast-2.amazonaws.com/diamondpriceprediction

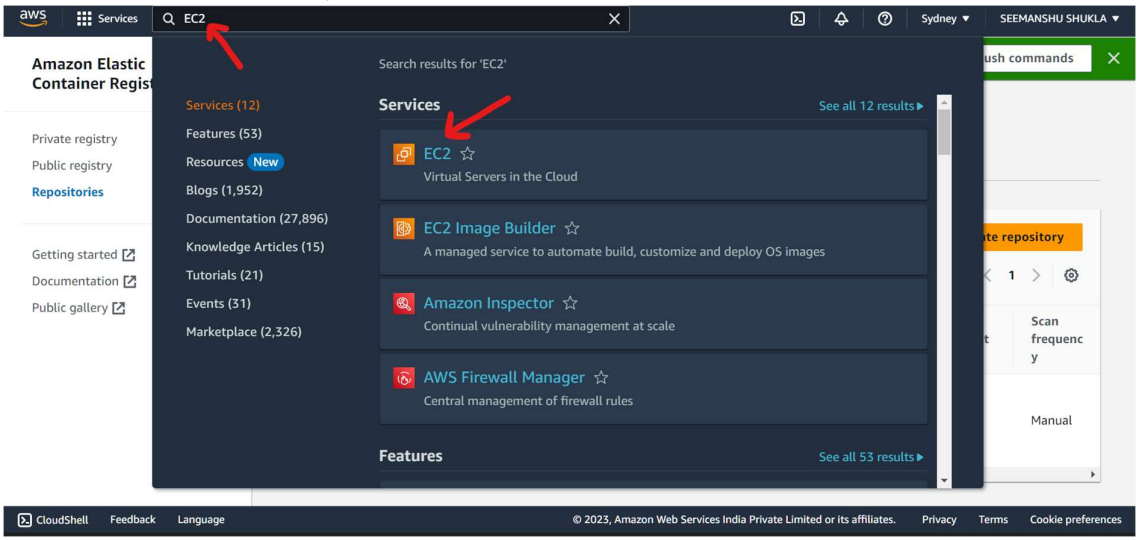
This url will be later used for setting secrete variable in GitHub in order to carry out deployment.



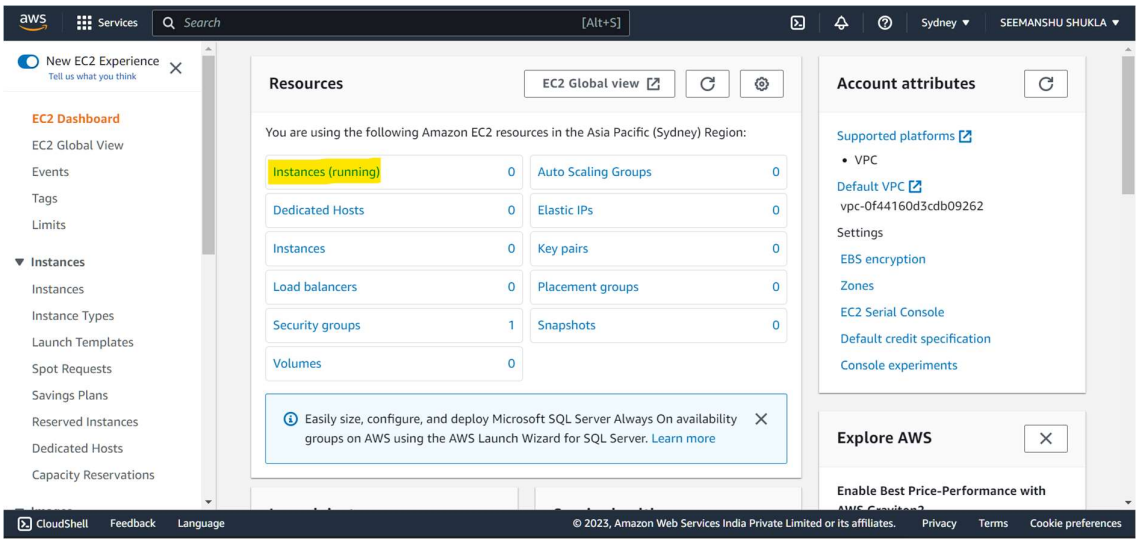
7. **<u>EC2 instance setup</u>:**

It is Virtual Server in the cloud (Linux based machine called Ubuntu will be used for current deployment).

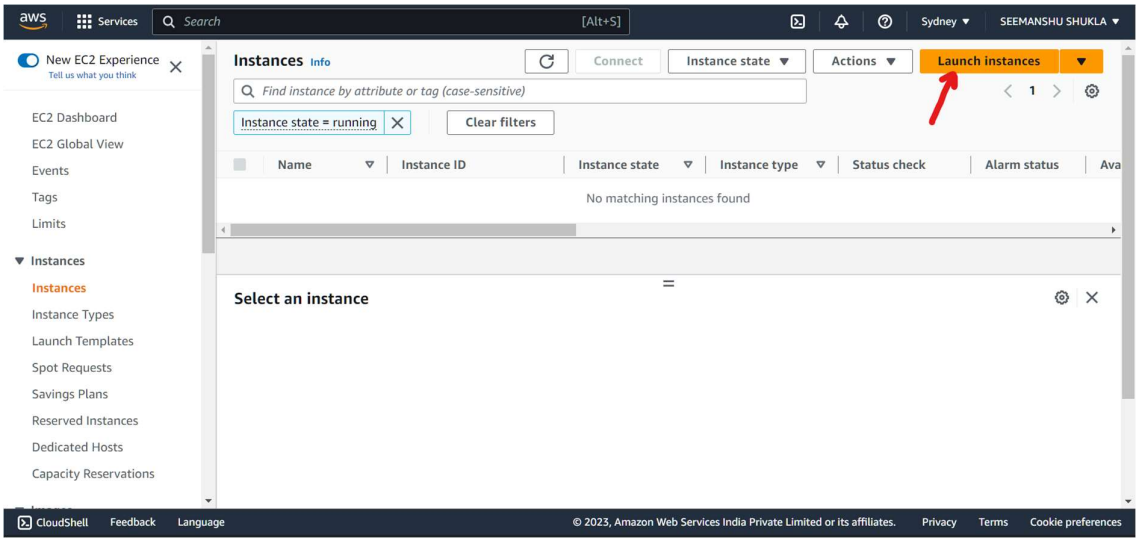
I. Search for **EC2** and open it.



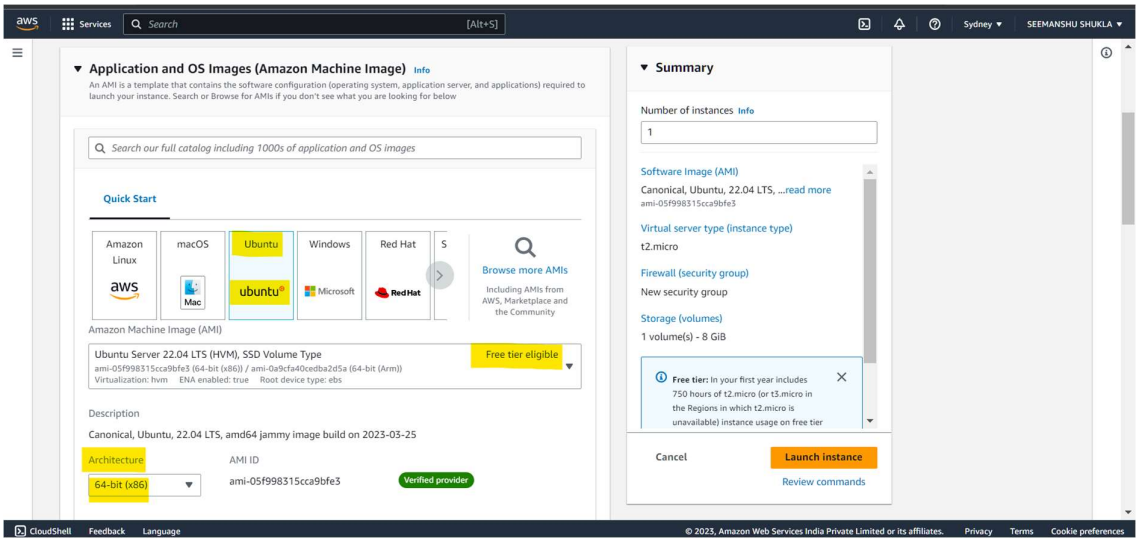
II. Click **Instances(running)** option:



III. Click **Launch instances**:

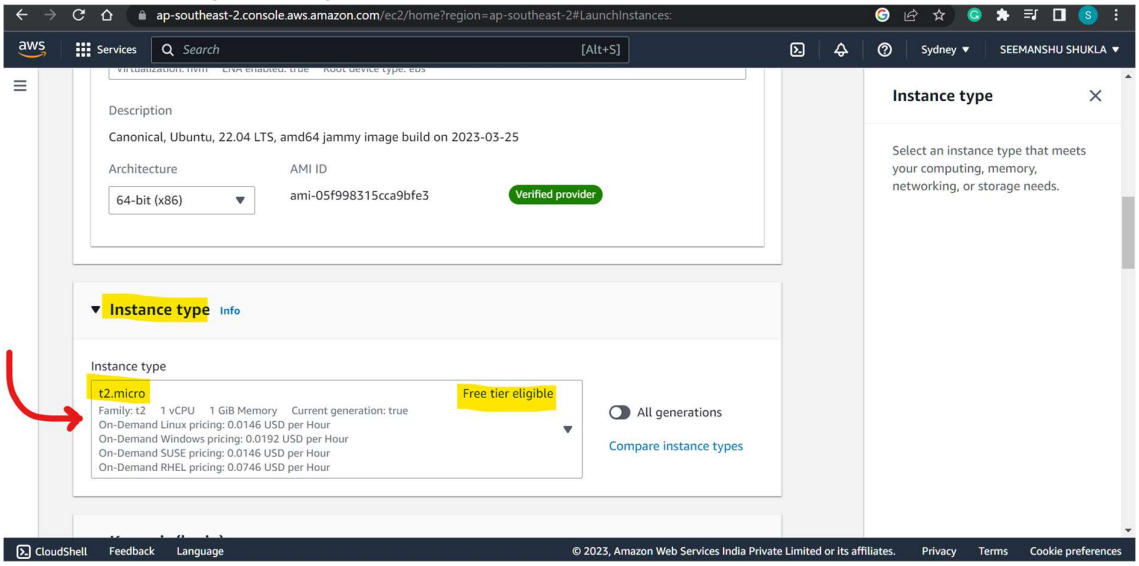


IV. Scroll down and select the OS image – **Ubuntu** that is OS based on which server should be build. For learning purpose to make sure that we are not getting charged anything extra make sure to opt for **Free Tier** images. Select **Architecture** as **64 bit**.

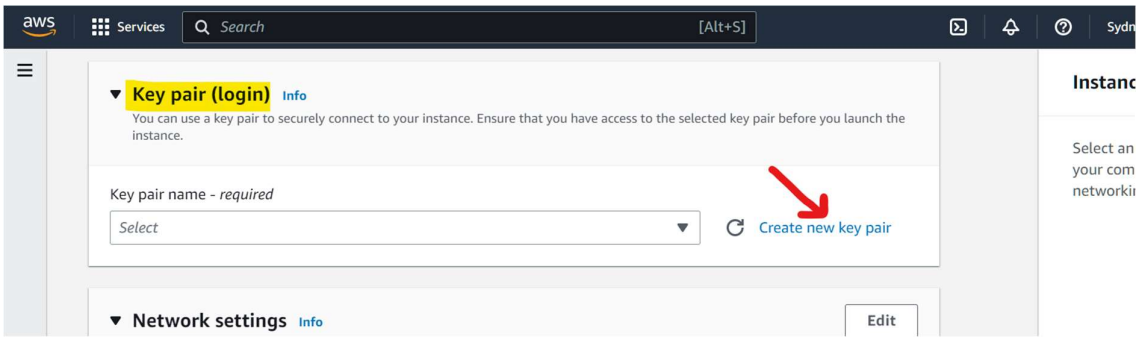


V. Scroll down and select the **Instance type**. For learning purpose choosing **Free Tier** version of it

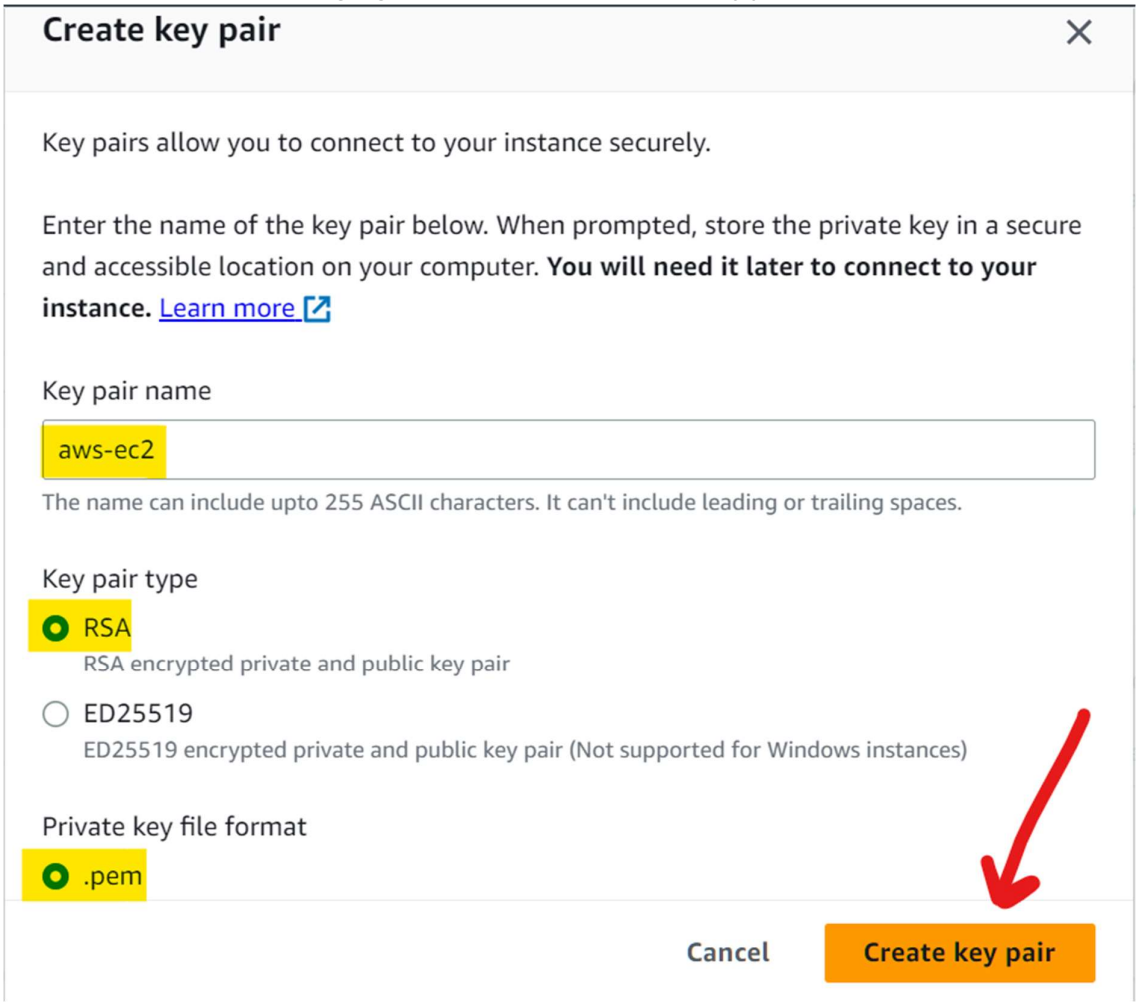
Instance type selects an instance type that meets your computing, memory, networking, or storage needs.



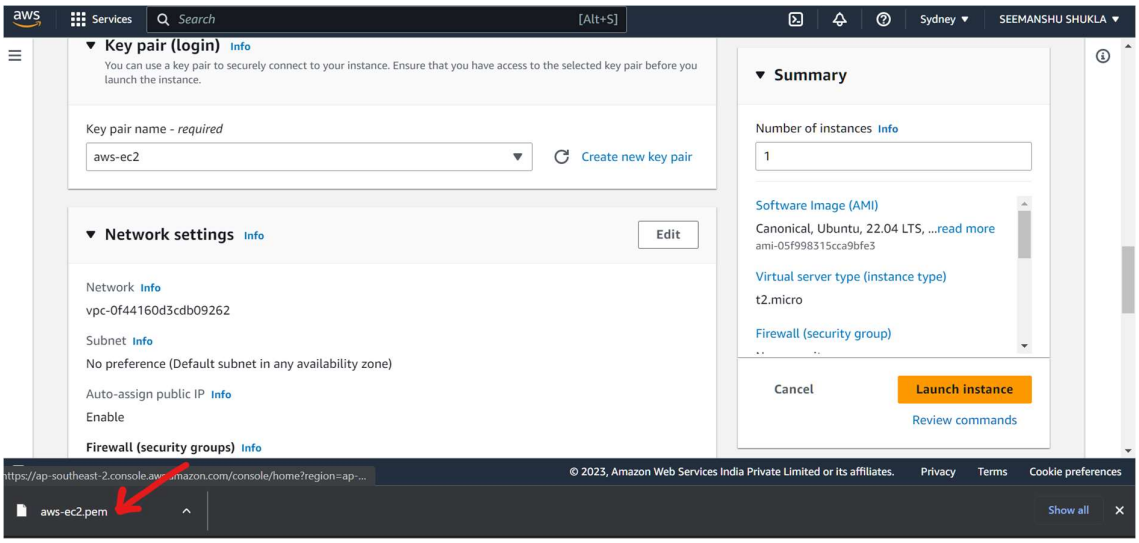
VI. Scroll down and click **Create new Key pair** which later can be used to securely connect with the instance.



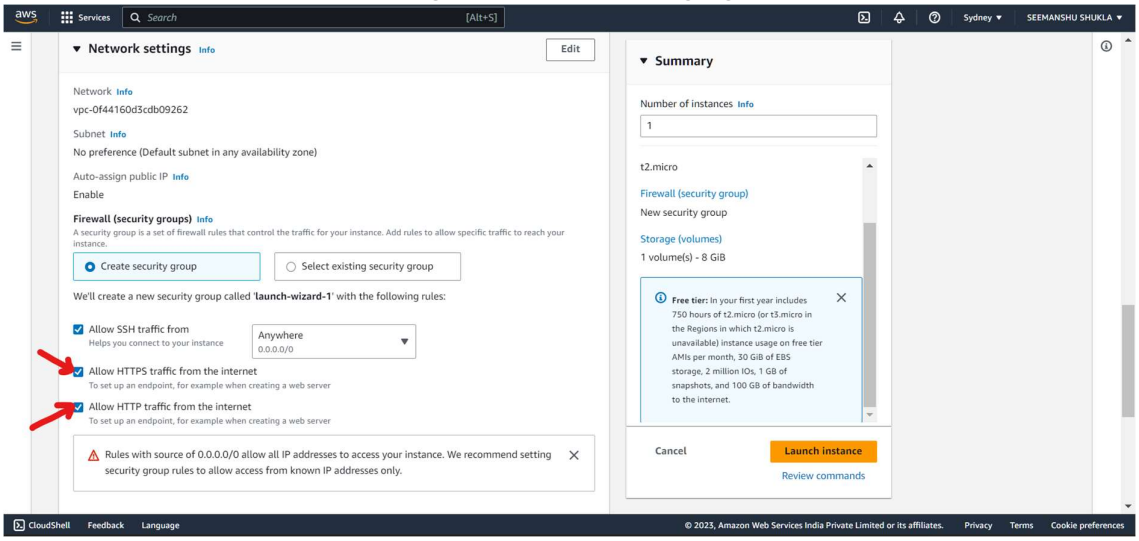
VII. Enter the details as highlighted below and click **Create key pair**:



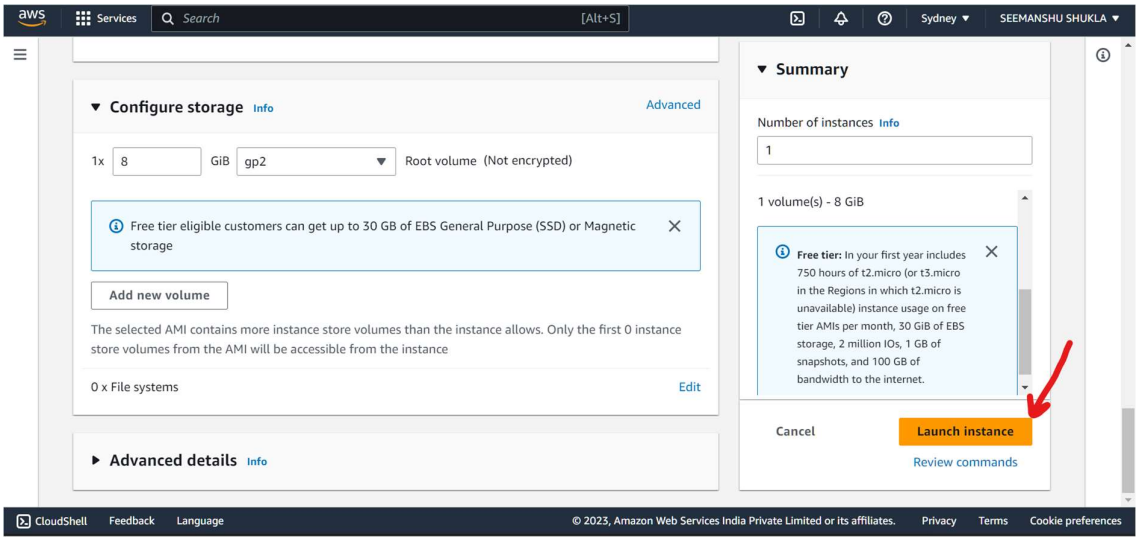
VIII. A file will be downloaded once key is successfully created. This downloaded file then need to be attached as to some 3rd party console using which we can establish a secure connection with our EC2 instance.
For E.g. MobaXterm is the 3 rd part console based application that can be used to securely connect with our EC2 instance. For this while configuring instance over MobaXterm we will be asked to attach the secure key file, where we will be attaching the current .pem file.



