******Follow me :- Vinay Pramanik in LinkedIn for more ******

Project 1

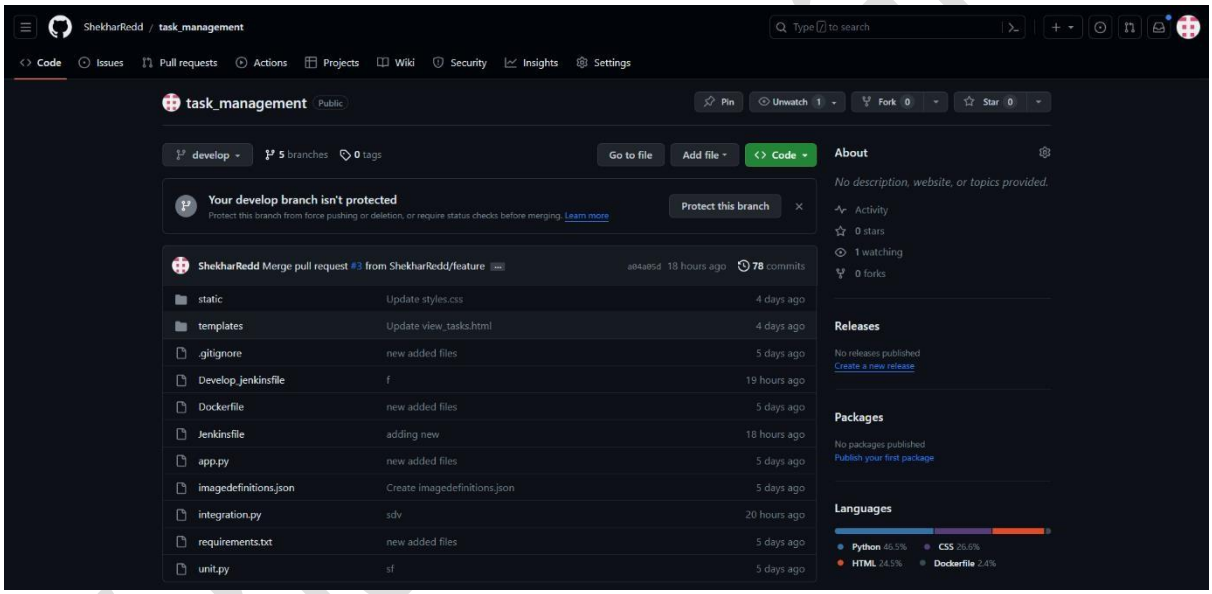
Continuous Integration and Continuous Deployment (CI/CD) Pipeline

Sub Task 1:

Set up a version control system using Git.

Create a New Repository on GitHub:

- Go to GitHub.
- Log in or create a new account.
- Click the "New" in the top left corner to create a New repository.



Pushing code to a Git repository involves a few steps using command line.

- **Navigate to Your Project Directory:**
  cd /path/to/your/project
- **Initialize a Git Repository:**
  git init
- **Add Your Files:** git add .
- **Commit Your Changes:**
  git commit -m "Initial commit"
- **Link Your Local Repository to GitHub:**

git remote add origin your_repository_url.git

- **Push Your Changes to GitHub:** git push -u origin master

Establish branching strategies (e.g., feature branches, release branches).

- Branching strategies in Git refer to the ways in which developers organize and manage branches in a Git repository.

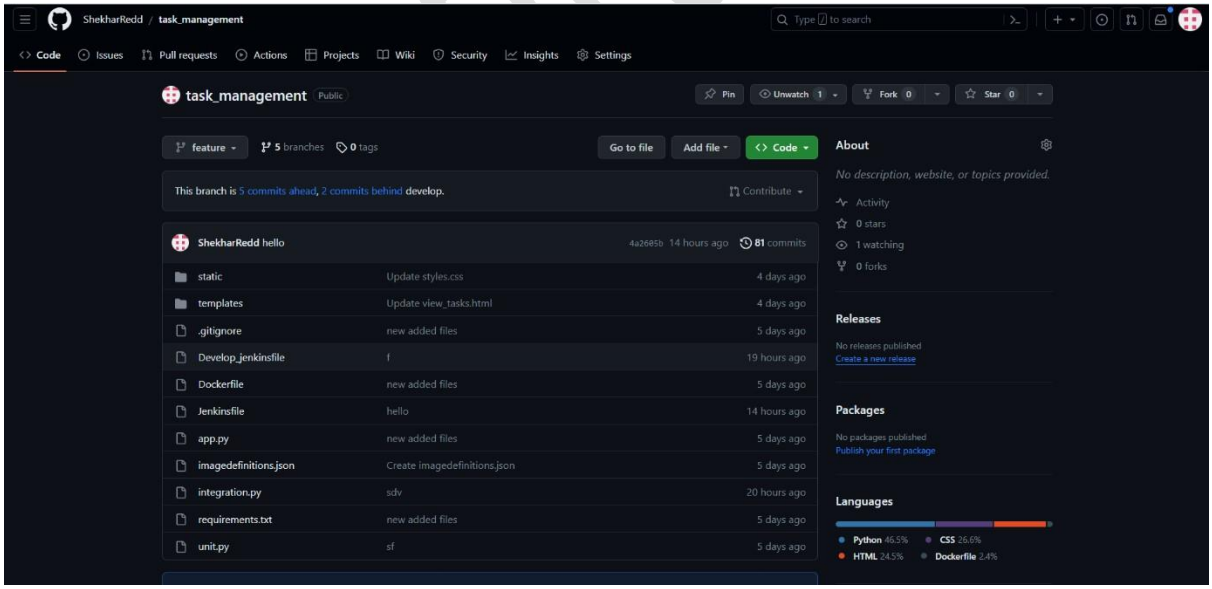
Feature branches:

- A feature branch is a copy of the main codebase where an individual or team of software developers can work on a new feature until it is complete.
- Create a new feature branch from the main branch
- git checkout -b feature

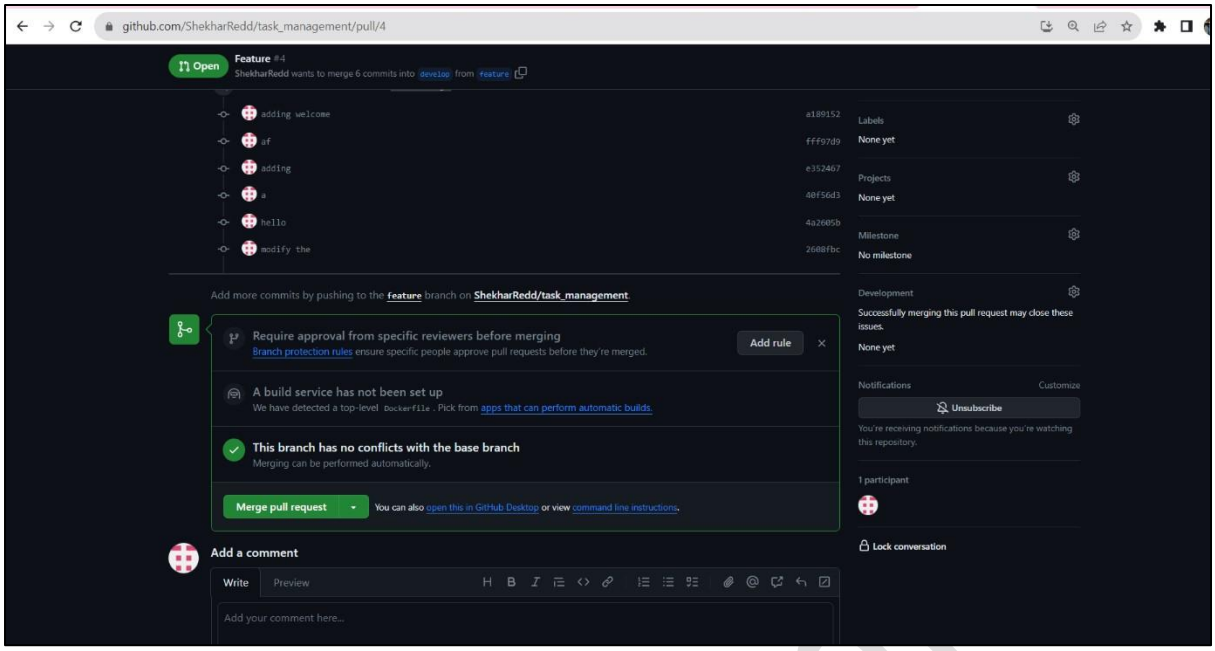
Now, you're on the new feature branch. Start making changes to your code, add new files, or implement the feature.

- **Once you've made changes, stage and commit them to the feature branch.** git add .
  git commit -m "Implement new feature"
- **If you want to push the feature branch to the repository use the command**.

  git push origin feature



- After pushing to the feature branch in your git repo it shows a banner at the top “feature” has recent pushes less than a minute ago”.
- Click on Compare & Pull request and merge the pull request. Then your feature branch is successfully merged to the main branch.