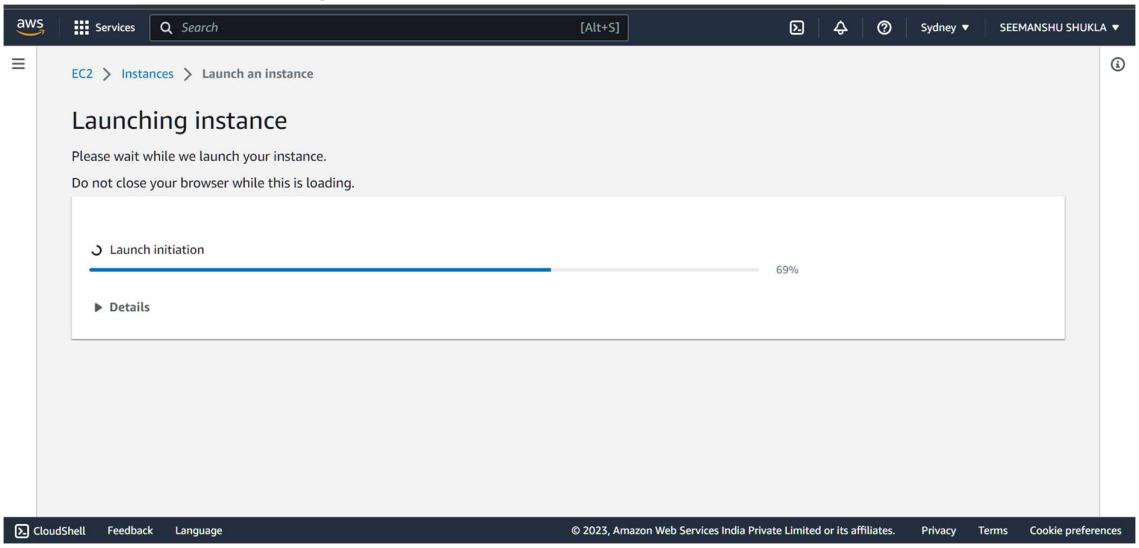
IX. Scroll down to **Network settings** and check all the highlighted check boxes:



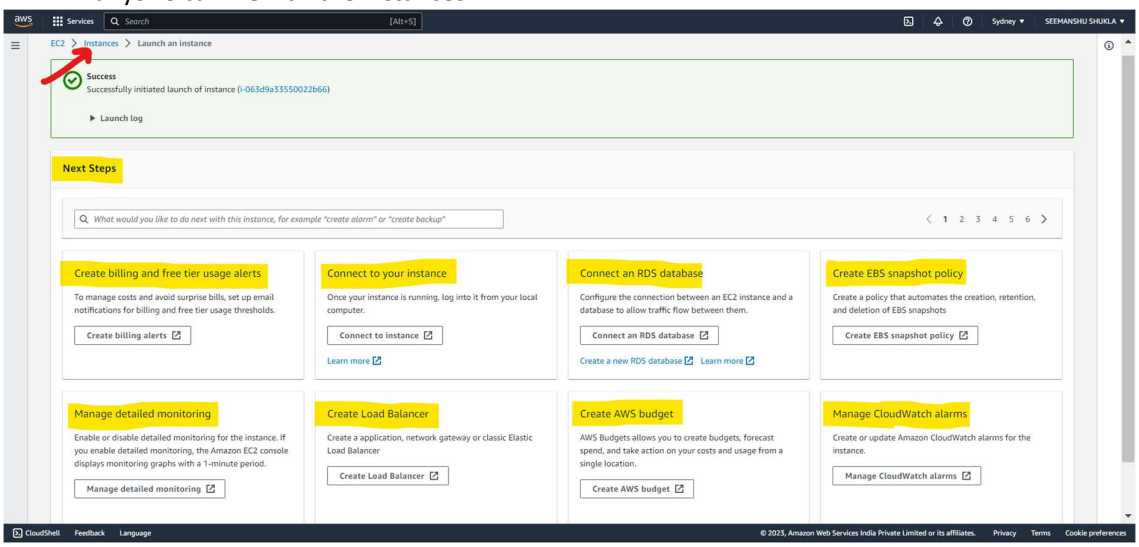
X. After this click **Launch instance**:



XI. Launch initiated. Make sure that we are not closing this tab or browser while instance in launching.

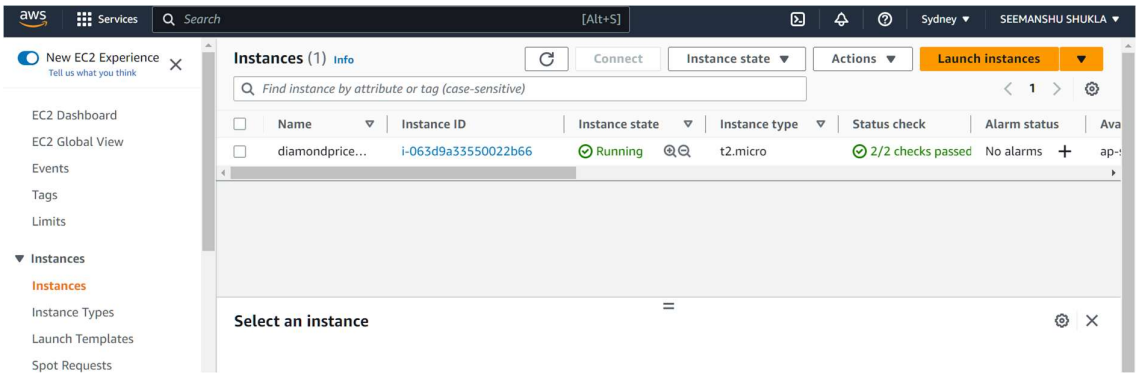


XII. Now one ideally needs to perform all the mentioned Next steps. Before that lets check our newly created instance by clicking on **instance** option. Also, if we scroll down a bit there is an option with the name **View all instances** using which as well anyone can view all the instances

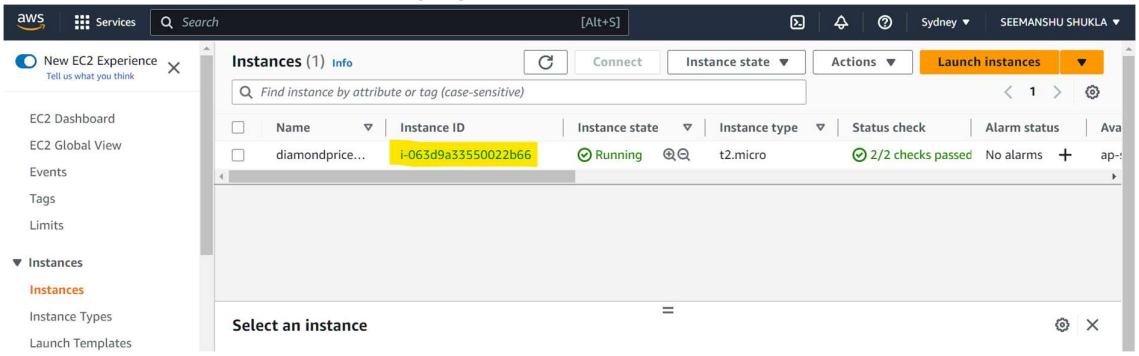


XIII. Now, we can see our instance with all the details in tabular form.

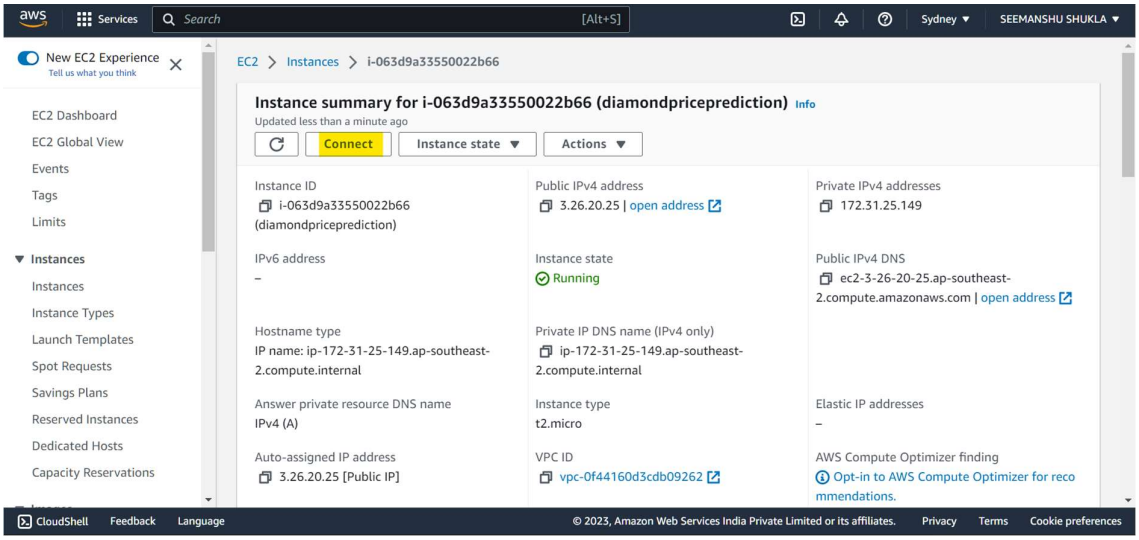
Once **Instance state** and **Status chec**k is turned into **Green** then it means that our EC2 instance has been set successfully and is now up and running.



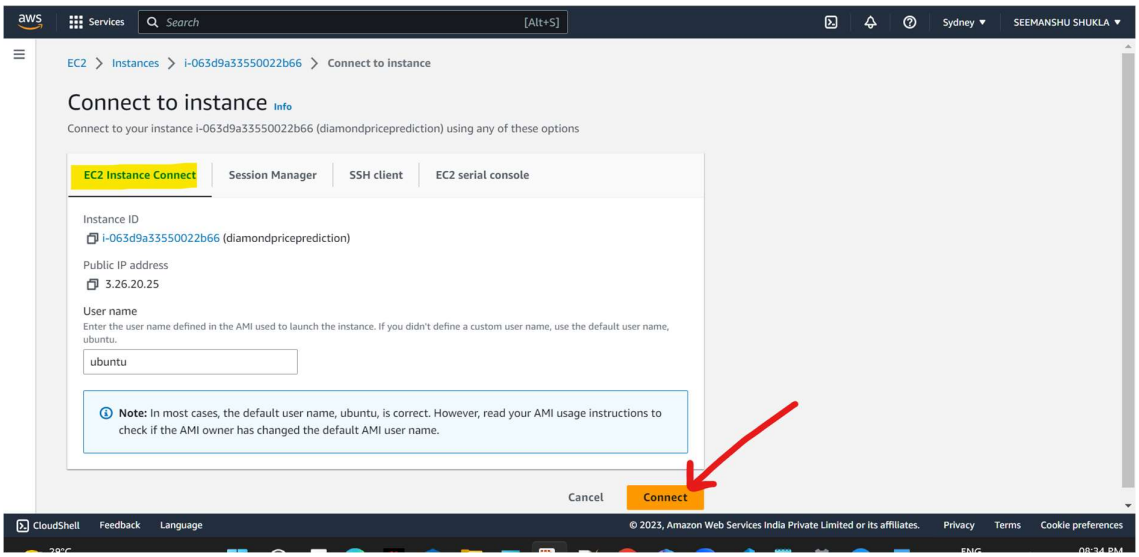
XIV. Click on the **Instance ID** as highlighted below:



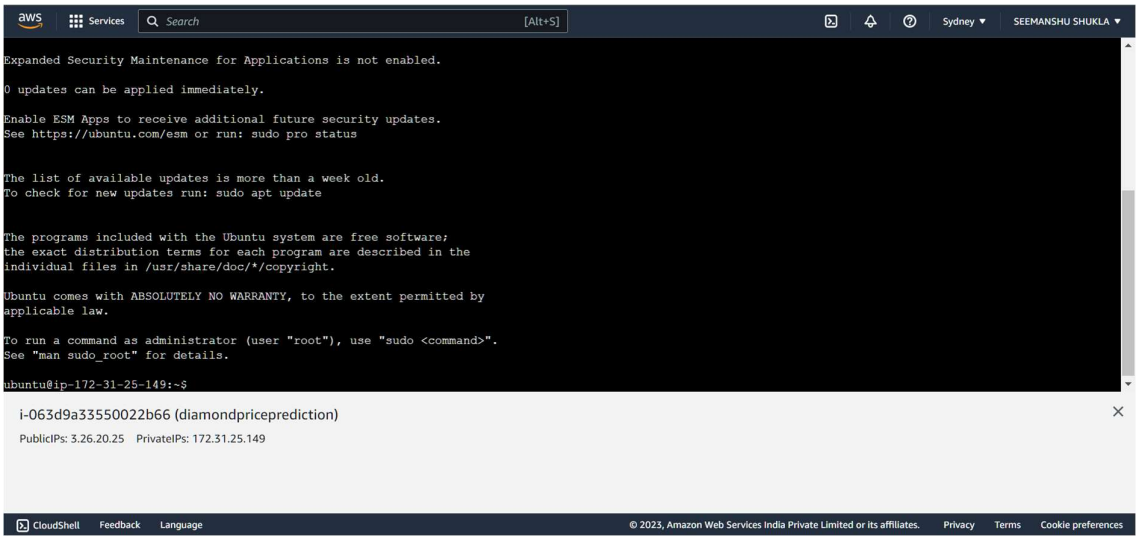
XV. Use **Connect** option to connect with the instance (server):



XVI. Under **EC2 instance Connect** click **Connect** option:



XVII. Now our instance/server will be launched which will be a linux based machine. We can access this machine using opened command prompt. Here we will be doing all the necessary configurations and installations. This command prompt is of EC2 instance that we have just created:



8. Docker Setup In EC2 commands to be Executed:

Run all the below commands in EC2 instance via cmd CLI. Also, please make sure to run below command in the sequential order.

#optinal

sudo apt-get update -y

sudo apt-get upgrade

#required

curl -fsSL https://get.docker.com -o get-docker.sh "for installing docker on EC2"

sudo sh get-docker.sh

sudo usermod -aG docker ubuntu “changing to user mode so that we don’t need to write sudo cmd again and again”

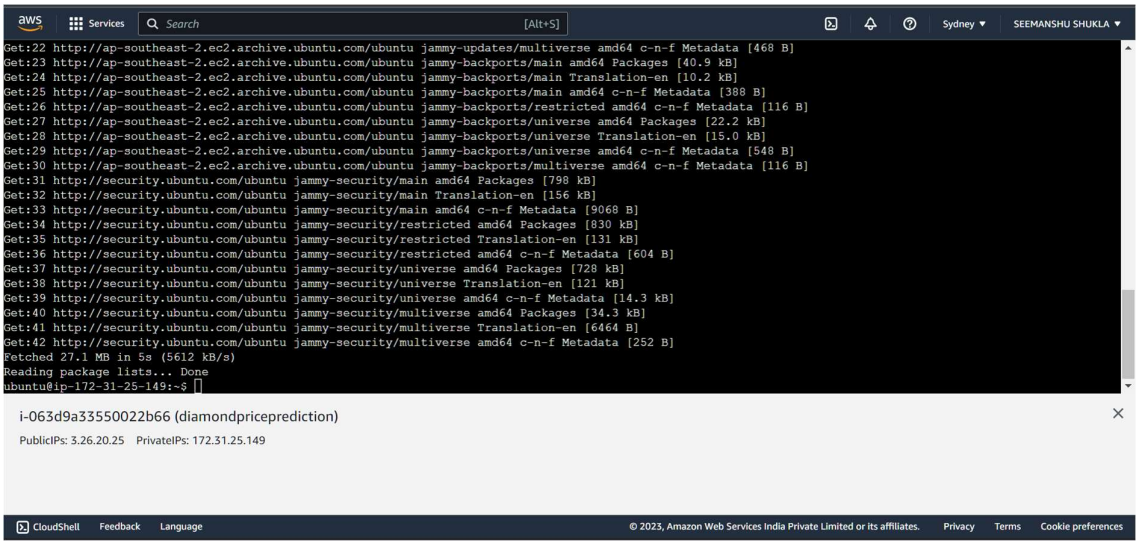
newgrp docker “creating a new group with name docker”

Docker Setup In EC2 commands to be Executed

```
#optinal

sudo apt-get update -y

sudo apt-get upgrade

#required

curl -fsSL https://get.docker.com -o get-docker.sh

sudo sh get-docker.sh

sudo usermod -aG docker ubuntu

newgrp docker
```

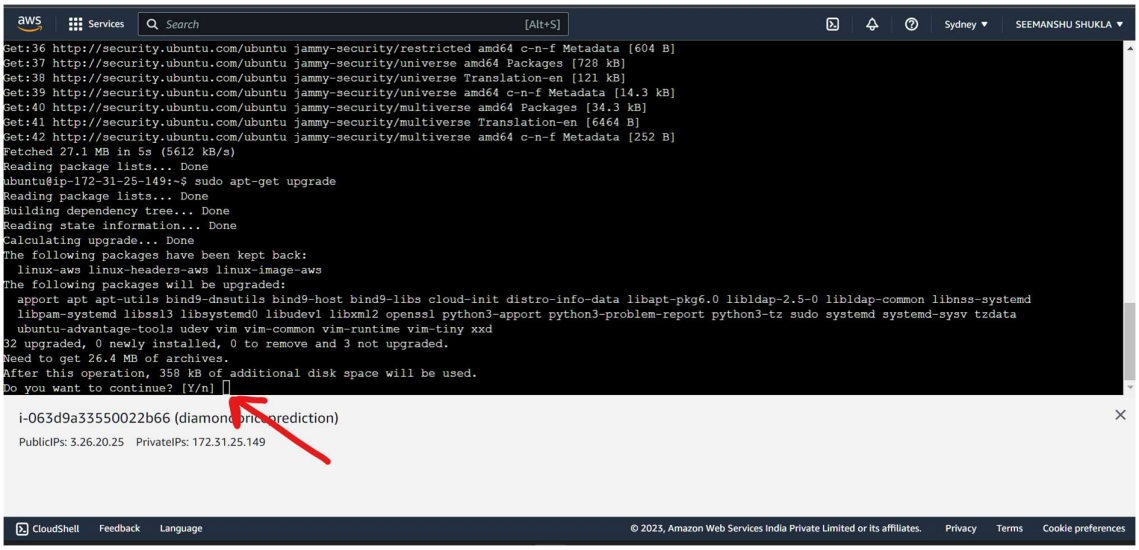
Now, we will execute above commands in sequential manner in our EC2 instance command prompt.

→First cmd:

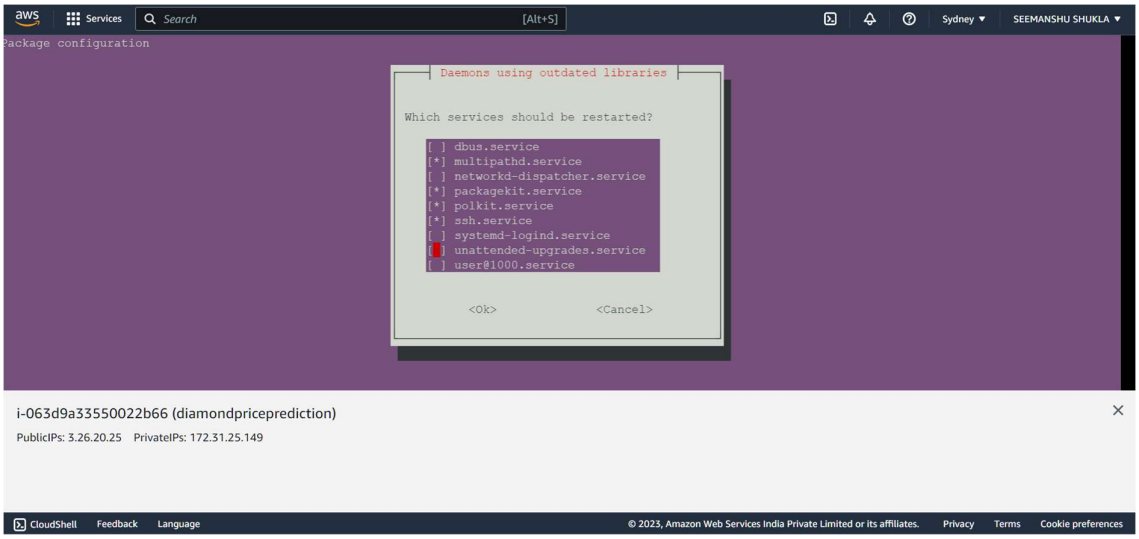


→Second cmd:

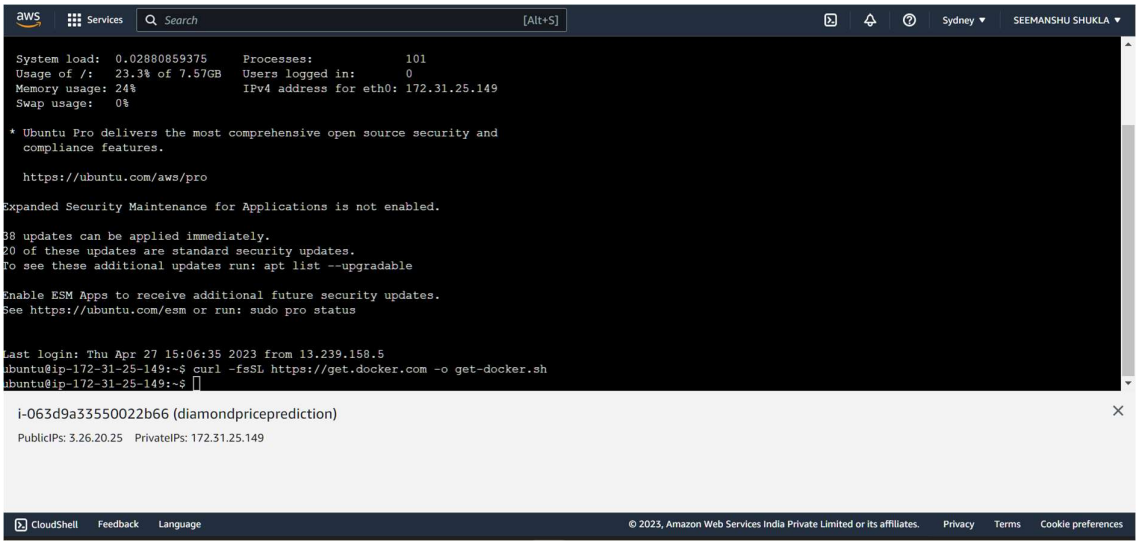
Give **“Y”** permission when asked Do you want to continue?



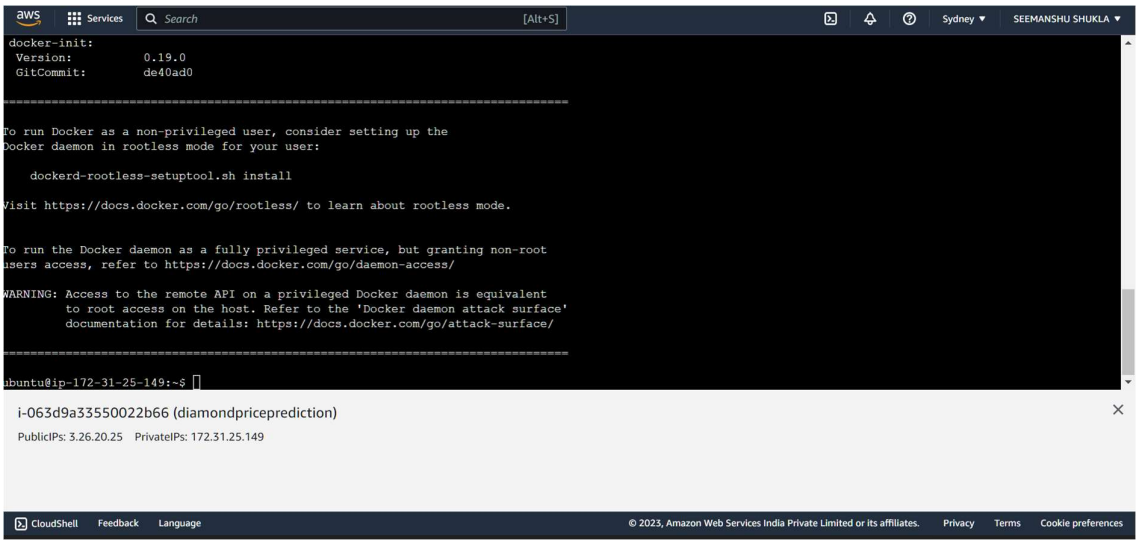
If prompted to this page then simply close the instance by closing the browser tab where instance is opened and open the instance again as we did earlier and run Third cmd.



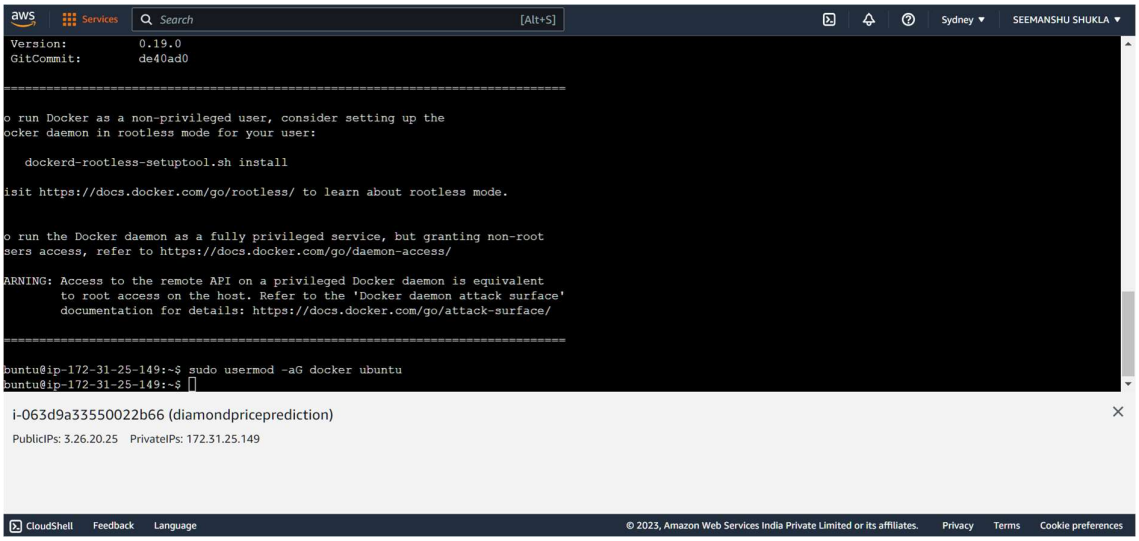
→Third cmd:



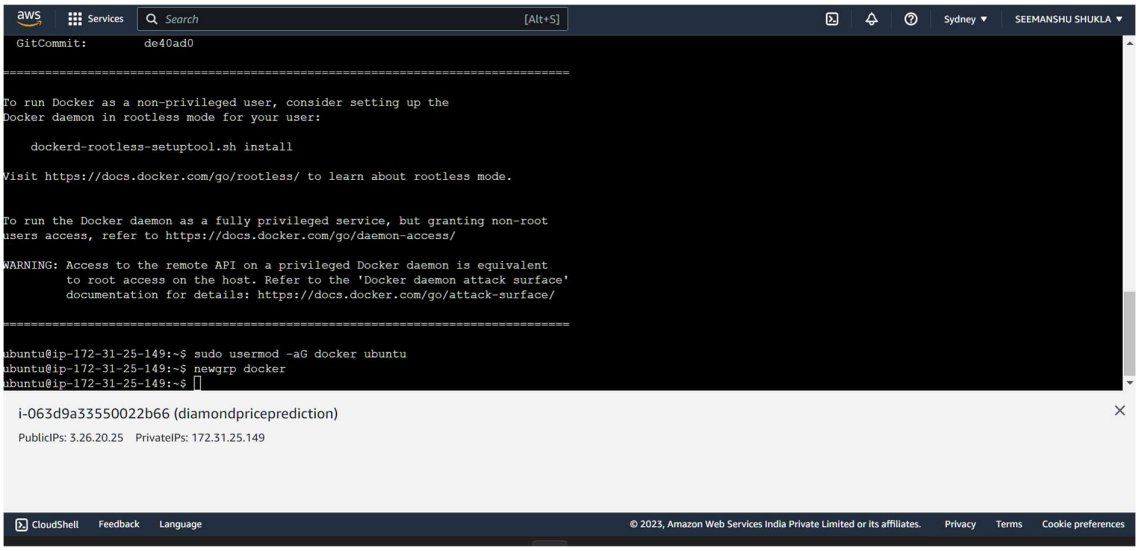
→Fourth cmd:



→Fifth cmd:



→Sixth cmd:

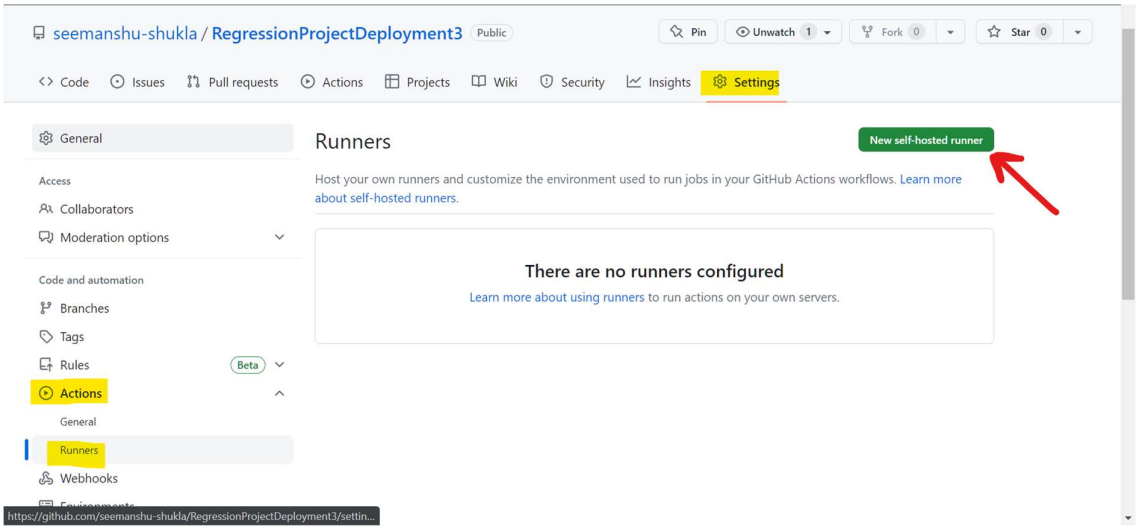


9. **Self-hosted runner setup:**

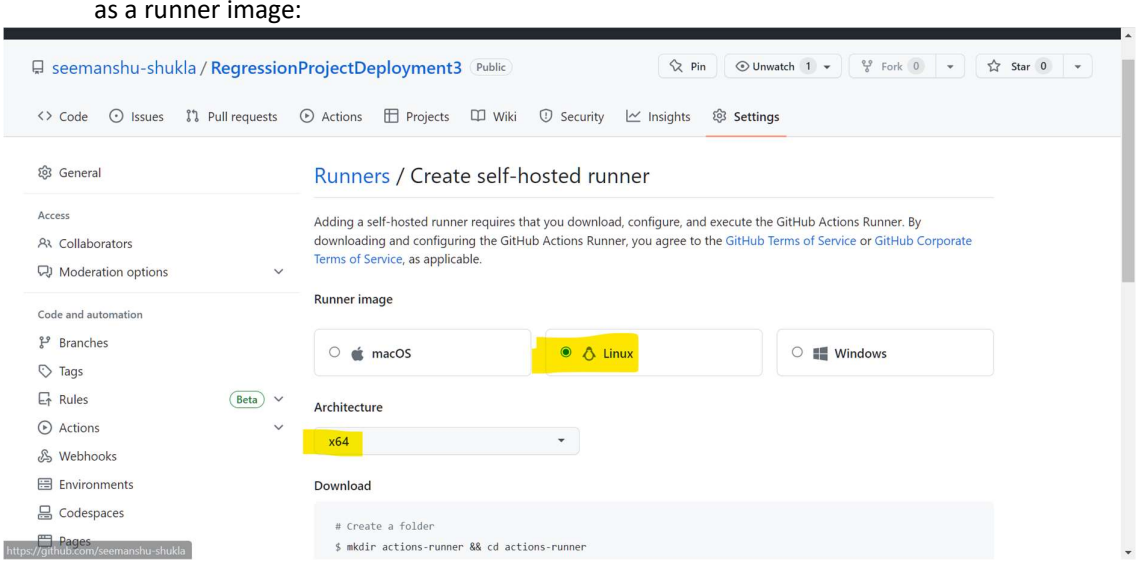
So when we push any new changes to our GitHub repository then our deployment should automatically listen it and push those changes to the existing deployment. To make this happen we use self-hosted runners.

I. Let's create a runner in GitHub: Traverse as stated below

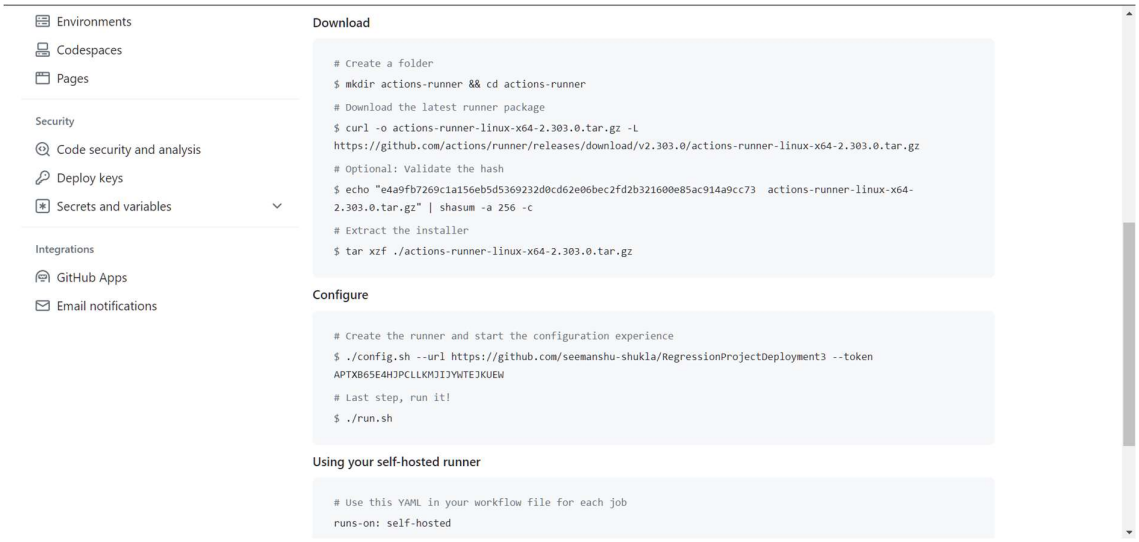
Open GitHub repo where application's codebase is pushed > Settings > Actions > Runners > and then click **New self-hosted runner**.



II. Since, our server(EC2 instance) is Linux (64 bit) based we will selecting Linux (64 bit) as a runner image:

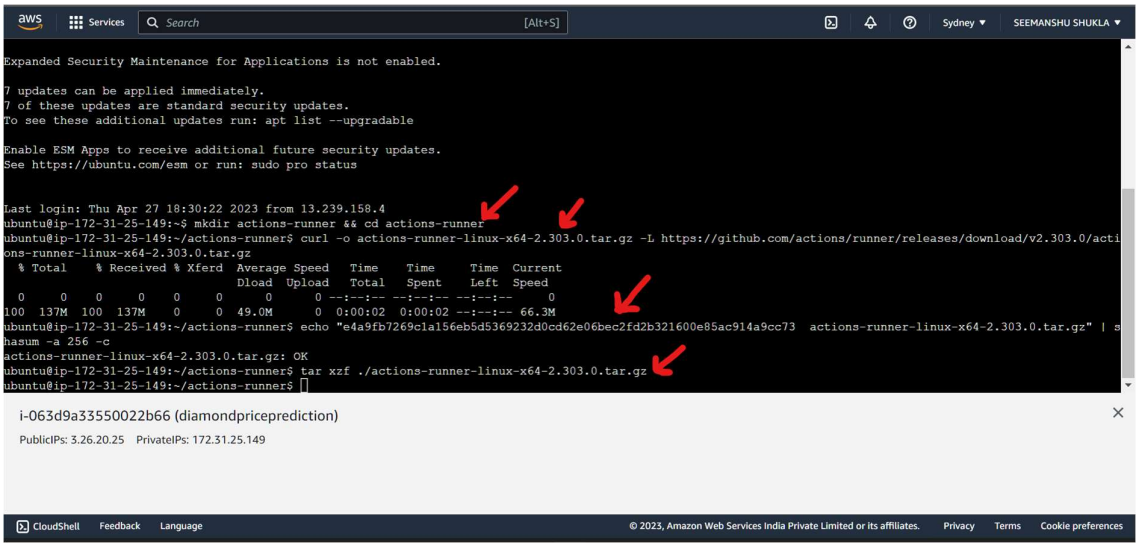


III. Scroll down and code template thus generated will be required to be executed in the server instance (EC2) post which our self-hosted runner will start listening to all the changes that comes into GitHub and will push it to our server(EC2 instance where app will be deployed).



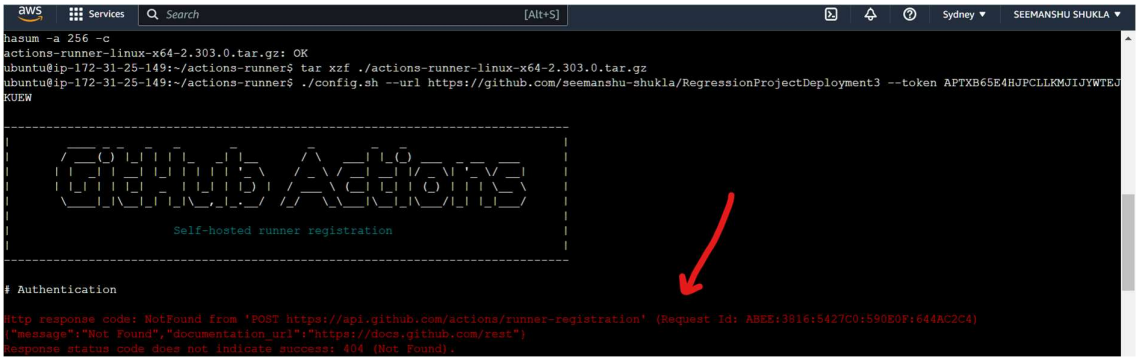
Run above commands in sequential order inside server instance via EC2 cmd prompt.

IV. For **Download**:



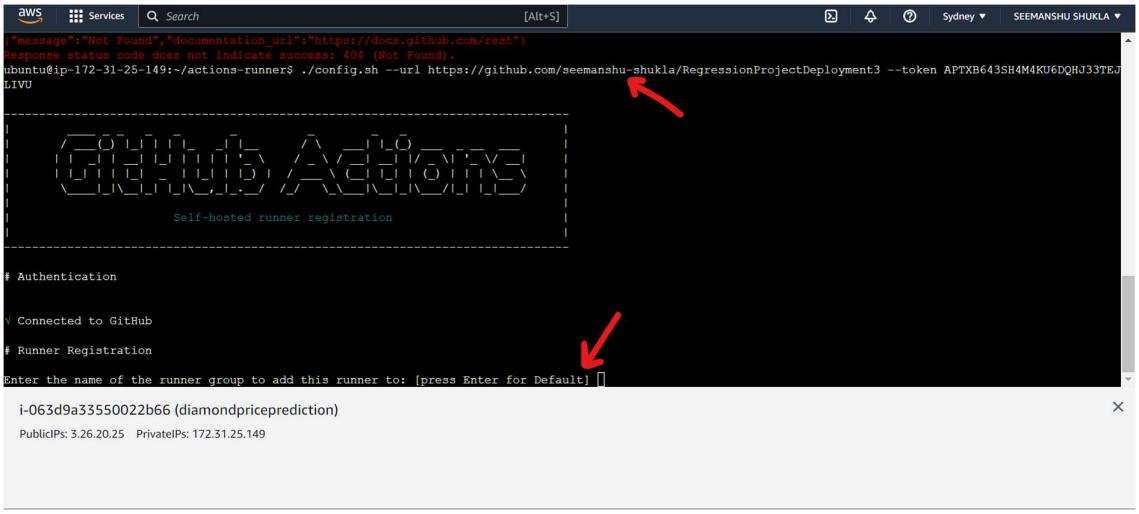
V. For **Configure**:

→After running first configure cmd was getting this error:



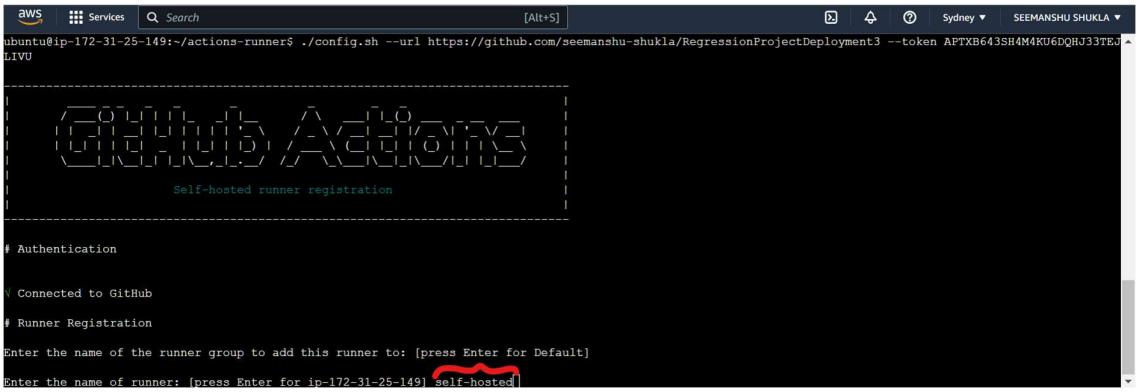
→As a resolution have just reloaded the GitHub runner page from where we generated runner code template for Linux based machine.

→After successfully running the **first cmd** for **Configure**:

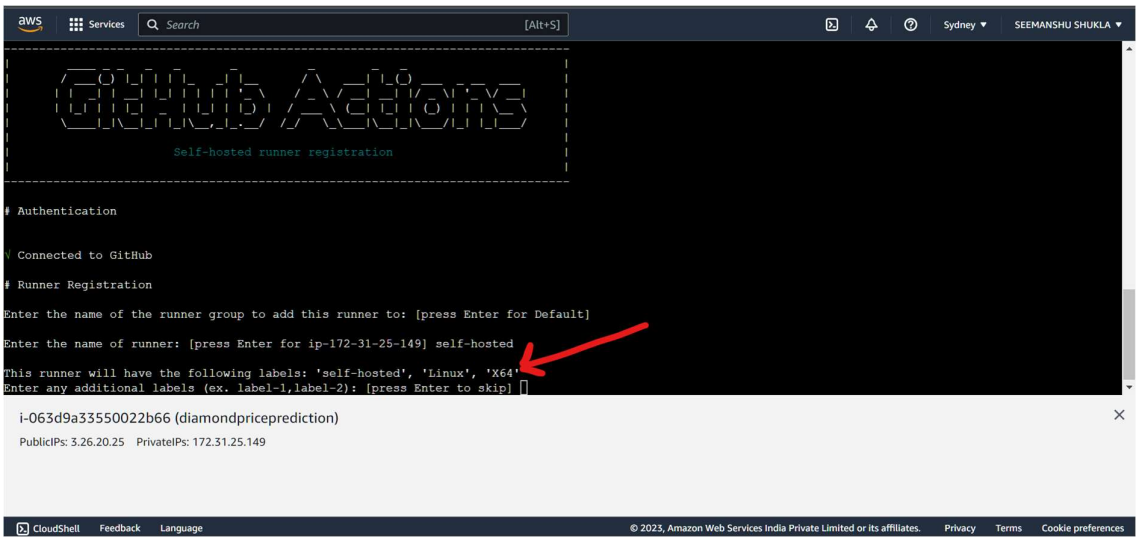


In above hit enter to pass default runner group name.

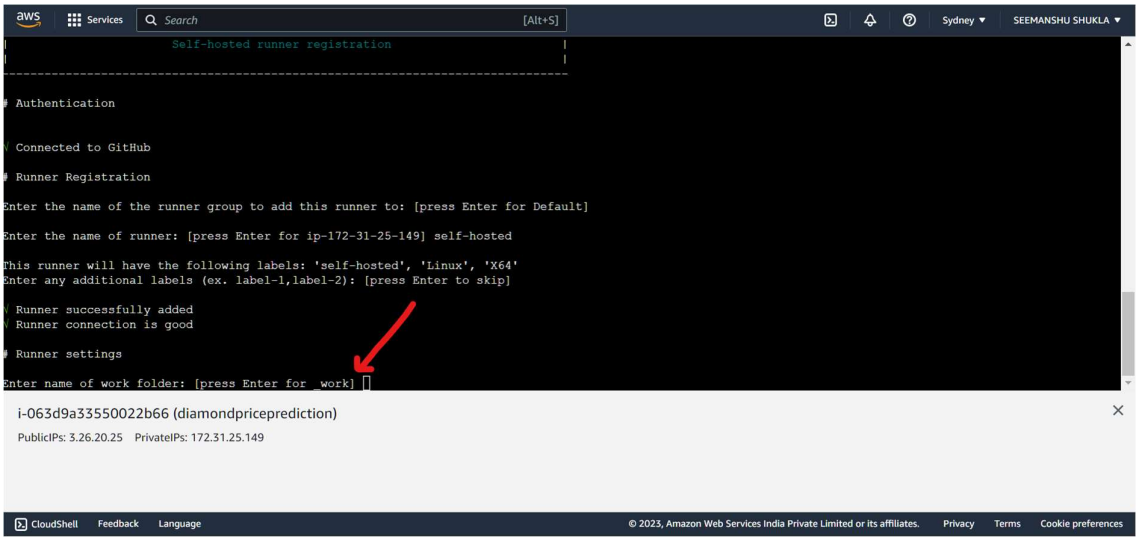
→**"self-hosted"** is the runner name that we usually given to a runner and then hit **Enter**:



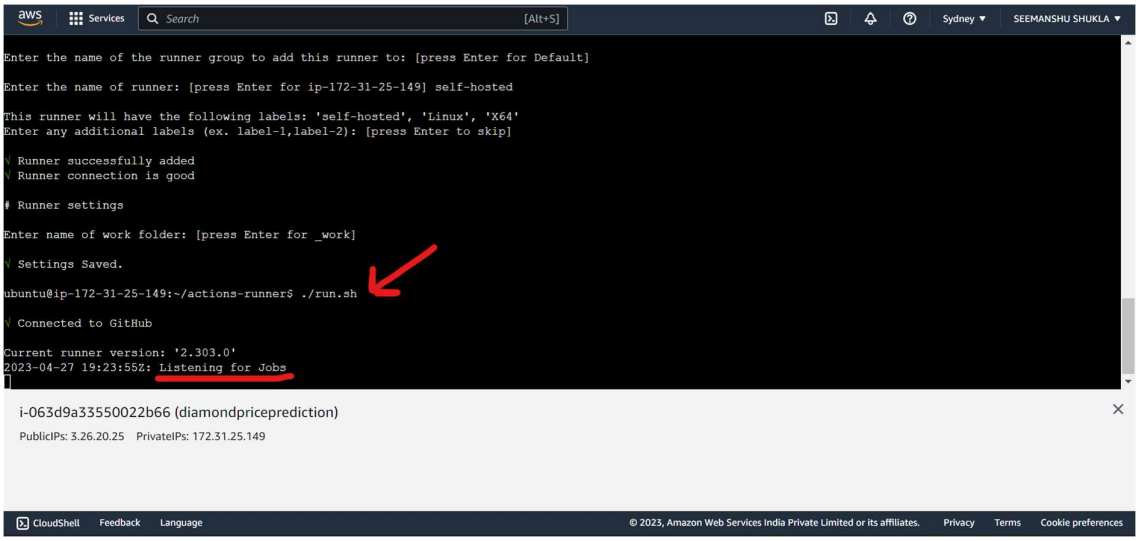
→Configuration/labels of runner getting displayed. After this hit **Enter** to skip the step of adding any additional labels:



→ To pass the default work folder name **"_work"** press **Enter**:



→ Finally, use cmd **"./run.sh"** to run the runner. Observe we get a message ***"Listening to Jobs"*** which means that the runner is now running and actively listening for new jobs(new push to GitHub repo). In case it listens the new job then it will push it to the server. This is how runner is used to achieve CD(Continuous Deployment).