Release branch:

A release branch in Git is a branch specifically created to prepare and stabilize a software release. This branch is used to isolate the work related to finalizing and testing the code for a new version before it is deployed to production.

- **Create a Release Branch:** git checkout main git pull origin main git checkout -b release/1.0.0
- **Make Changes and Bug Fixes:**

  git add .

  git commit -m "Fix issues and prepare for release"
- **Merge Release Branch into master:**

  git checkout master git

  merge release/1.0.0
- **Tag the Release:**

  git tag -a v1.0.0 -m "Release version 1.0.0"
- **Push Changes and Tags:**

  git push origin main git

  push –tags

Git commit message conventions:

Git Conventional Commit messages follow a specific format and convention to standardize the way developers write commit messages. This convention is particularly popular in projects that use semantic versioning and automated release processes.

A Conventional Commit message follows a structured format:

<type>(<scope>): <description>

[optional body]

[optional footer]

- **<type>**: Describes the purpose of the commit (e.g., feat, fix, chore).
- **<scope>**: Specifies the scope of the commit (optional but recommended).
- **<description>**: Briefly describes the changes introduced by the commit.
- **[optional body]**: Provides additional details about the changes (optional).
- **[optional footer]**: Includes any additional information or references (optional).

Example Conventional Commit: git commit -m "feat(user-auth):

implement two-factor authentication"

- Added support for two-factor authentication in the user authentication module.
- Implemented time-based one-time passwords (TOTP) for enhanced security.
- Updated user interface to prompt users for additional verification during login.
- Unit and integration tests have been added to ensure the security and functionality of two-factor authentication.
- Resolves #123

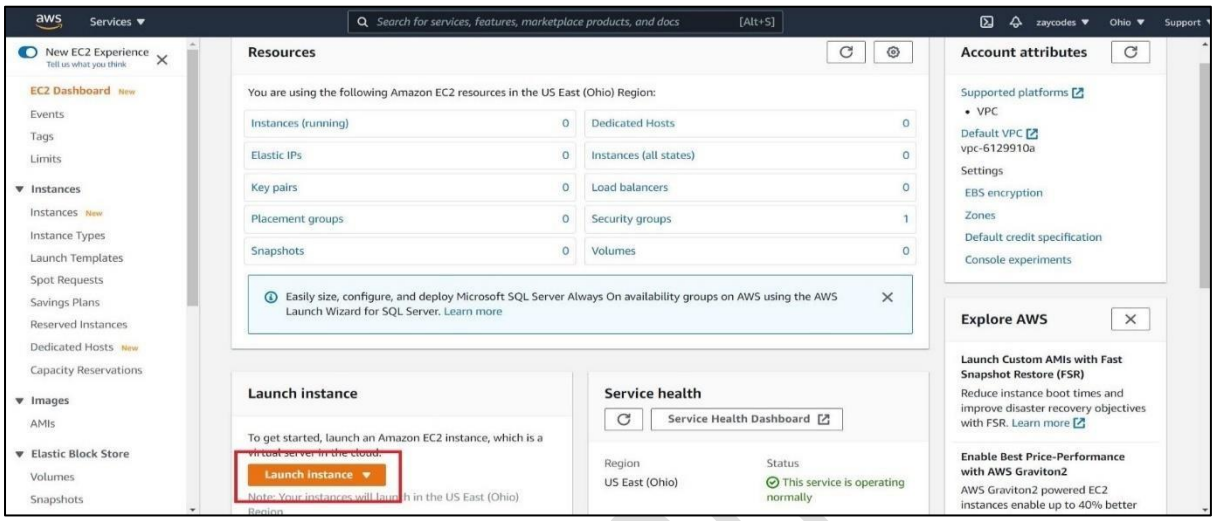
- **Type (feat):** Indicates that it's a new feature.
- **Scope (user-auth):** Specifies the scope of the changes, in this case, the user authentication module.
- **Description (implement two-factor authentication):** Briefly describes the purpose of the commit.
- **Details (- Added support...):** Provides additional details about what the commit does. Bullet points are used for clarity.
- **Reference (Resolves #123):** Includes a reference to an associated issue or task in your issue tracking system

Sub Task 2 :