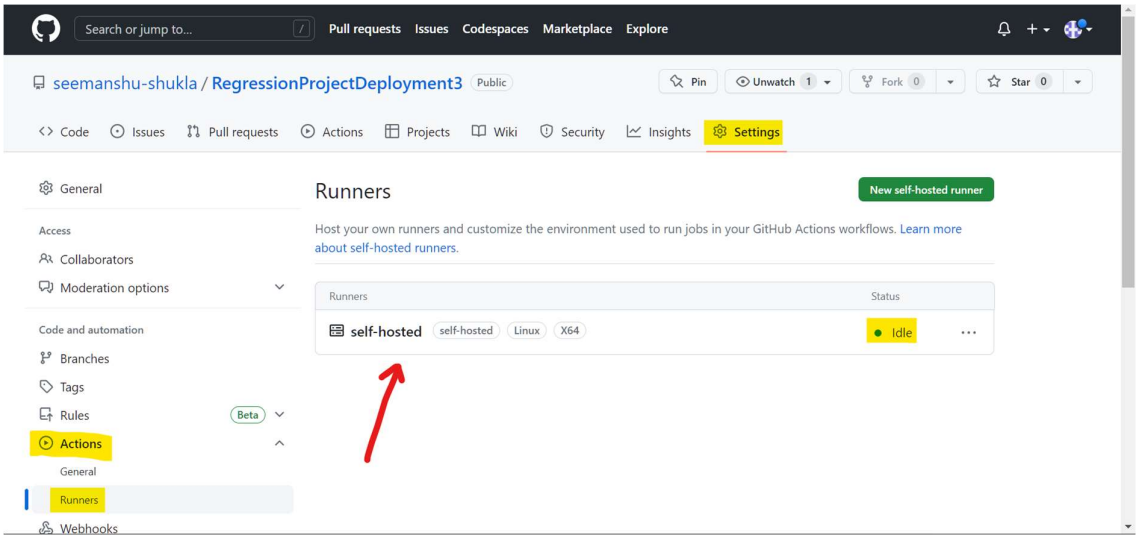


VI. Code under **Using your self-hosted runner**:

Runs on: self-hosted

This will be used in yaml file placed under .github/workflows folder.

VII. Now, in GitHub if we again go to Settings>Actions>Runners then we can see our runner that we just created. Currently, it is in **Ideal state** because no new changes are pushed to the GitHub repository. Moment when some changes are made runner will become **Active** post which it will push the changes to server(EC2 instance). This is how we are achieving continuous deployment (CD) in an attempt to create CI-CD pipeline.

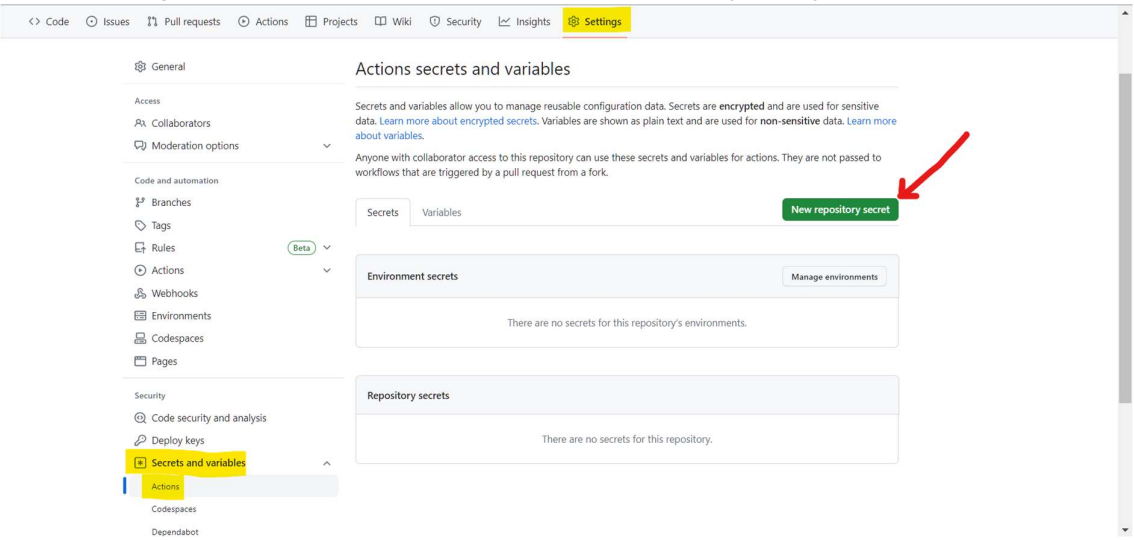


10. **<u>Setup GitHub secrets</u>:**
    Example-
    AWS_ACCESS_KEY_ID=
    AWS_SECRET_ACCESS_KEY=
    AWS_REGION = ap-southeast-2
    AWS_ECR_LOGIN_URL= demo>> 566373416292.dkr.ecr.ap-south-1.amazonaws.com
    ECR_REPOSITORY_NAME = simple-app

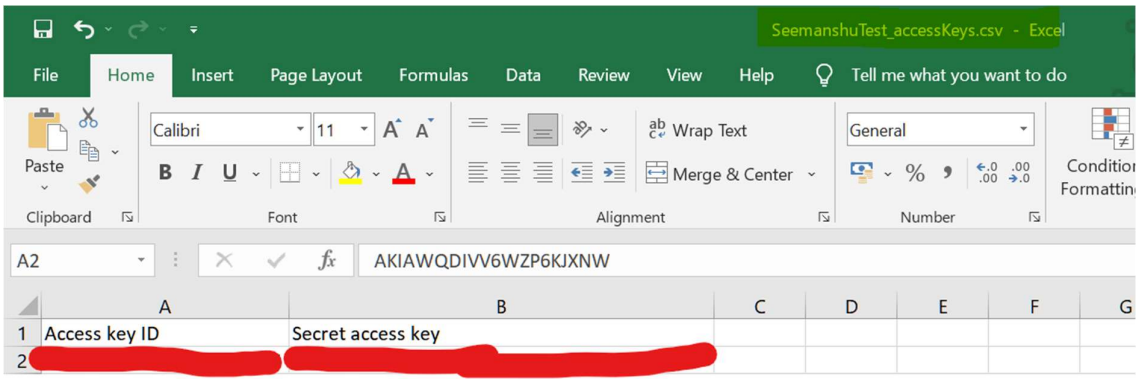
> **Setup github secrets:**
>
> AWS_ACCESS_KEY_ID=
>
> AWS_SECRET_ACCESS_KEY=
>
> AWS_REGION = us-east-1
>
> AWS_ECR_LOGIN_URI = demo>> 566373416292.dkr.ecr.ap-south-1.amazonaws.com
>
> ECR_REPOSITORY_NAME = simple-app

GitHub is the **deployment provider**. In order to make this deployment work we will need to configure our GitHub with some secrets. Also, please note that the deployment will be happening by using **jobs** mentioned under **.yaml workflow** file which will be using these GitHub secrets as well.

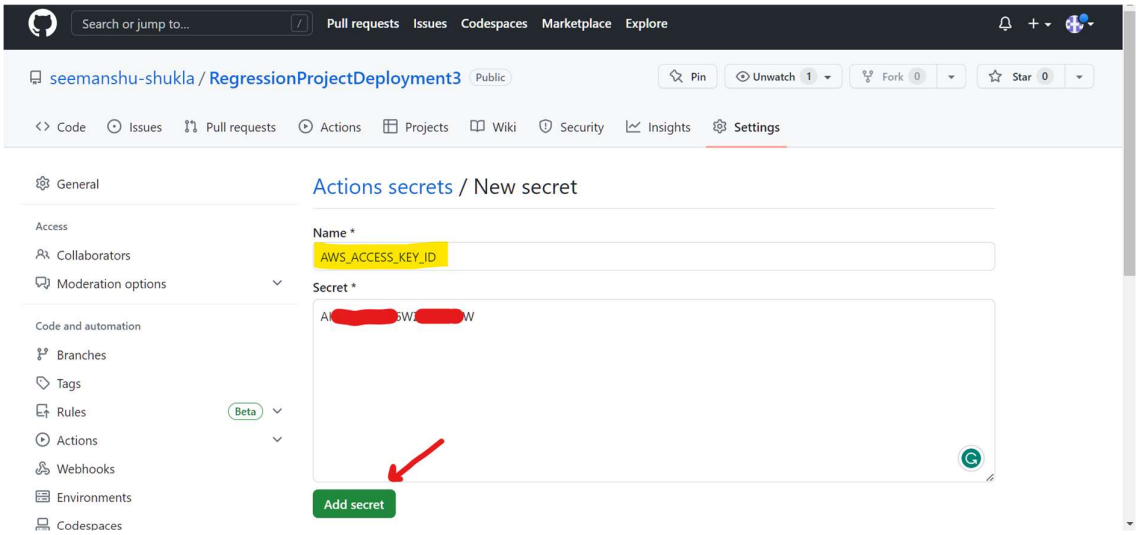
For creating these secrets traverse through the below path:
GitHub>Settings>Secrets and variables>Actions then click **New repository secret**



A) Setting up **AWS_ACCESS_KEY_ID** secret:
→Value for this secrete variable can be found in the **.csv file** that we downloaded after creating **IAM user *"SeemanshuTest"*** in **Step 5**:

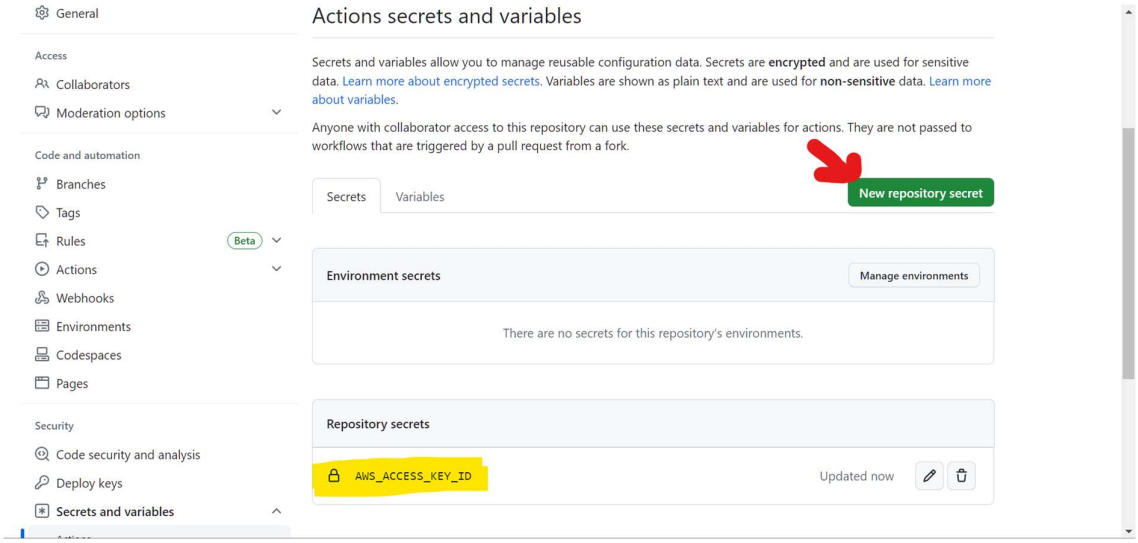


→Enter secret name as highlighted below and secret value copied from ***"Access key ID"*** column from **.csv file** and click **Add secret**:

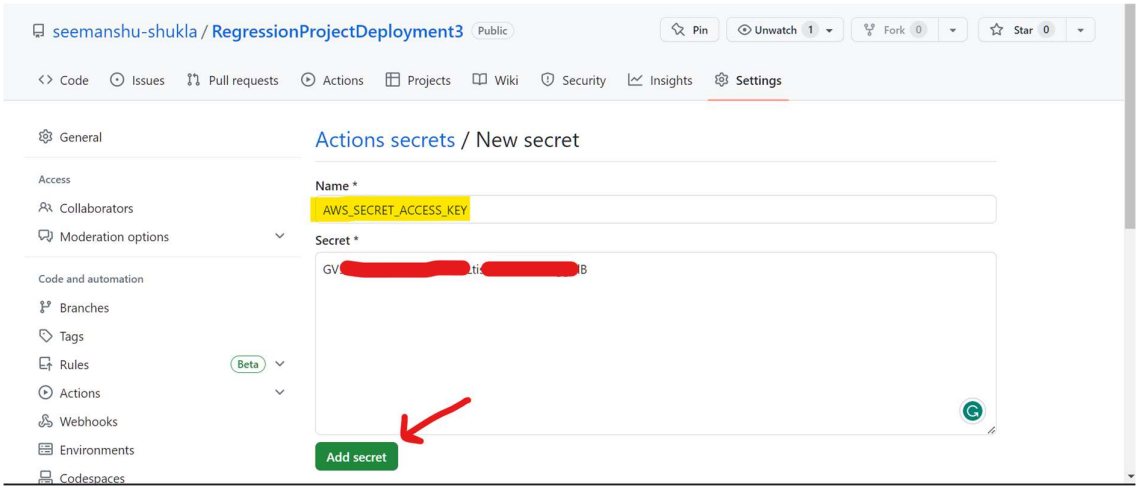


B) Setting up **AWS_SECRET_ACCESS_KEY** secret:

➔ Below we can see that the secret that we set in above step is now reflecting under **Repository secrets** section. Click on **New repository secret:**

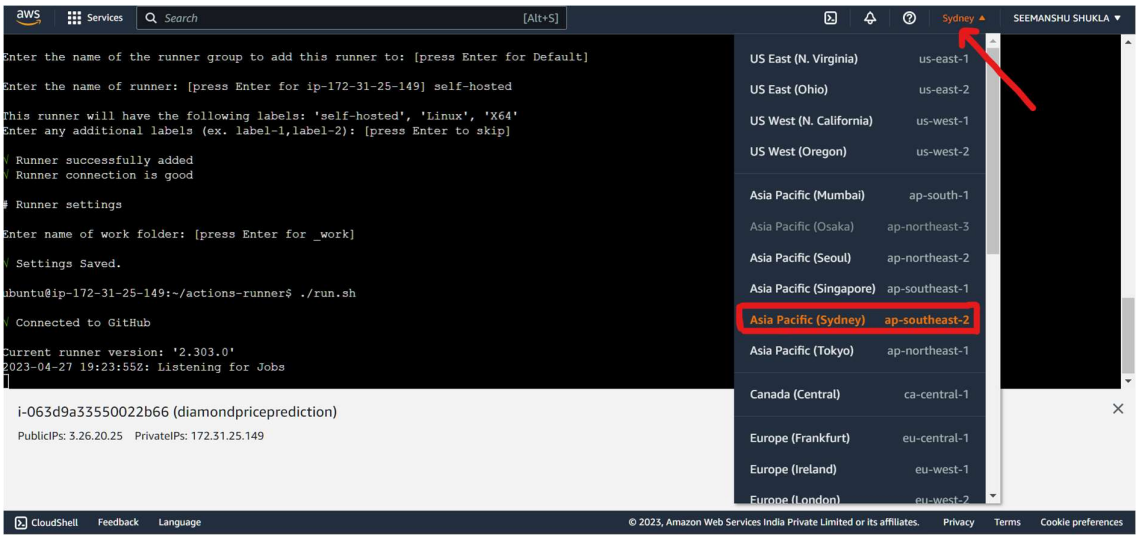


➔ Enter secret name as highlighted below and secret value copied from *"**Secret access key**"* column from **.csv file** and click **Add secret**:

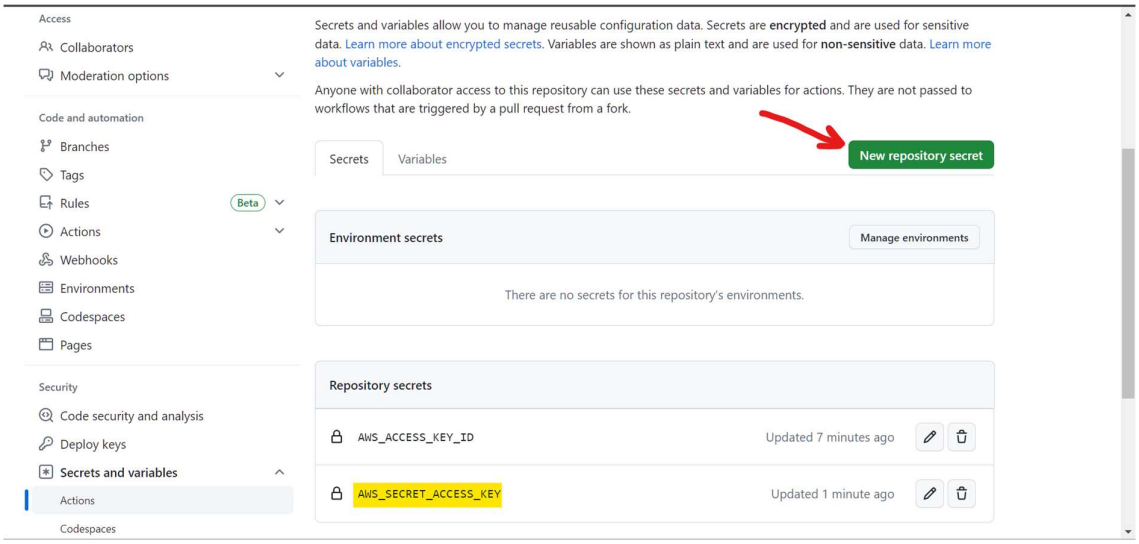


C) Setting up secret **AWS_REGION** with value **ap-southeast-2**:

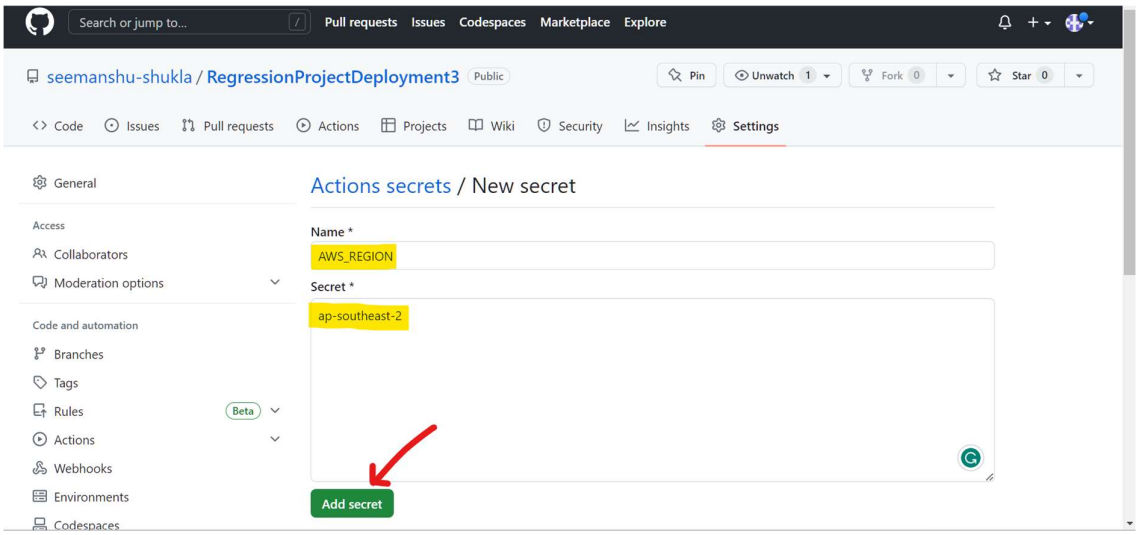
➔ To find AWS_REGION go to AWS console and on top we get an option to check AWS_REGION. In our case our instance is available in **Sydney** region which is represented as **ap-southeast-2:**



➔ Below we can see that the secret that we set in above step is now reflecting under **Repository secrets** section. Click on **New repository secret:**



➔ Enter the highlighted secret name and secret value based on AWS region(as explained above) and click **Add secret:**



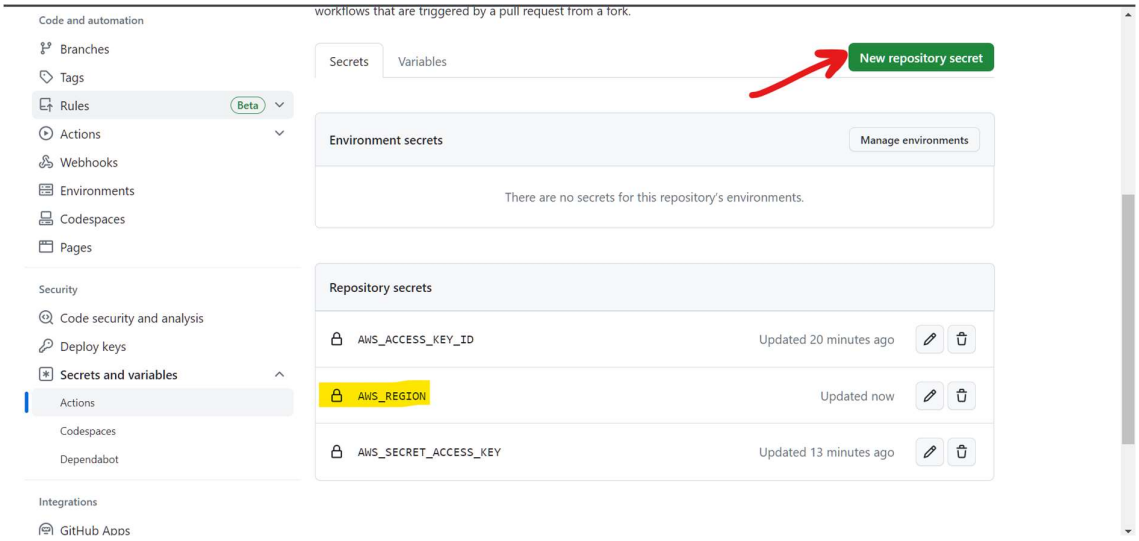
D) Setting up **AWS_ECR_LOGIN_URI** secret:

Recall In **Step 2** while creating **ECR** repository we copied a **URL**. This URL will be used as secret value here. In our case copied URL is given below:

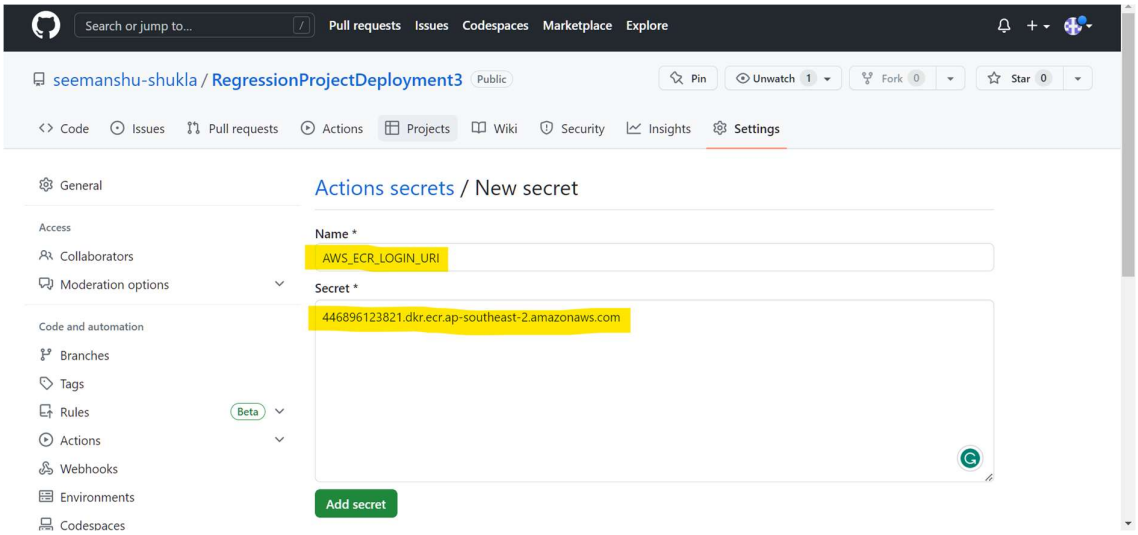
446896123821.dkr.ecr.ap-southeast-2.amazonaws.com/diamondpriceprediction
In above link ***"diamondpriceprediction"*** represents ECR repository name which will be used while setting **ECR_REPOSITORY_NAME** secrete variable.

Please Note:
For AWS_ECR_LOGIN_URI we will remove ECR repository name and then pass the resultant link.
That is we will pass: ***446896123821.dkr.ecr.ap-southeast-2.amazonaws.com***

➔ Below we can see that the secret that we set in above step is now reflecting under **Repository secrets** section. Click on **New repository secret:**

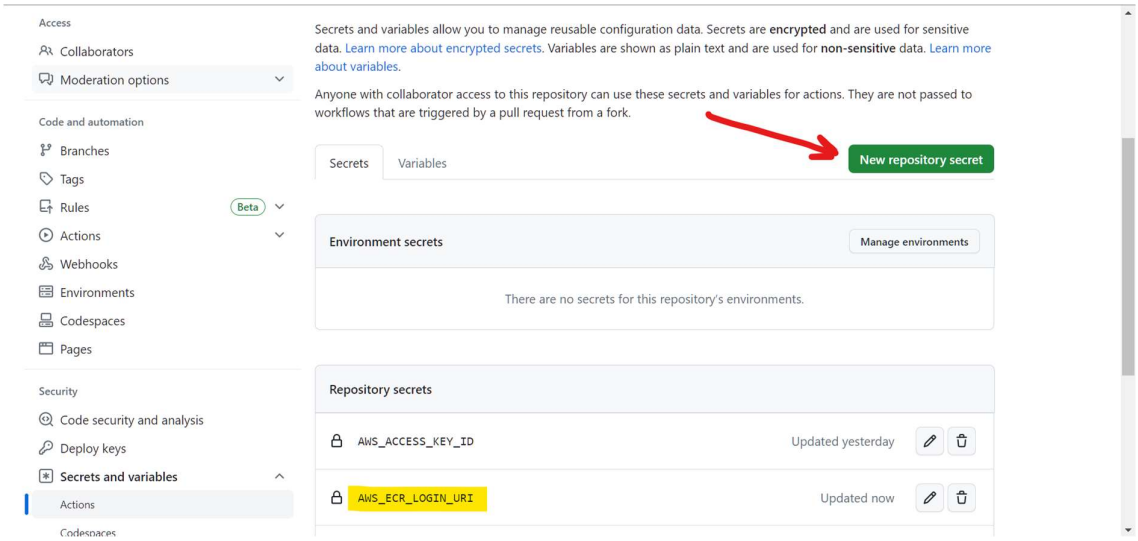


➔ Pass name and secrete as highlighted below and then click **Add secret:**

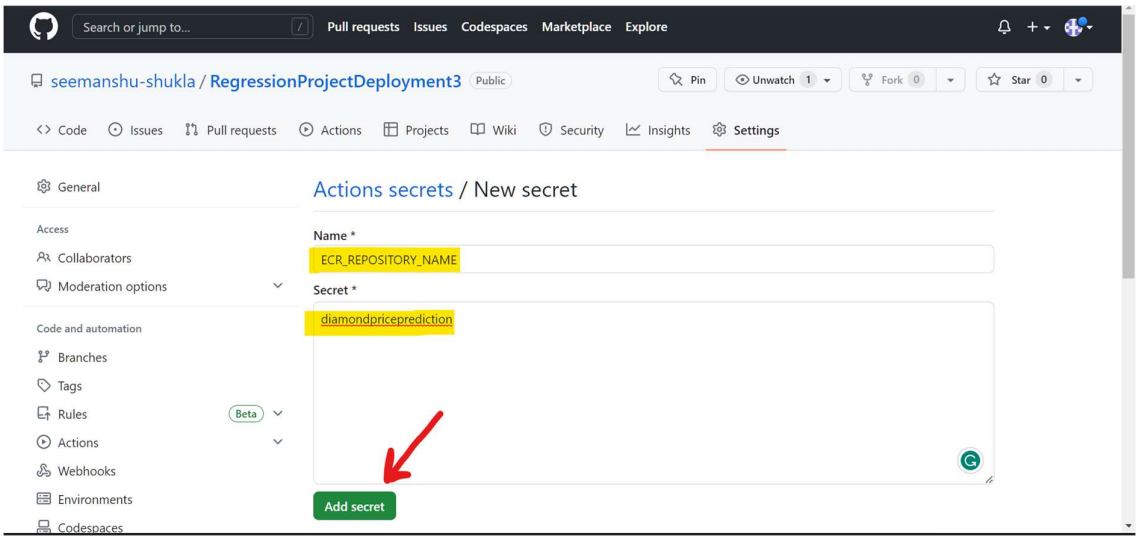


E) Setting up the **ECR_REPOSITORY_NAME** secret:
In the URL used above that last substring followed by "/" is ECR_REPOSITORY_NAME, that is ***"diamondpriceprediction"***

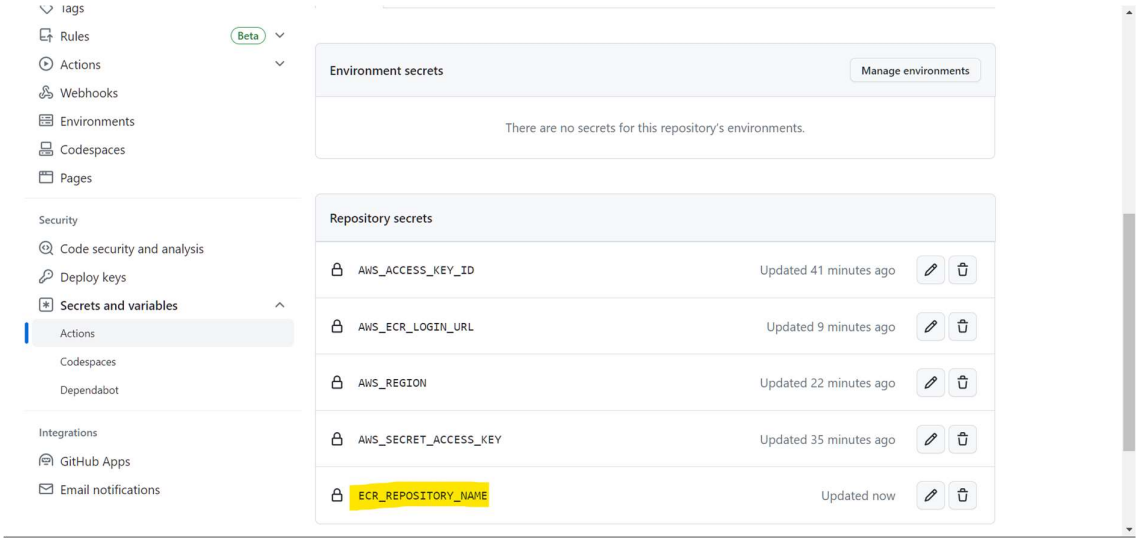
➔ Below we can see that the secret that we set in above step is now reflecting under **Repository secrets** section. Click on **New repository secret:**



➔ Enter the below highlighted variable name and secrete. In my case ECR name is **diamondpricepredication**. After this click **Add secret:**

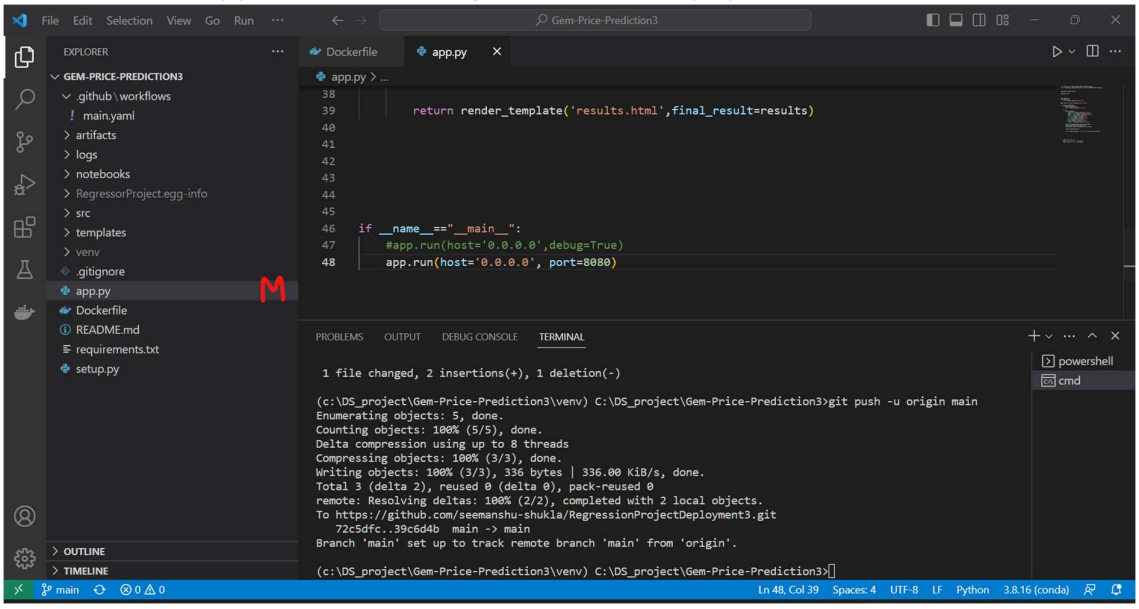


➔ In the below snip we can see that our secret variable **ECR_REPOSITORY_NAME** is now reflecting under **Repository secrets** section meaning that it has been created successfully:



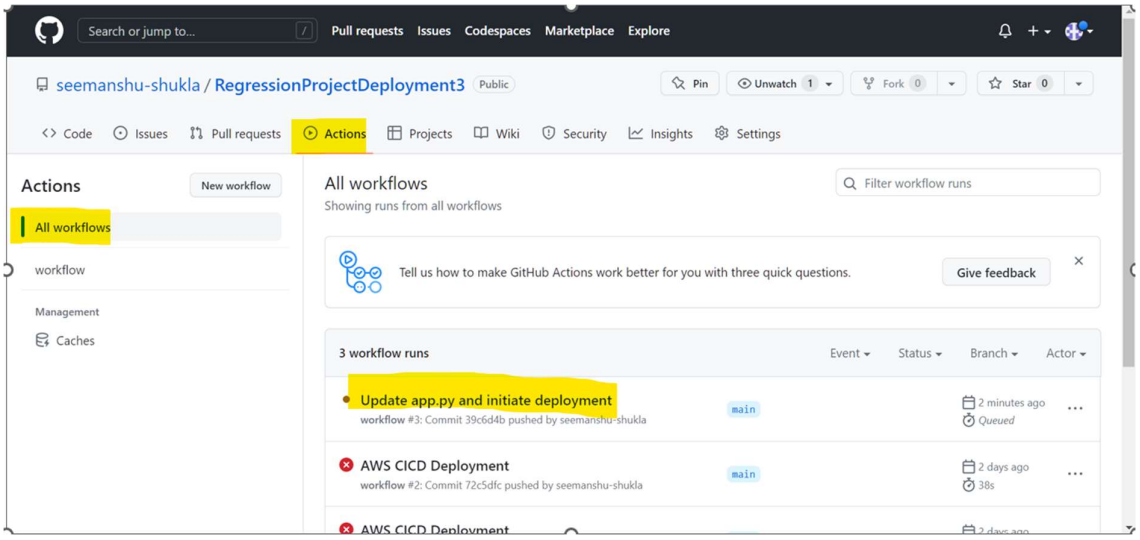
11. **<u>Starting the deployment and visualizing it inside GitHub</u>:**

Make some changes like add an extra space in app.py and push it to GitHub after which **self-hosted Runner** will start listening for the **jobs** as defined under .**yaml workflow file** after which deployment will get initiated in CICD fashion as defined under .yaml file. As we initiate the deployment **"actions/checkout@v3"** (as defined under .yaml) will scan the entire GitHub repository and will look for the latest changes. As this is the first time runner is getting activated hence, it will push entire app in the server. Later based on changes made in codebase it will only push the latest changes to the server deployment:

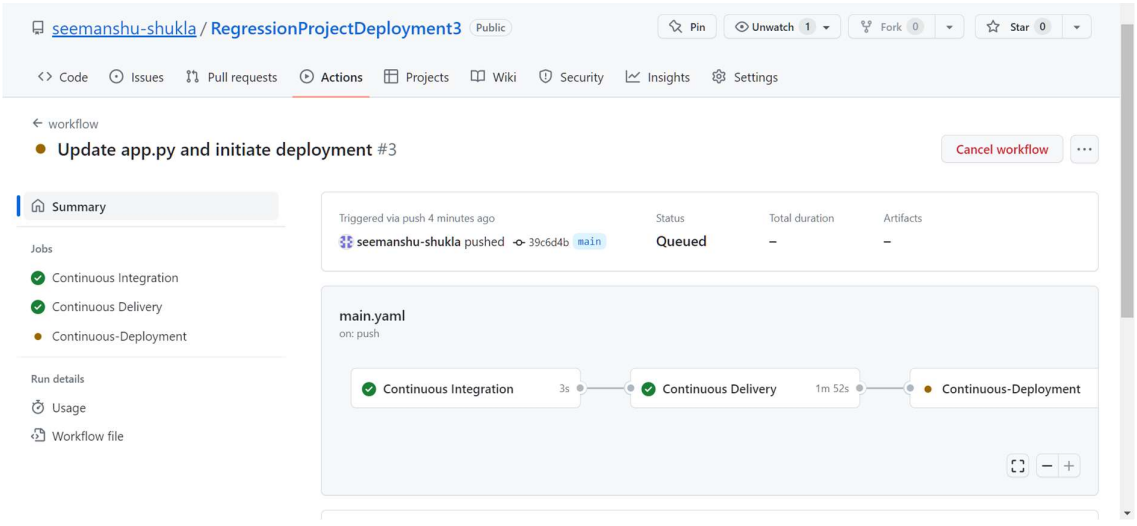


In above snip observe **M** adjacent to app.py file in Vscode which represents modified as we have added extra space just to initiate the deployment.

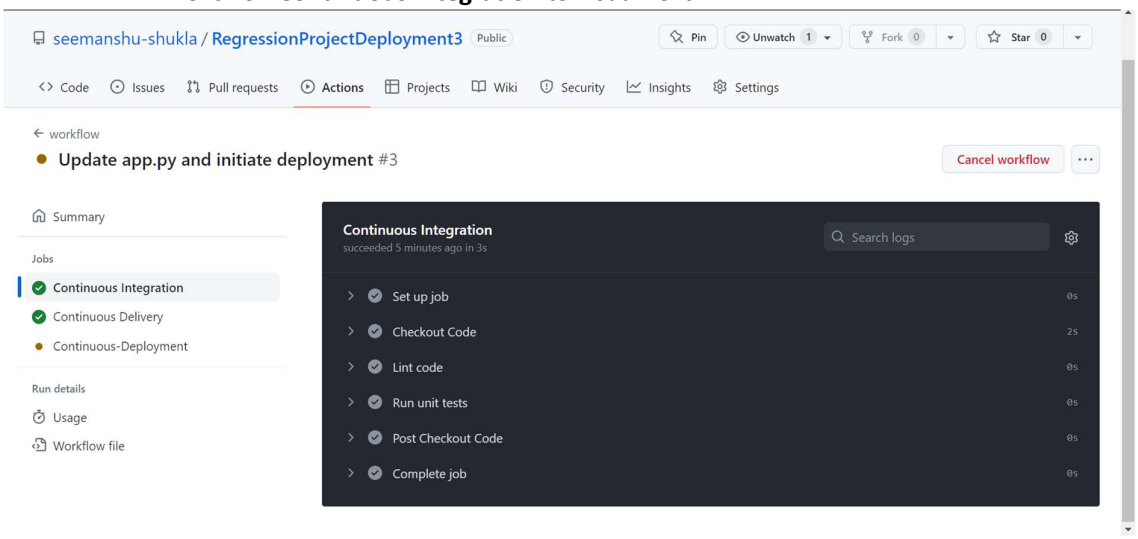
➔ As our changes are successfully committed to GitHub then observe that under **Actions** a new workflow is created:



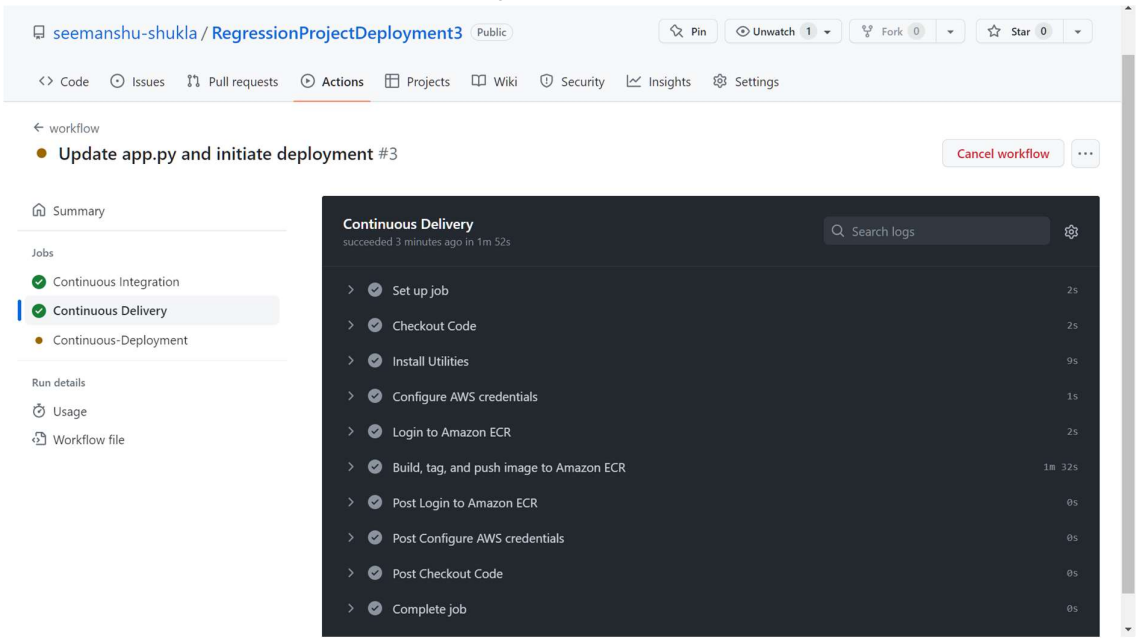
➔ If we open this workflow we can visualize our entire deployment workflow happening in CICD fashion as defined under the **workflow file** that is **.yaml file**:



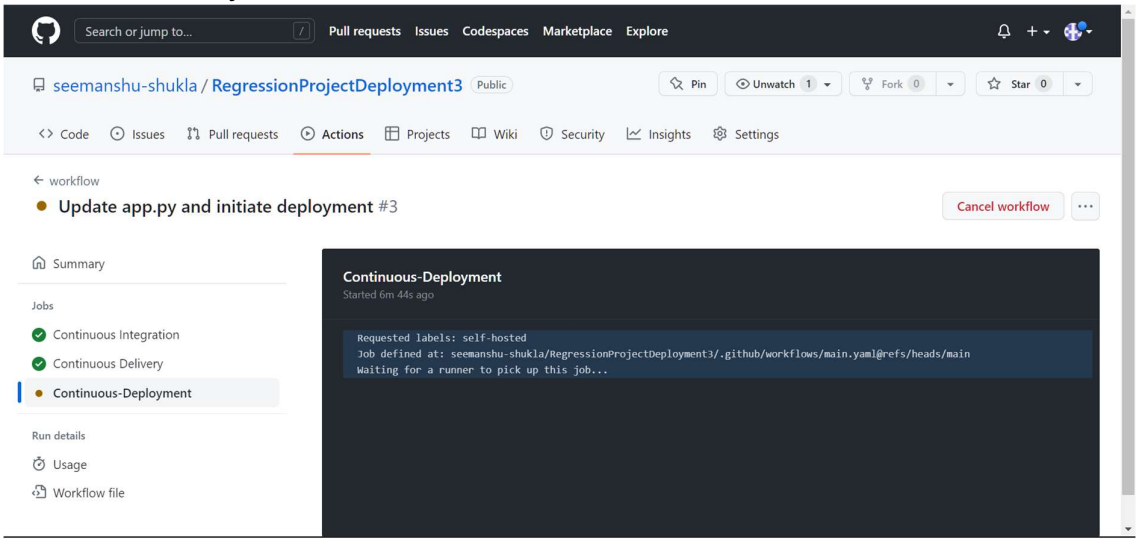
➔ Click on **Continuous Integration** to visualize it:



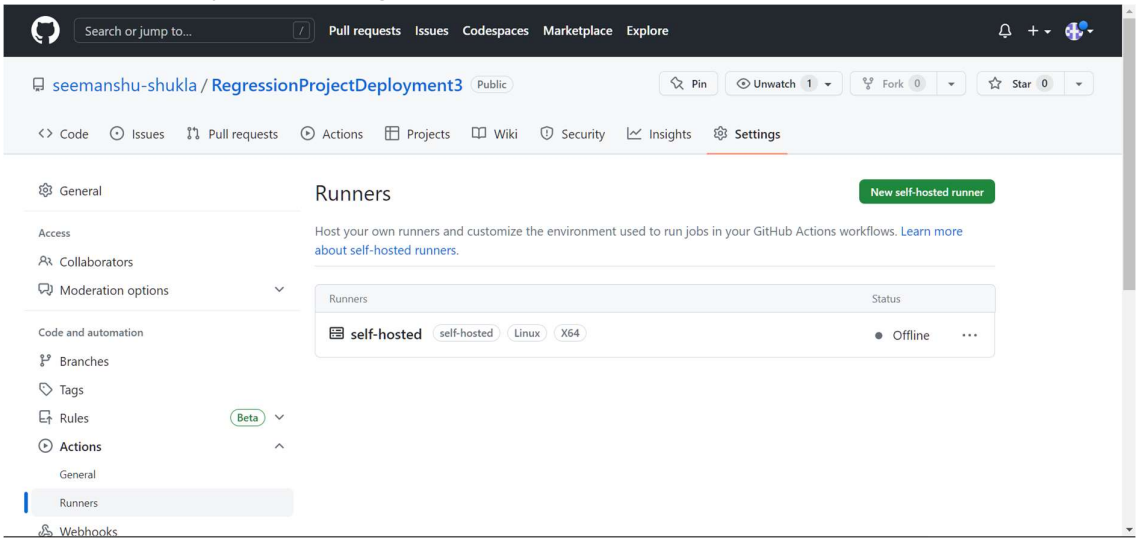
➔ Click on **Continuous Delivery** to visualize it:



➔ Click on **Continuous-Deployment** to visualize it. Below we can clearly see clearly see we are getting a message that "***Waiting for a runner to pick up this job…"***

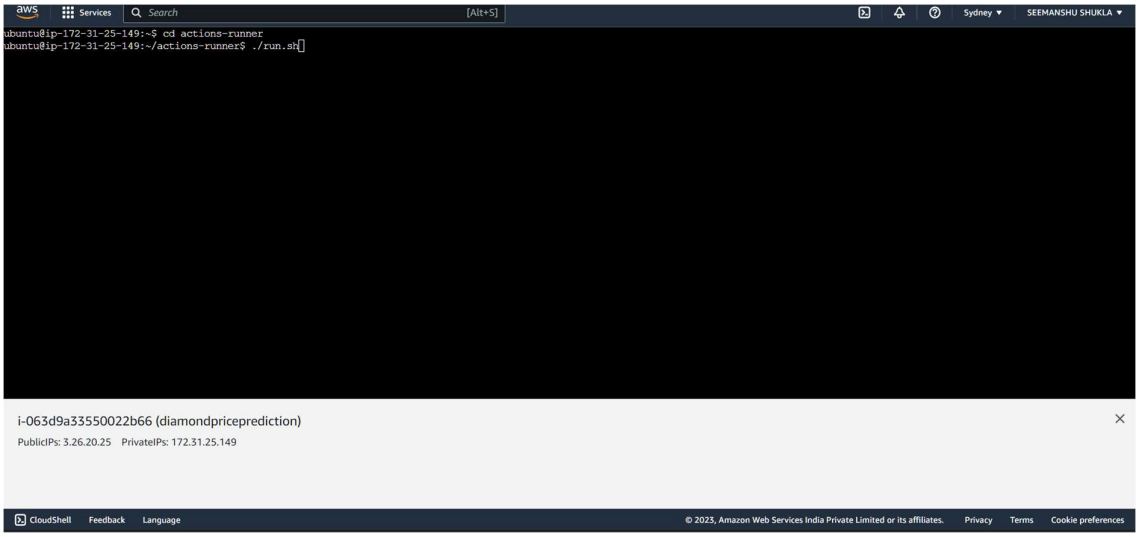


➔ On further investigating found that our **Runner** was in the **Offline** state. This is the reason why we were getting the message that "***Waiting for a runner to pick up this job…" .*** In my case after creating and running the runner I started the deployment on next day. So, may be due to inactivity my runner's object state changed to Offline.

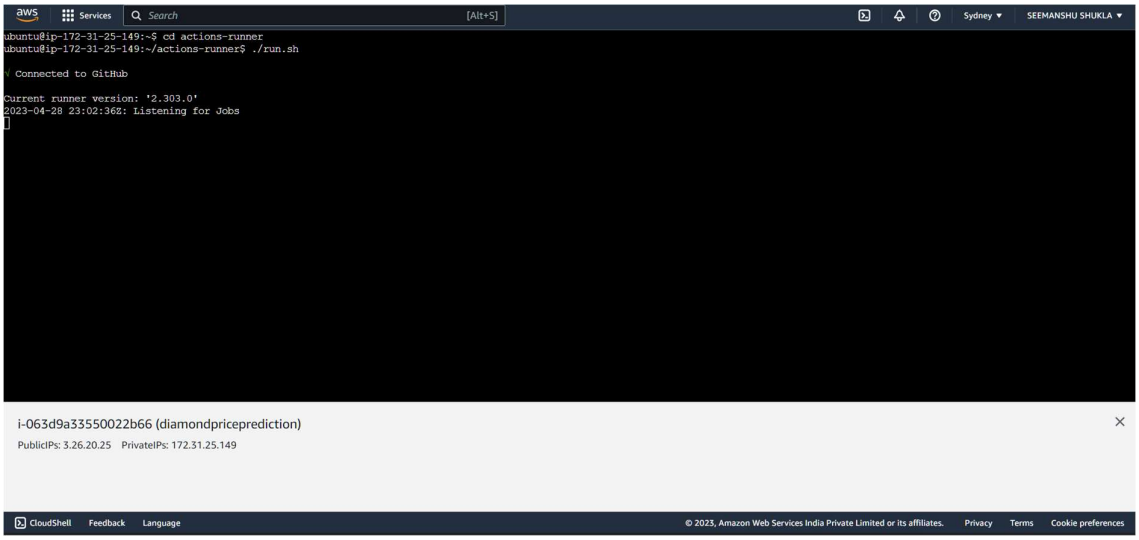


Please note: Before initiating deployment make sure that runner is not in Offline state otherwise deployment will stop/halt at Continuous-Deployment stage.

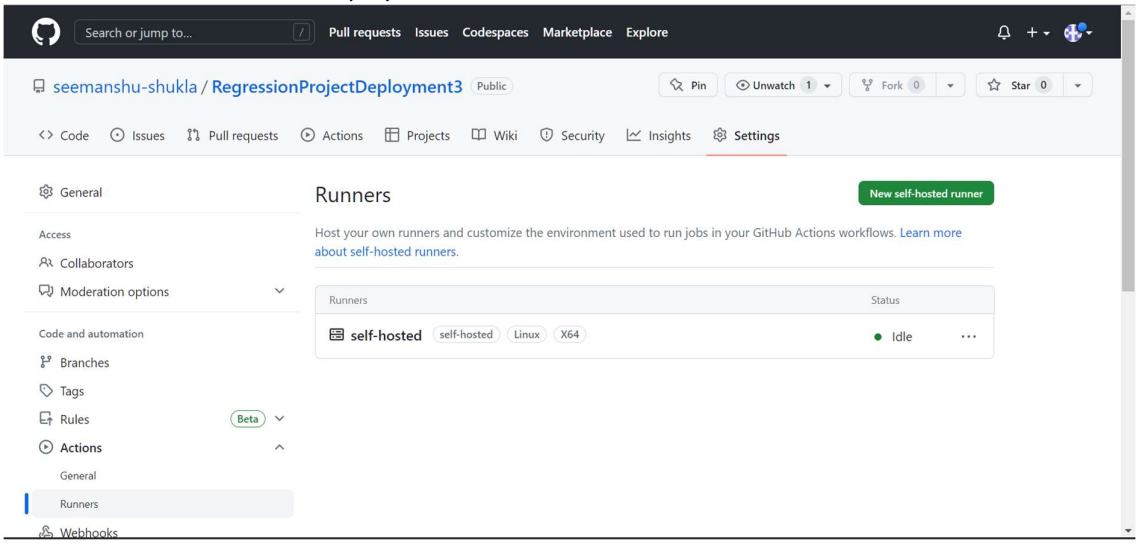
➔ As a resolution we again runed the runner inside EC2 instance as we did earlier while setting up runner in EC2 in **Step 9.** For this first we changed the directory to **actions-runner** using **“cd actions-runner”** command and then used **“./run.sh”** command to run the runner.



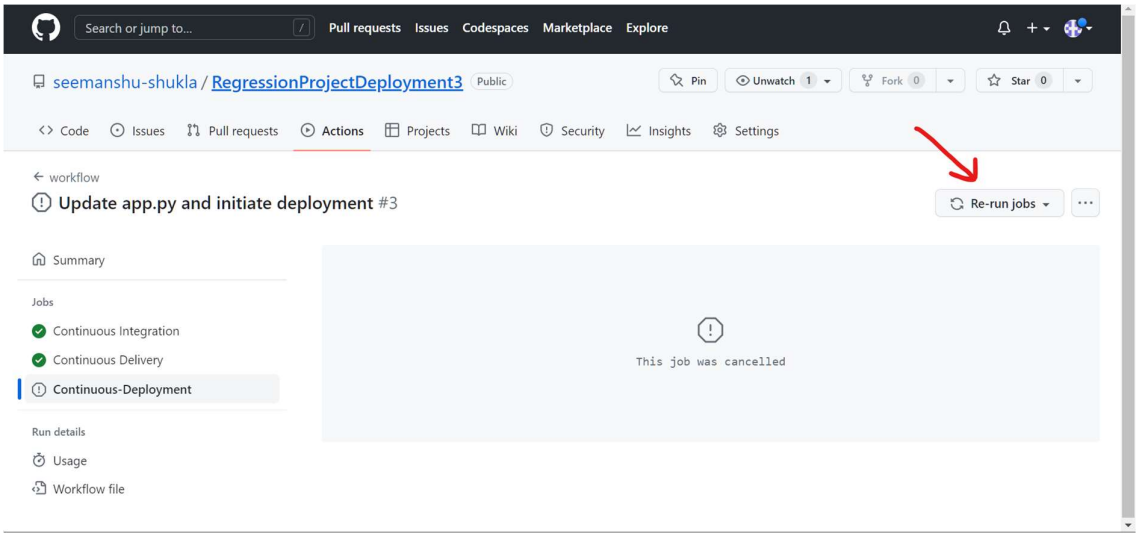
➔ We can see below that now we are getting a message that ***“Listening for Jobs”***



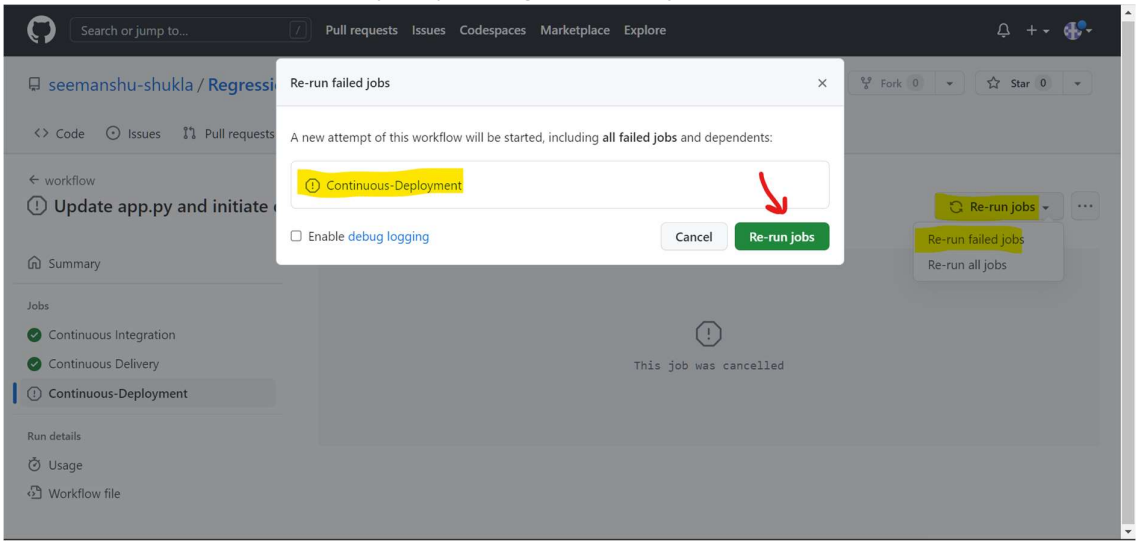
➔ To verify, lets now check the state of our Runner object. Now we can see that it is changed from **Offline** to **Ideal**. Therefore, now we are good to resume our deployment.



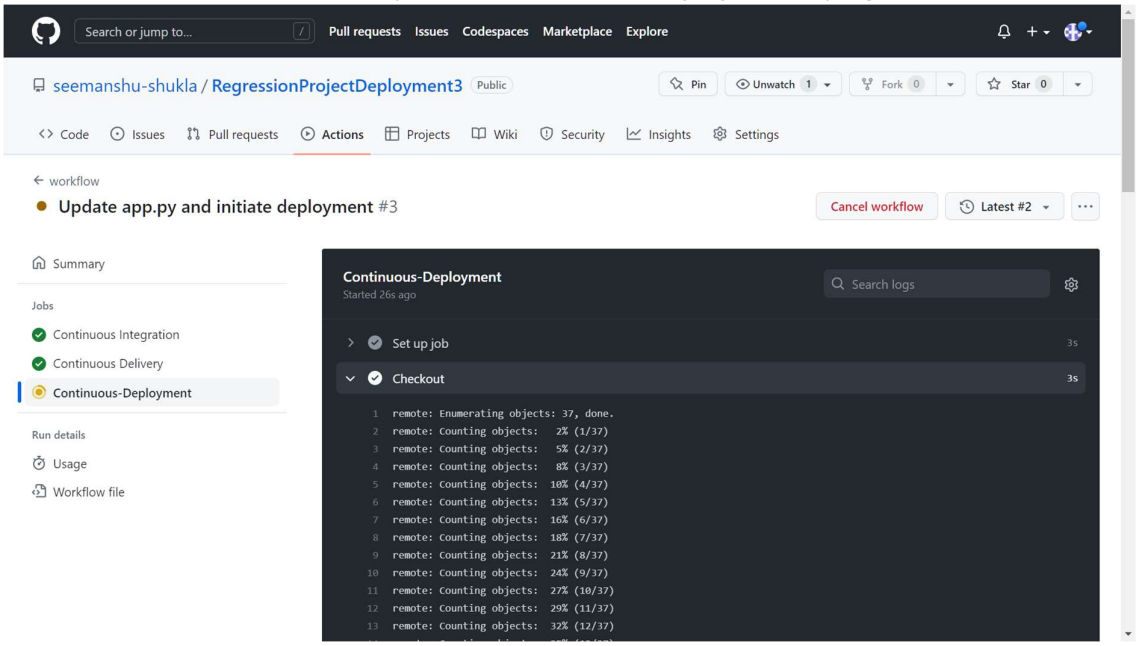
➔ Now, we will go back to our workflow and re-run the failed jobs to complete the deployment. Click **Re-run jobs** drop down:



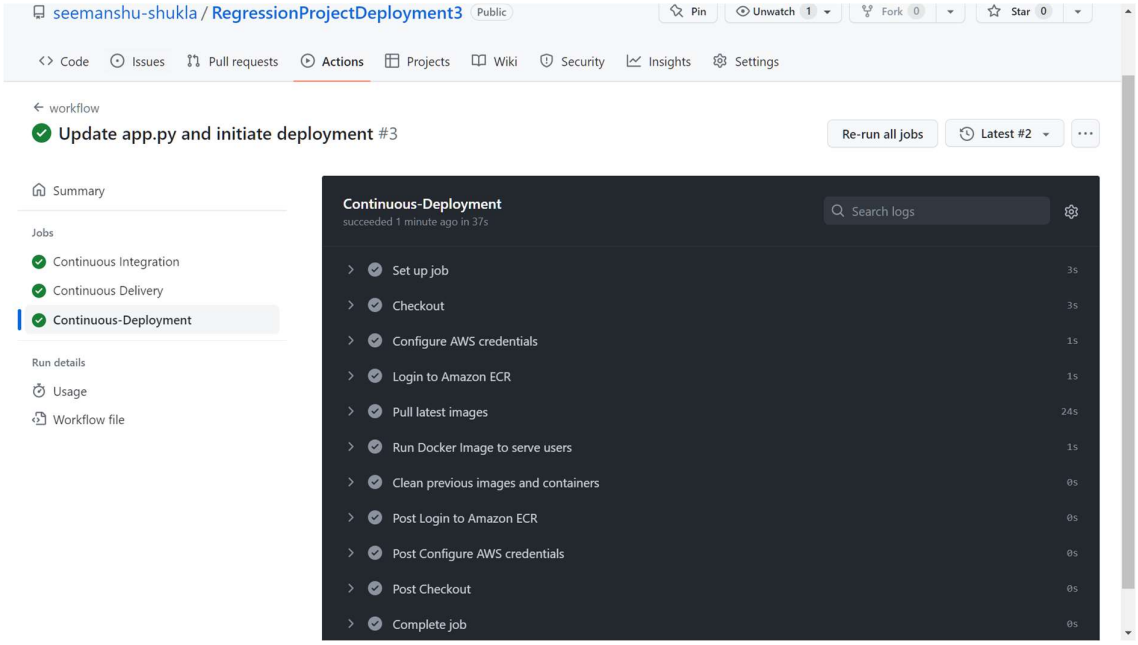
➔ Re-run the failed job by clicking the **Re-run jobs:**



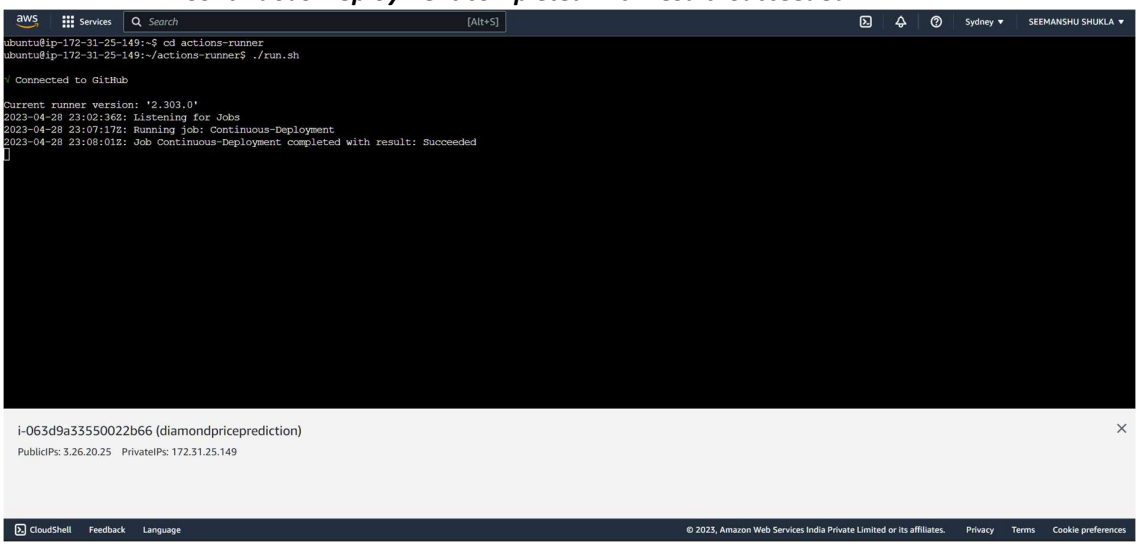
➔ Now, we can clearly visualize **Continuous-Deployment** in progress state:



➔ Once, a green check mark is added adjacent to **Continuous-Deployment** then it signifies that it is implemented successfully.



➔ Also, if we go back to our EC2 instance we can see the message ***"Job Continuous-Deployment completed with result: Succeeded"***:



12. **Some additional changes**:

➔ In workflow .yaml file we mentioned port as 8080:8080 (docker container port and instance port mapping) while running the docker image but nowhere it was configured.
So, we need to configure this port at EC2 instance as well as application(app.py) level which later will run as a docker container.

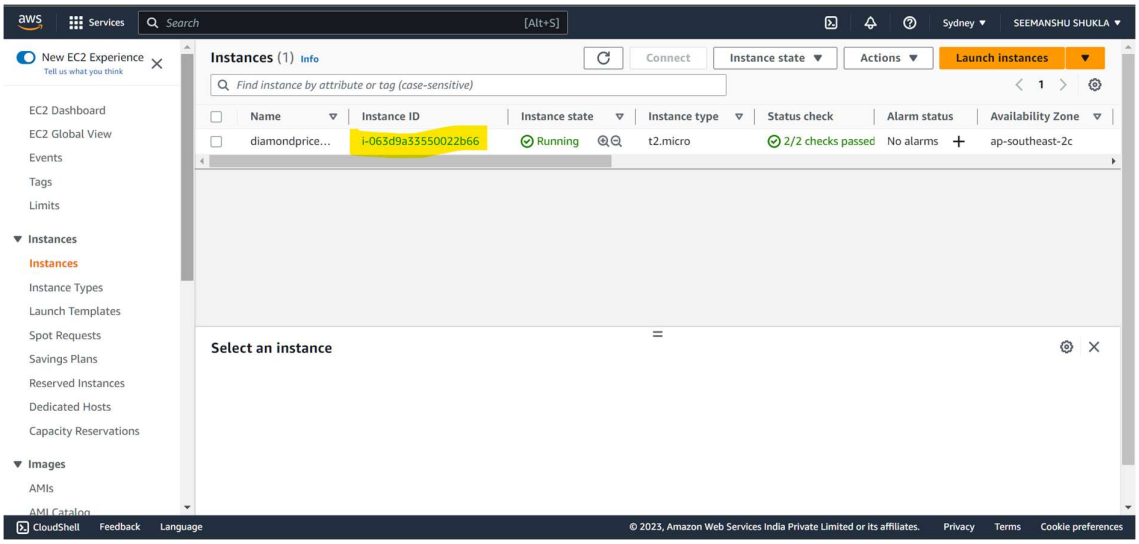
.yaml file:

github.com/seemanshu-shukla/RegressionProjectDeployment3/blob/main/.github/workflows/main.yaml

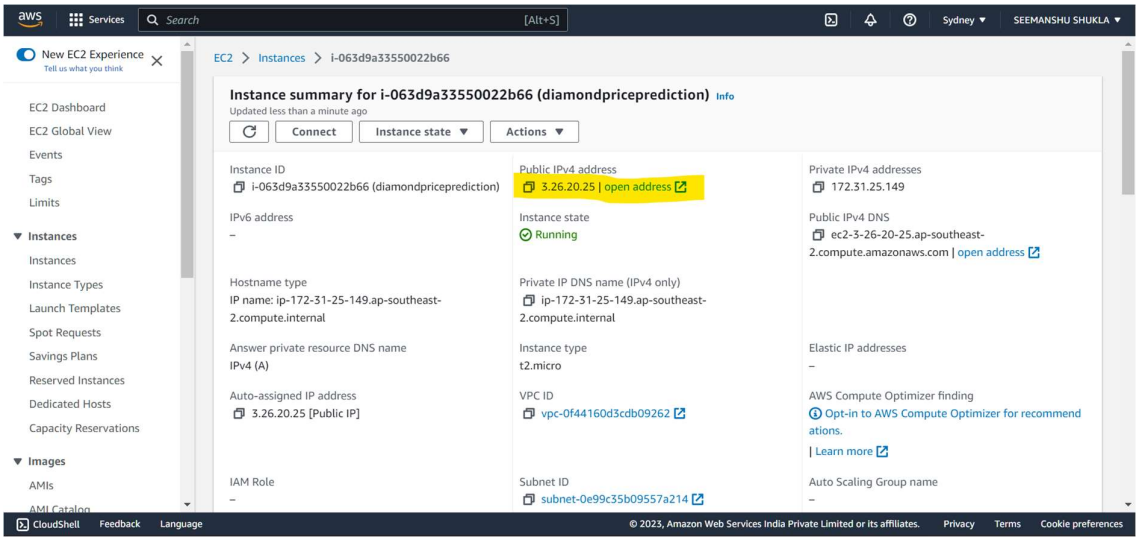
```yaml
      - name: Checkout
        uses: actions/checkout@v3

      - name: Configure AWS credentials
        uses: aws-actions/configure-aws-credentials@v1
        with:
          aws-access-key-id: ${{ secrets.AWS_ACCESS_KEY_ID }}
          aws-secret-access-key: ${{ secrets.AWS_SECRET_ACCESS_KEY }}
          aws-region: ${{ secrets.AWS_REGION }}

      - name: Login to Amazon ECR
        id: login-ecr
        uses: aws-actions/amazon-ecr-login@v1


      - name: Pull latest images
        run: |
         docker pull ${{secrets.AWS_ECR_LOGIN_URI}}/${{ secrets.ECR_REPOSITORY_NAME }}:latest

      # - name: Stop and remove  container if running
      #   run: |
      #     docker ps -q --filter "name=mltest" | grep -q . && docker stop mltest && docker rm -fv mltest

      - name: Run Docker Image to serve users
        run: |
         docker run -d -p 8080:8080 --ipc="host" --name=mltest -e 'AWS_ACCESS_KEY_ID=${{ secrets.AWS_ACCESS_KEY_ID }}' -e 'AWS_SECRET_ACCESS_KEY=${{ secrets.AWS_SECRET_ACCES
      - name: Clean previous images and containers
        run: |
         docker system prune -f
```

➔ If we run our deployed application using public domain of EC2 instance without configuring 8080 port at both EC2 instance and container/app level then we will be getting an error. To eliminate this error we need to configure 8080 port at both the mentioned levels.