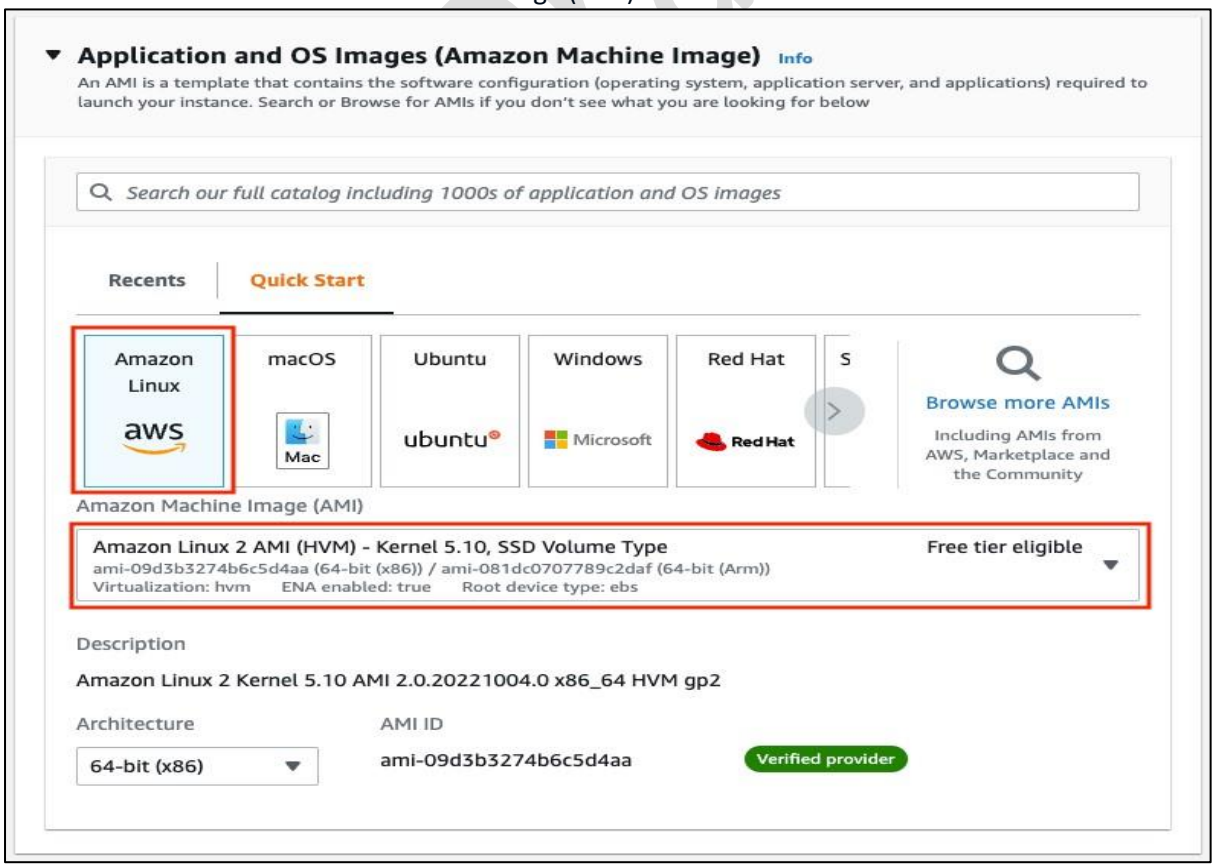
Configuring Jenkins on AWS EC2 – LINUX:

Launching an Amazon EC2 instance:

- Sign in to the AWS Management Console.
- Open the Amazon EC2 console by selecting EC2 under **Compute**. From the Amazon EC2 dashboard, select **Launch Instance**.



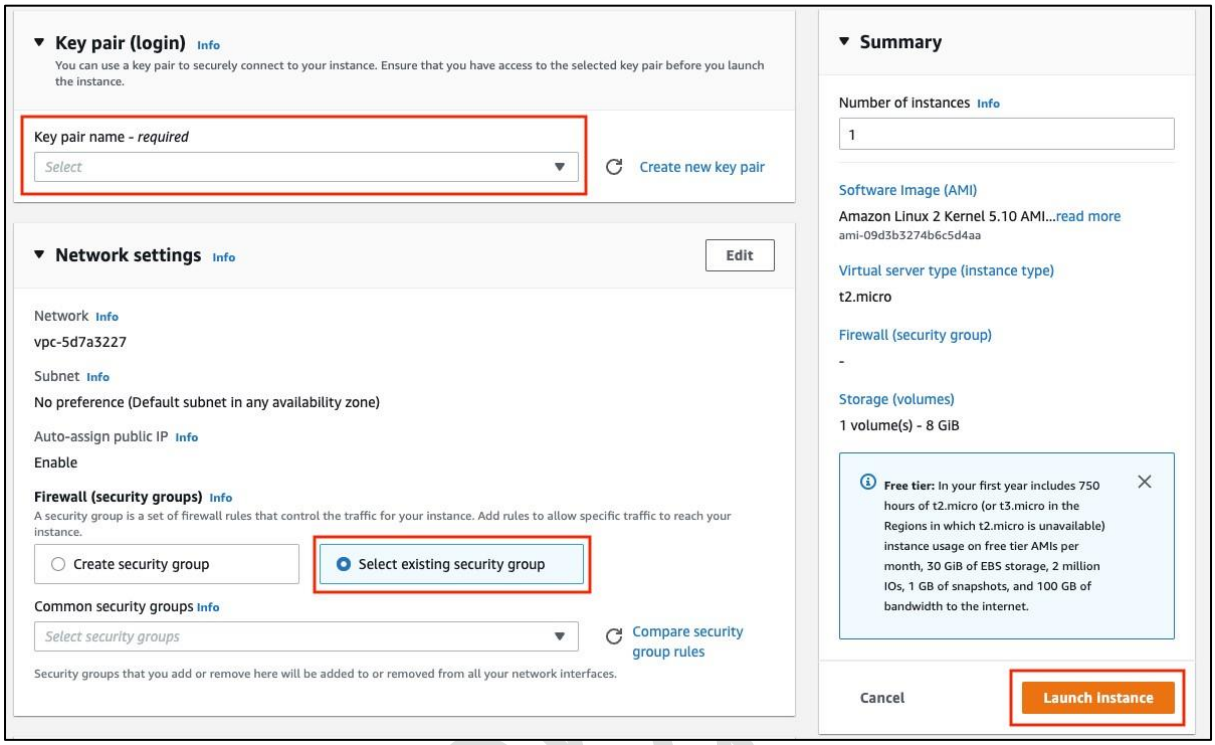
- Choose an Amazon Machine Image (AMI). Select Amazon Linux AMI



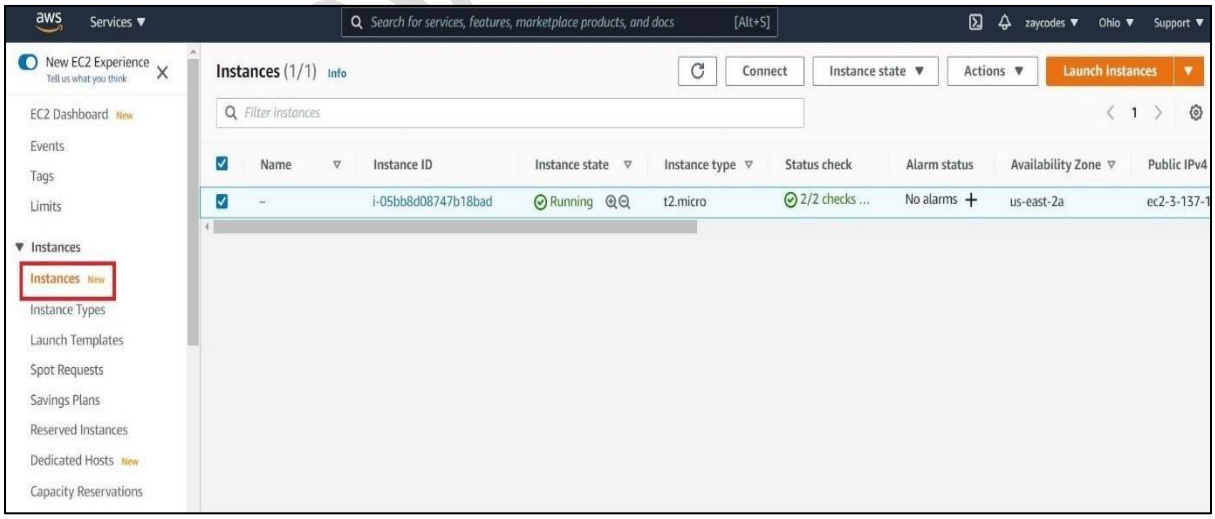
- Scroll down and select the key pair you have already created or create a new key pair.

- Select an existing security group or create a new security group Select the security group.

 Select **Launch Instance**.



- In the left-hand navigation bar, choose **Instances** to view the status of your instance. Initially, the status of your instance is pending. After the status changes to running, your instance is ready for use.



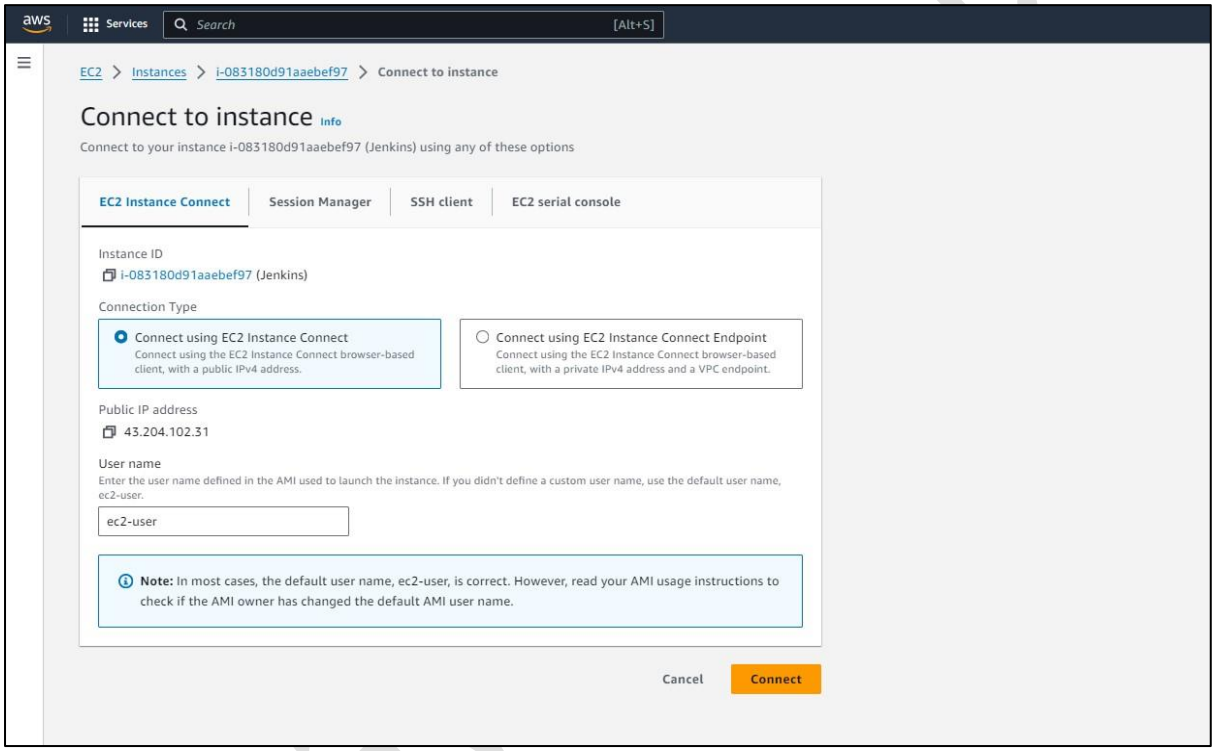
- Now that the Amazon EC2 instance has been launched, Jenkins can be installed properly.

In this step you will deploy Jenkins on your EC2 instance by completing the following tasks:

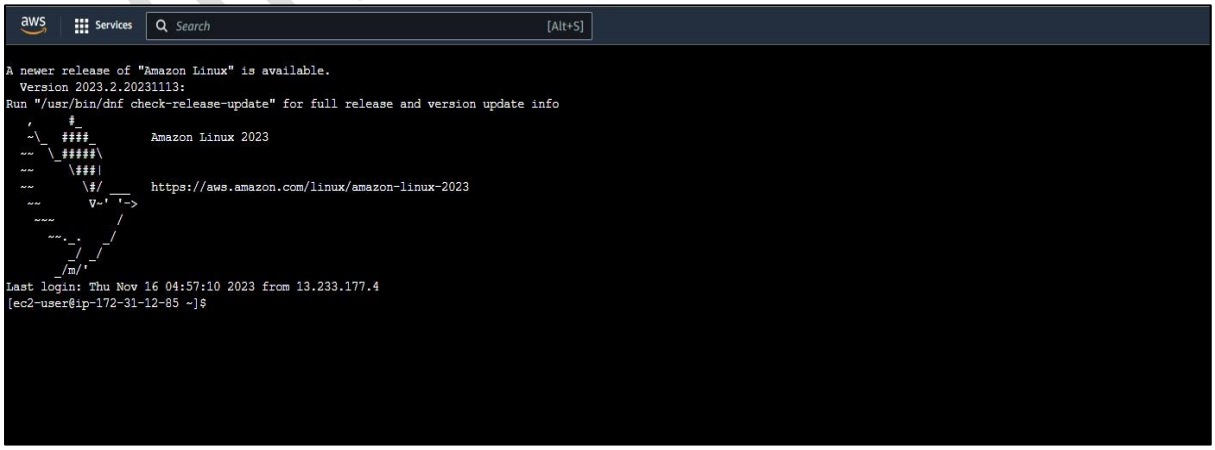
1. Connecting to your Linux instance
2. Downloading and installing Jenkins using docker
3. Configuring Jenkins

To connect to your instance using the browser-based client from the Amazon EC2 console:

- In the navigation pane, choose **Instances**. Select the instance and choose **Connect**.
- Choose **EC2 Instance Connect**.



- Verify the User name and choose **Connect** to open a terminal window.



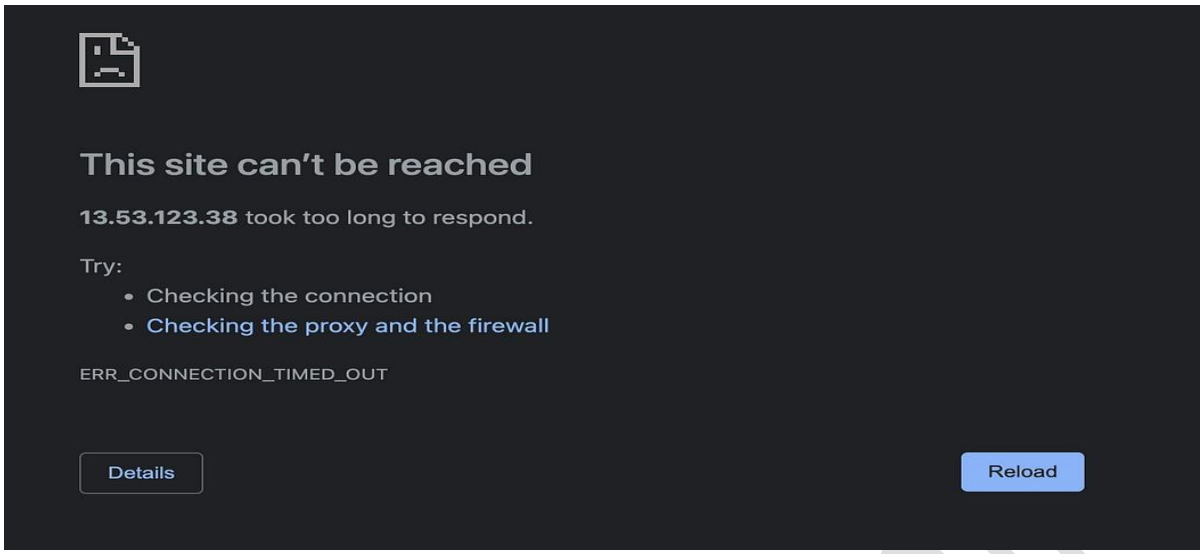
Downloading and installing Jenkins:

- Ensure that your software packages are up to date on your instance **sudo yum update**
- First, we need to complete the Docker installation.
  sudo yum install docker
- To run Docker commands without using sudo each time, we should execute the commands below.
  sudo groupadd docker sudo
  usermod -aG docker ${USER}
- To check Docker, let's run the command below.
  docker info

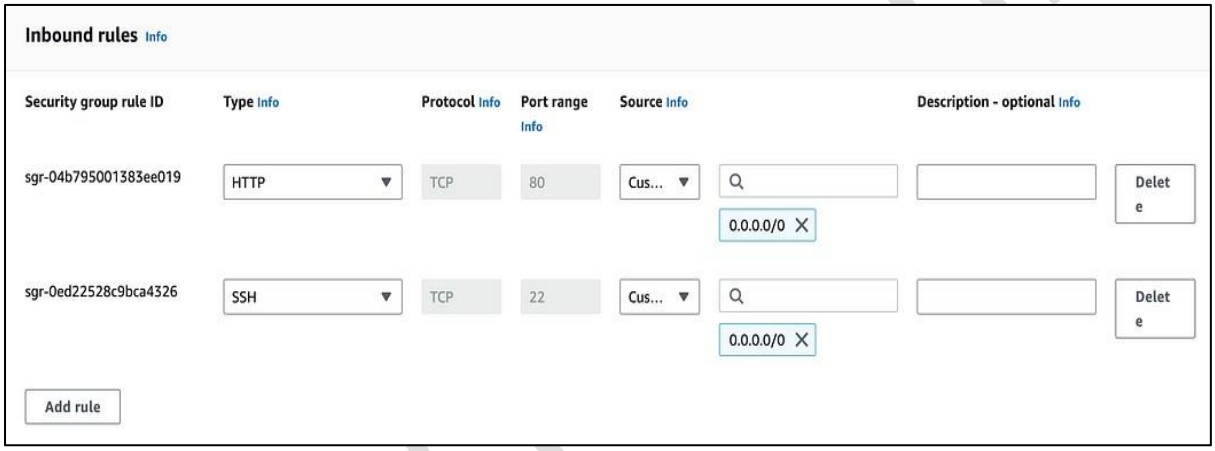
```
[ec2-user@ip-172-31-21-151 ~]$ docker info
Client:
 Version:    24.0.5
 Context:    default
 Debug Mode: false
 Plugins:
  buildx: Docker Buildx (Docker Inc.)
    Version:  v0.0.0+unknown
    Path:     /usr/libexec/docker/cli-plugins/docker-buildx

Server:
ERROR: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?
errors pretty printing info
[ec2-user@ip-172-31-21-151 ~]$
```

- To start docker **sudo service docker start sudo systemctl enable docker.service**
- Next, we can start the Jenkins container.
- Before starting the container we need to create the volume to attach the volume to the Jenkins container.
- Jenkins volumes provides a way to persist and manage data generated by docker containers.
- To create a volume **docker volume create jenkins_home**
- Run the container **docker run -u root -d -p 80:8080 -v jenkins_home:/var/jenkins_home –v /var/run/docker.sock:/var/run/docker.sock --name jenkins jenkins/Jenkins**
- The purpose of mounting the /var/run/docker.sock file into a Jenkins container is to allow the Jenkins instance to interact with the Docker daemon on the host machine.
- The -v /var/run/docker.sock:/var/run/docker.sock option maps the Docker daemon's socket on the host (/var/run/docker.sock) to the same path inside the container.

- When we try to access Jenkins, as seen in the image, the page is unreachable.



- For this, we need to add inbound rules to the Security Group.

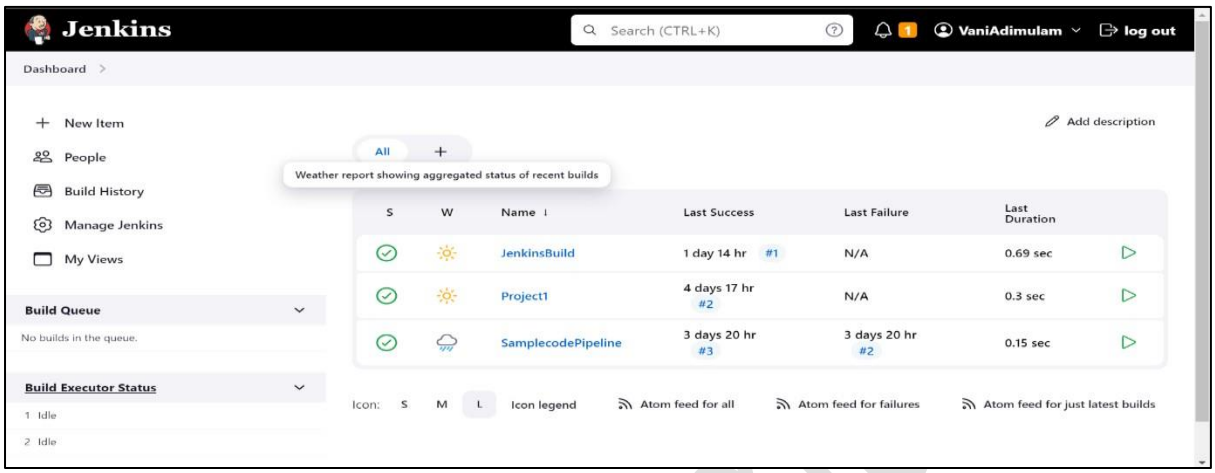


- Now we can access Jenkins.

- To obtain the password, we first need to connect to the Jenkins container and print the "initialAdminPassword" to the console. **docker exec –it jenkins bash cat /var/jenkins_home/secrets/initialAdminPassword**

  Now Copy the text and paste in the Administrator password

- After successful authentication we can access the Jenkins page:



- Execute this command in the Jenkins container to install the docker inside the Jenkins container.

  curl https://get.docker.com/ > dockerinstall && chmod 777 dockerinstall && ./dockerinstall