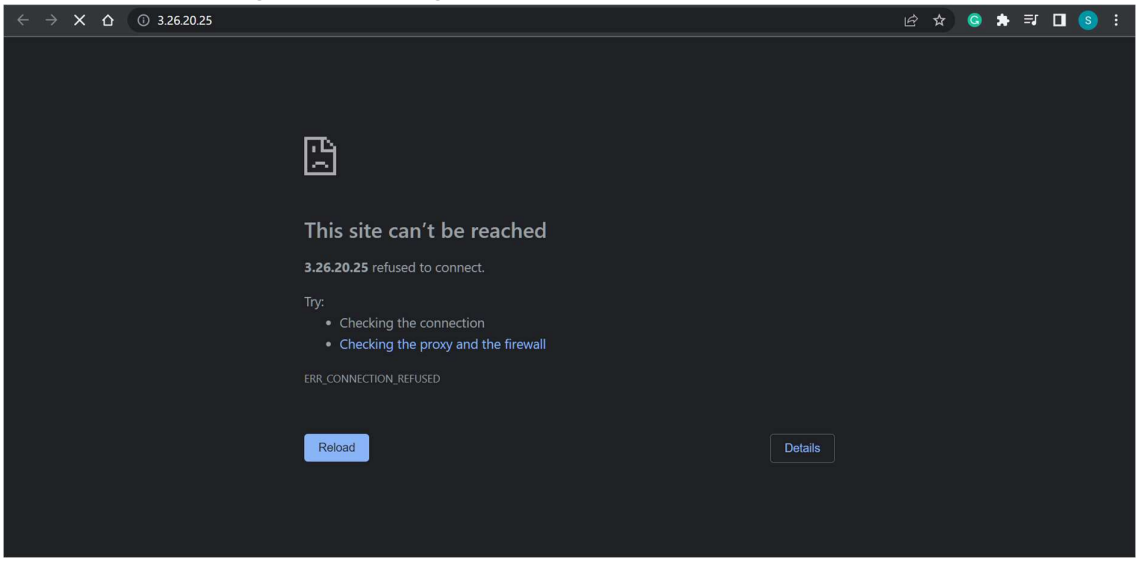
➔ Going to EC2 instance and under Instances clicking highlighted **Instance ID**:



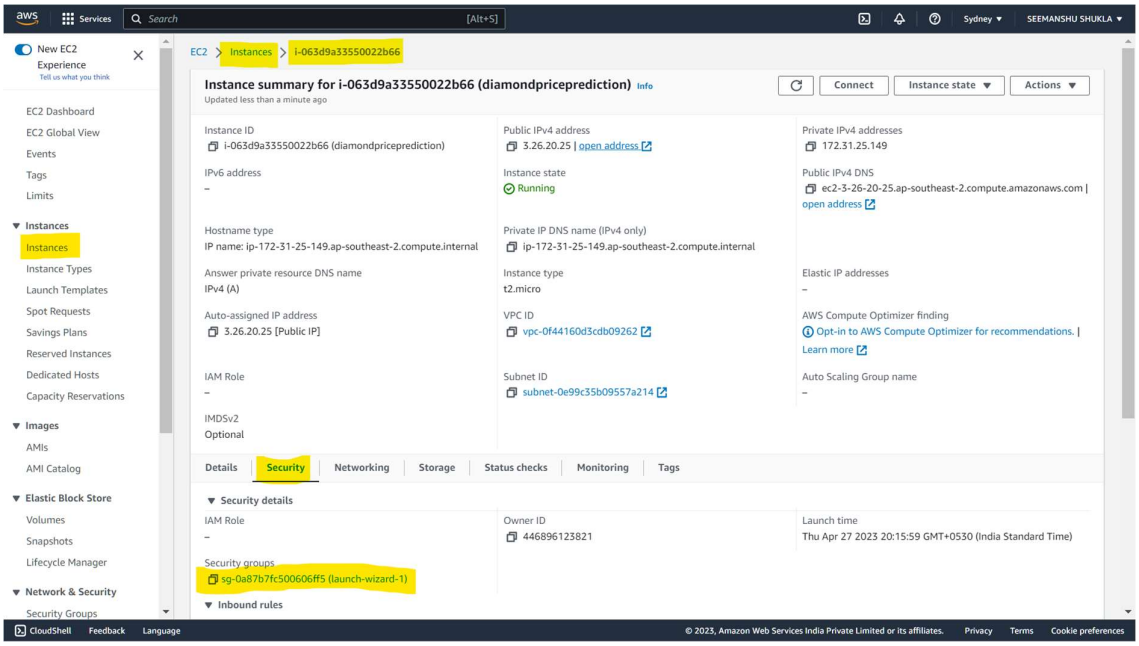
➔ Copying the **Public IPv4 address** and opening it in the new tab of browser:



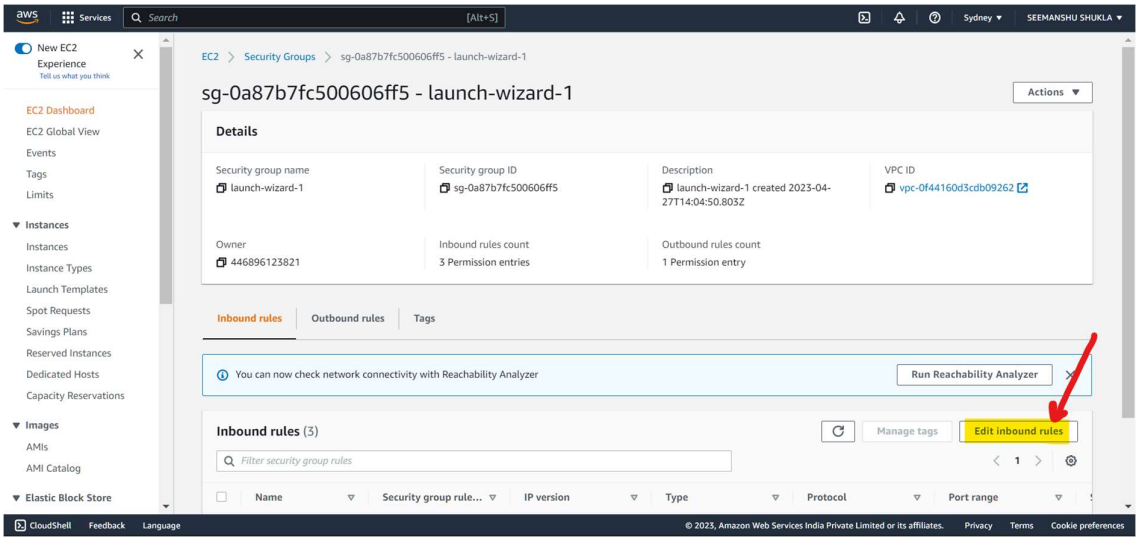
➔ Getting an error stating ***"This site can't be reached"***.



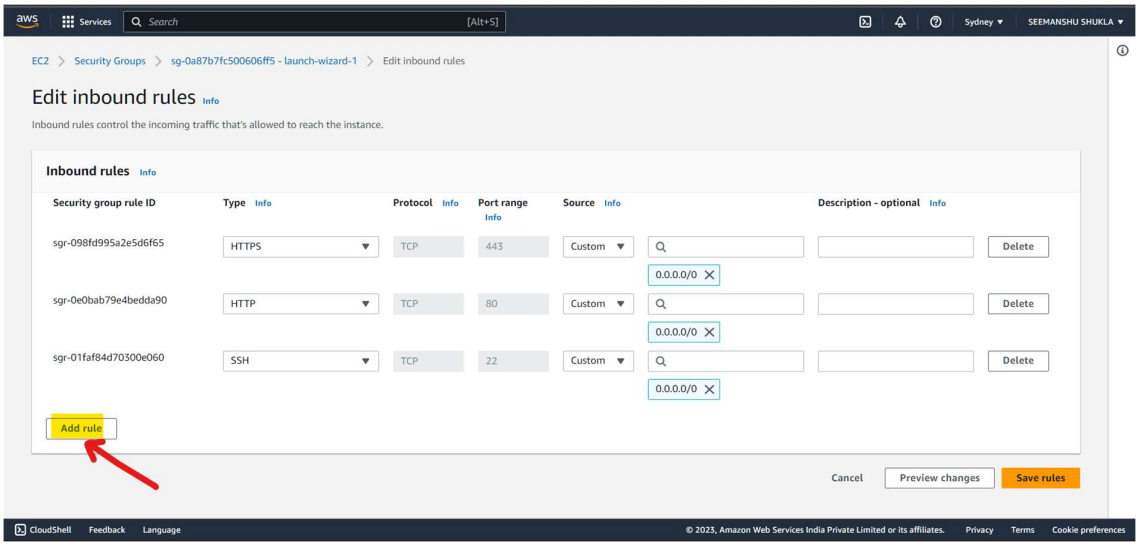
➔ To counter the above issue as discussed we will **Firstly, configure the 8080 port in EC2 instance**. Open the instance where application is deployed and open **Security groups** under **Security** section.



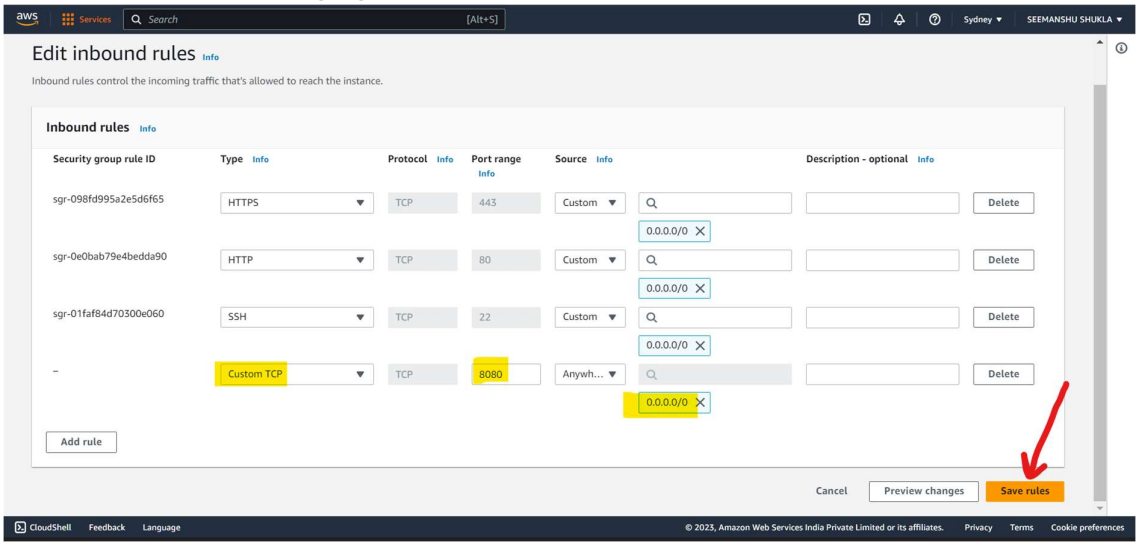
➔ Click **Edit inbound rules**:



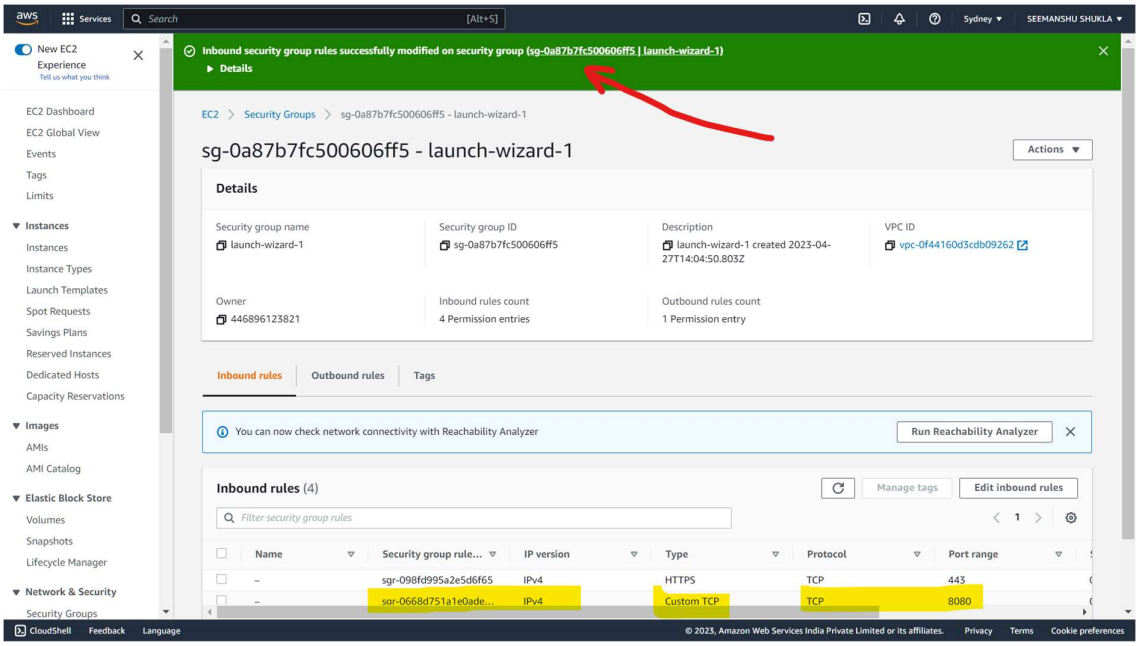
➔ Click **Add rule**:



➔ Enter the highlighted details and click **Save rules**:



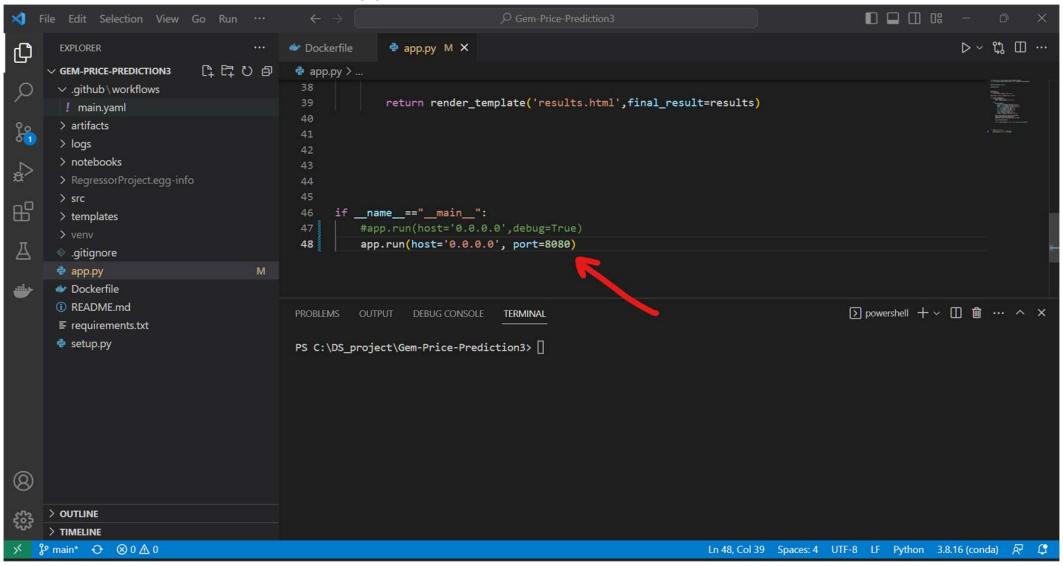
➔ Inbound rule added successfully. This means that port 8080 has been successfully configured at EC2 instance level.



➔ **<u>Secondly, we will configure 8080 port at app.py level</u>** which will be running as a docker container inside the EC2 instance (server). Changes done at app.py level will also need to be pushed to the GitHub and from GitHub to server via runner.

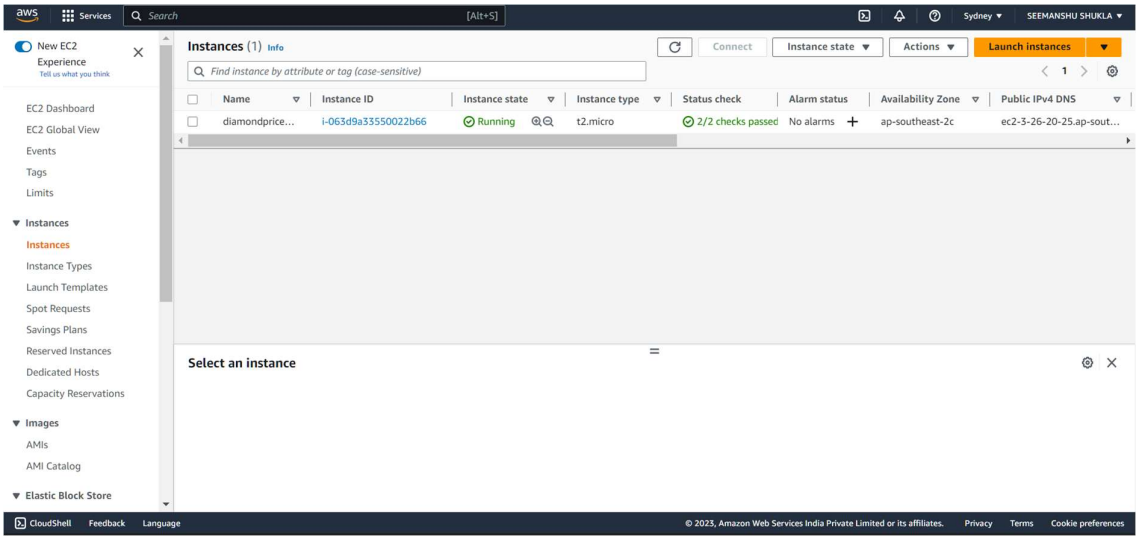
Since, app.py will run as a docker container so app.run(host=”0.0.0.0”, port=8080) will signify that run this application inside docker container’s local host using 8080 port number.

Please note, that our docker image for the entire application (app.py and all other dependencies like setup.py, src, artifacts etc.) is pushed inside AWS ECR private repository. From here, this docker image is getting pulled inside EC2 instance where this image is running as dockized or containerized application.

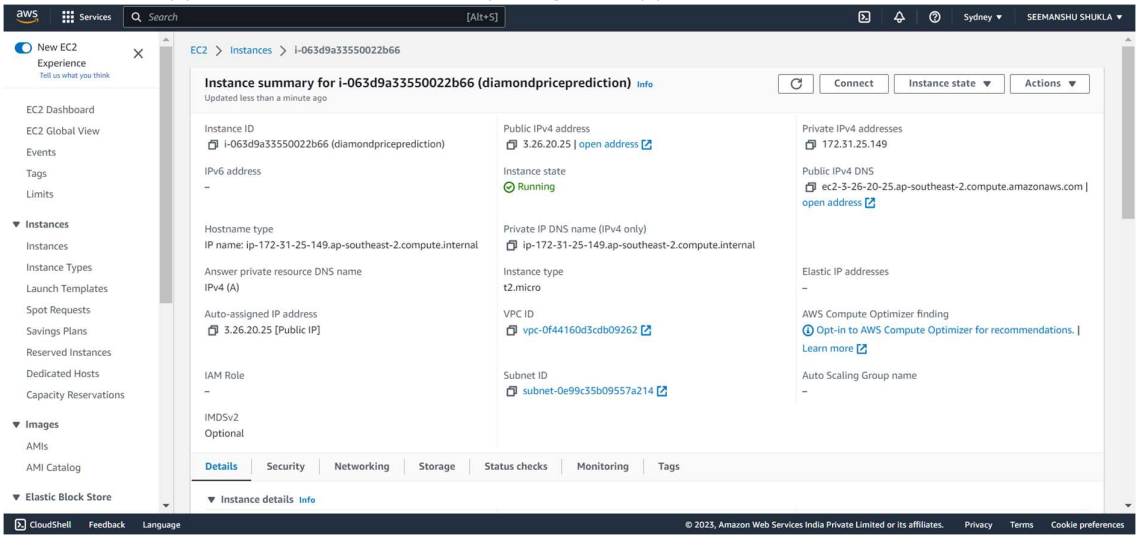


13. **<u>Accessing the deployed application</u>:**

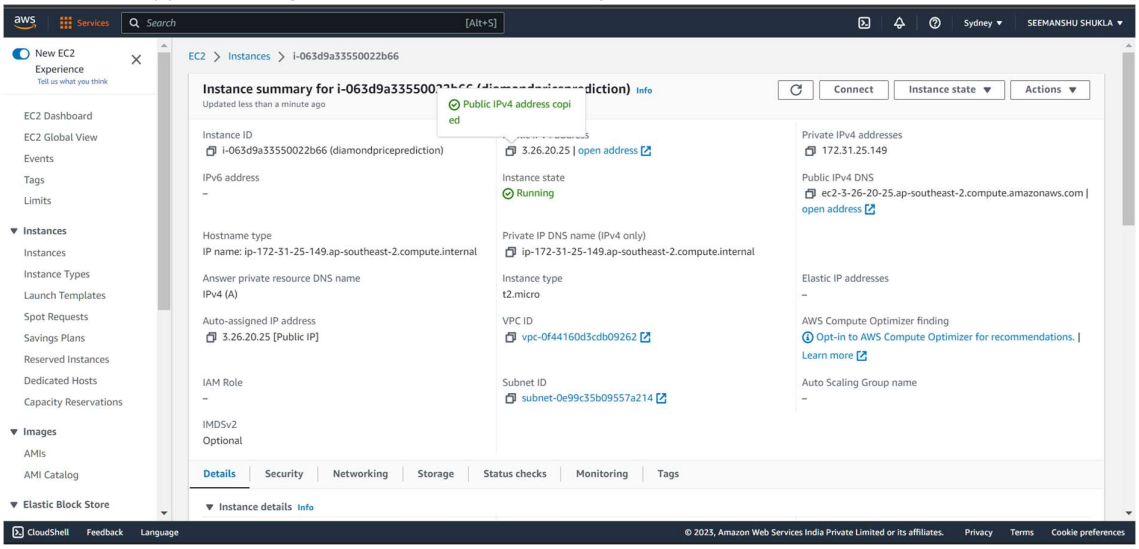
➔ Go **to EC2 instance** and click on **Instance ID** under **Instances**:



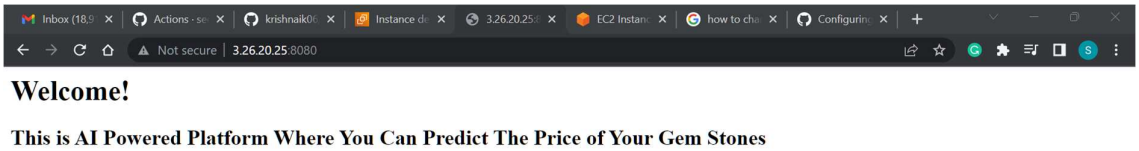
➔ Copy the **Public IPv4 address** by using the copy icon:



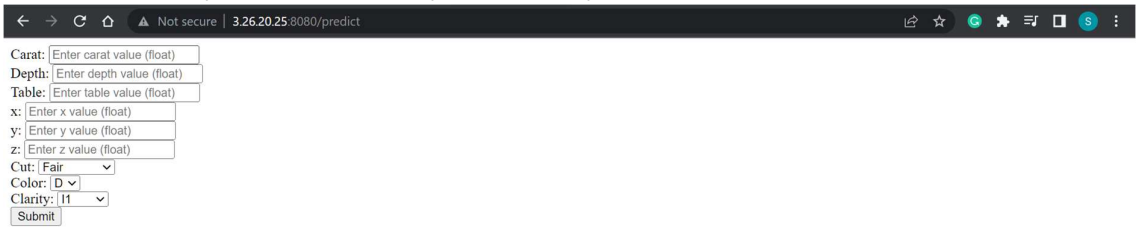
➔ Once URL is copied successfully then a green check mark with green text will appear stating the ***Public IPv4 address copied***.



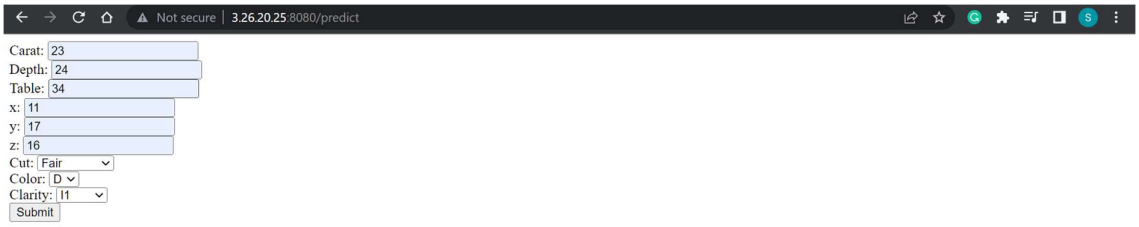
➔ Open a new tab in the current browser and paste the copied URL as shown below:



➔ User **"/predict"** to get re-directed to predict web page where we can provide the input based on which prediction on price will be made.



➔ Enter the values and click **Submit**:



➔ Getting redirected to results web page that is depicting the predicted price of the gem stone based on input features:

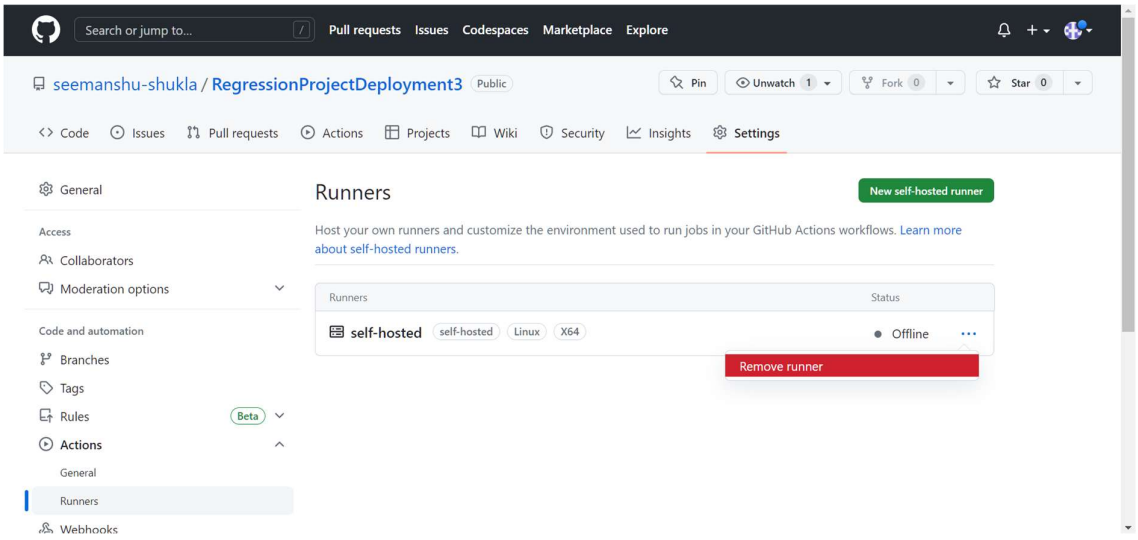
---

14. **<u>Deleting deployment</u>:**

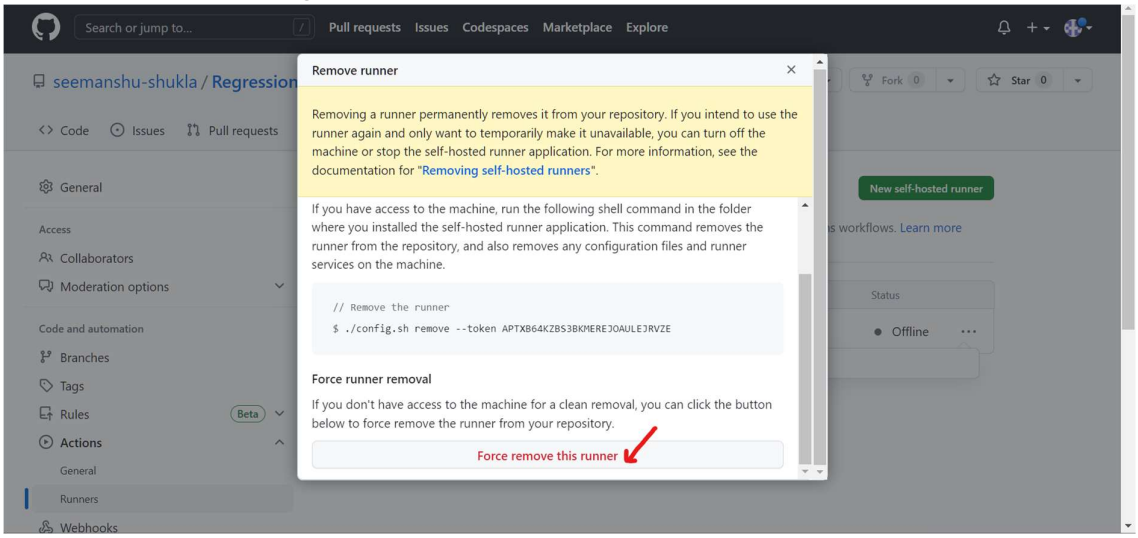
Since, this deployment was done just for learning purpose we will be deleting this deployment so that AWS does not charge us anything extra.

A. Delete **Runner** in **GitHub**:

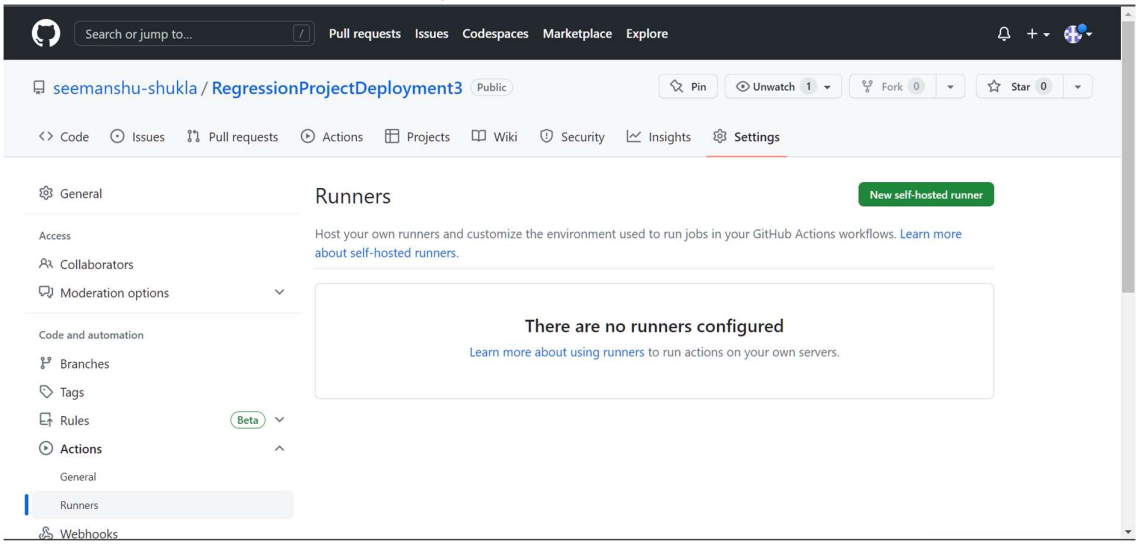
➔ Go to GitHub > Settings > Actions > Runners and select the **"…"** icon and click **Remove runner**



➔ In the following screen click **Force remove this runner**:

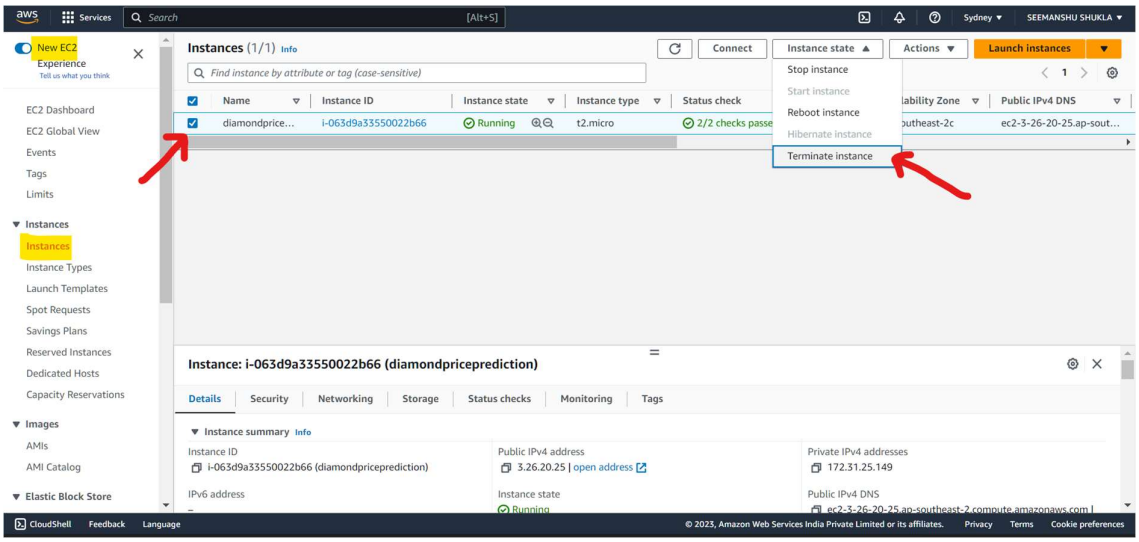


➔ Runner deleted successfully:

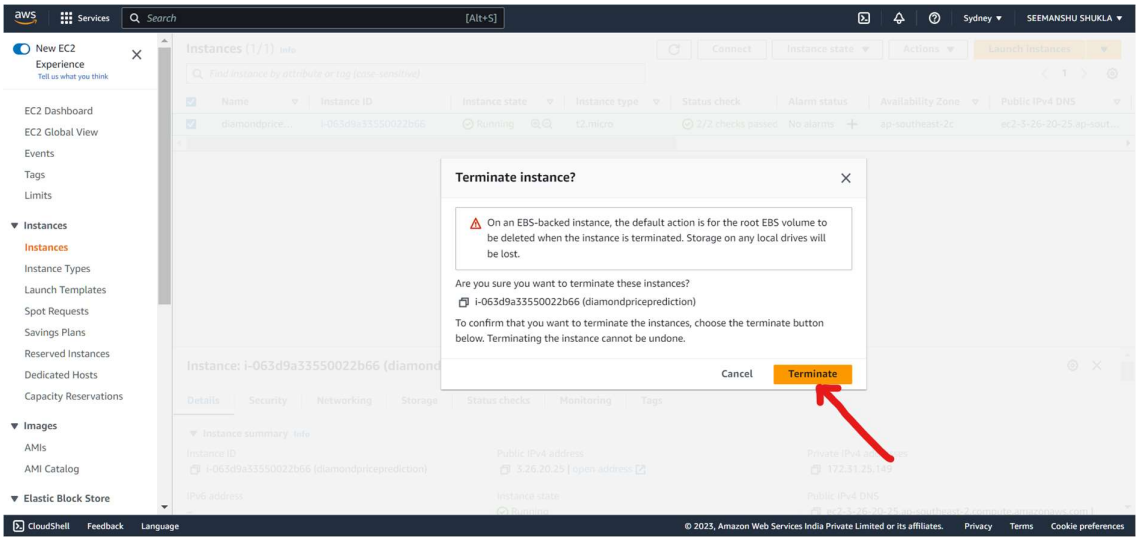


B. Deleting the **EC2 instance**:

➔ Go to **EC2 instance** > select check box for running instance > on top click **"Instance state"** drop down and select **Terminate instance:**



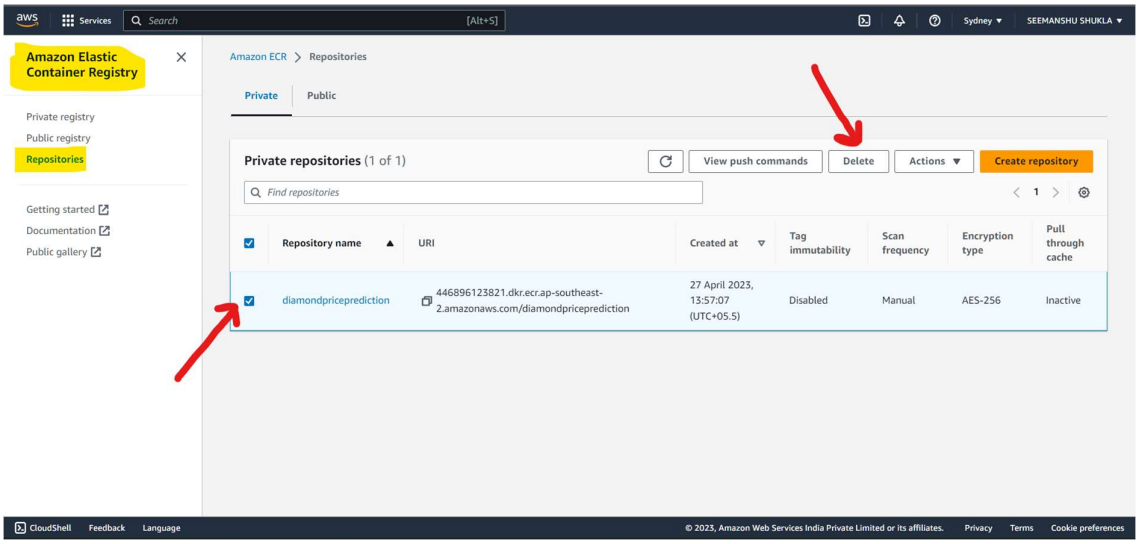
➔ Click **Terminate**:



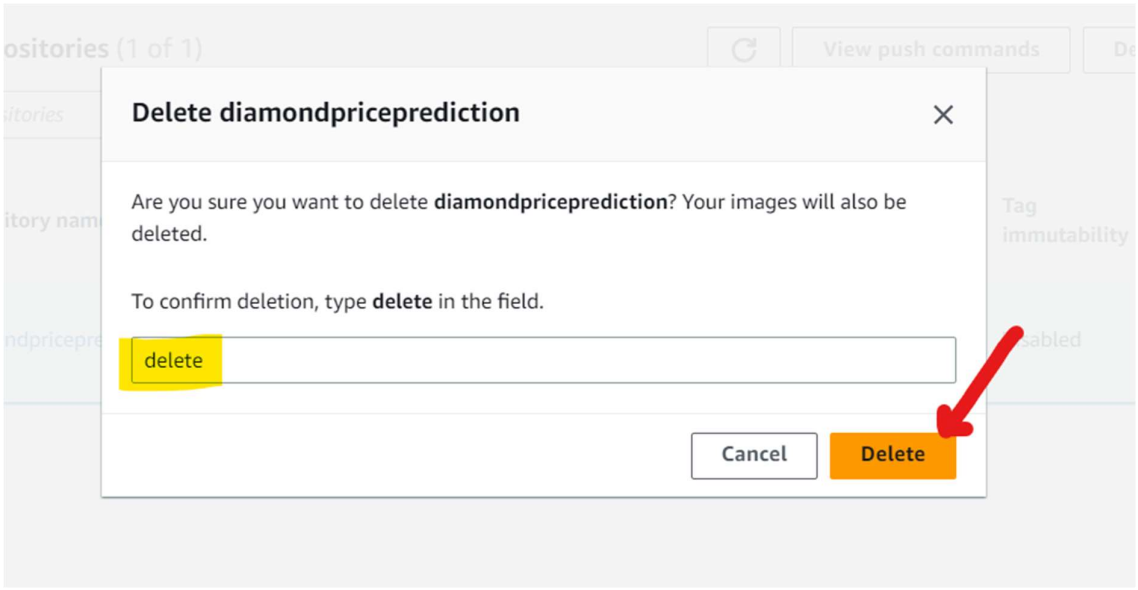
➔ Post this the **Instance state** will be changed from **Running** to **Terminated**.

C. Deleting **ECR(Elastic Container Repository)**:

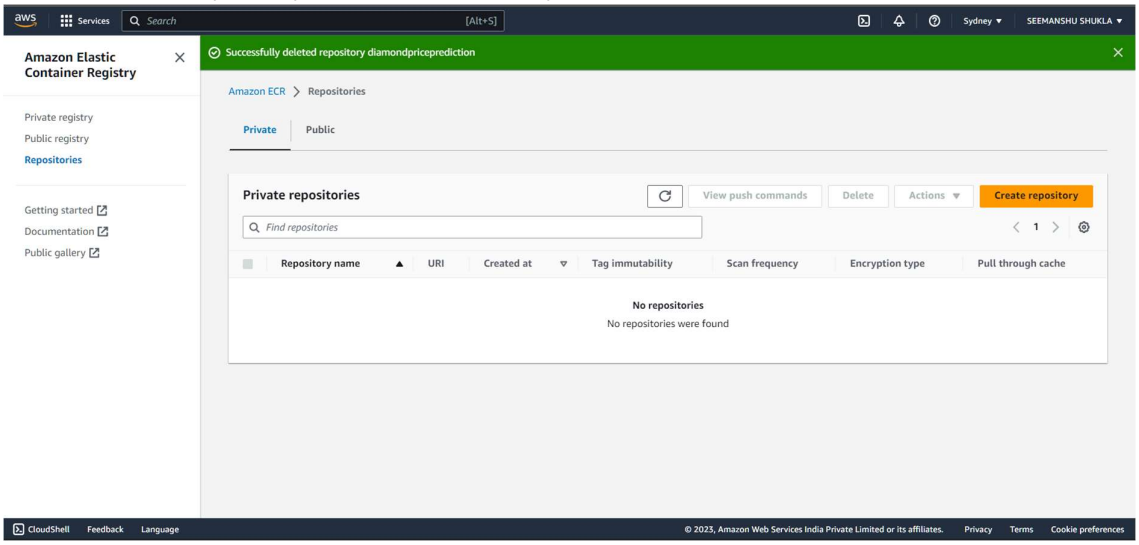
➔ Open **ECR** and under **Repositories** select the check box adjacent to repository that needs to be deleted and then on top click the **delete** option:



➔ As instructed type **delete** and click **Delete button**:

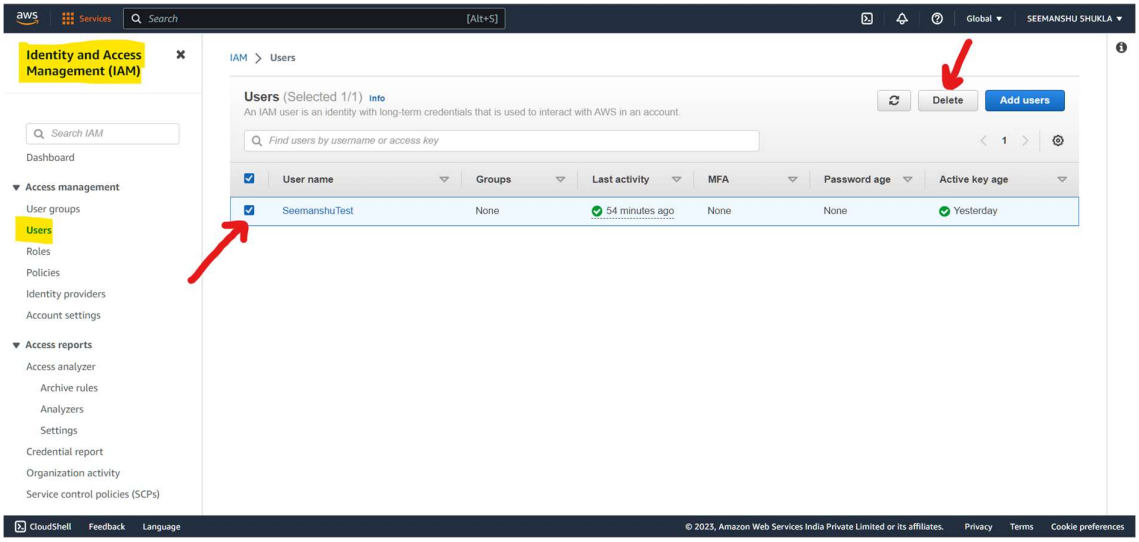


➔ ECR repository is deleted successfully:



D. **Deleting the IAM user(secrete key that got generated as .csv file which later was used to define our secrete variables):**

➔ Go to **IAM** and under **Users** select check box adjacent to the user that needs to be deleted and click **Delete** option on top:



➔ IAM user is deleted successfully: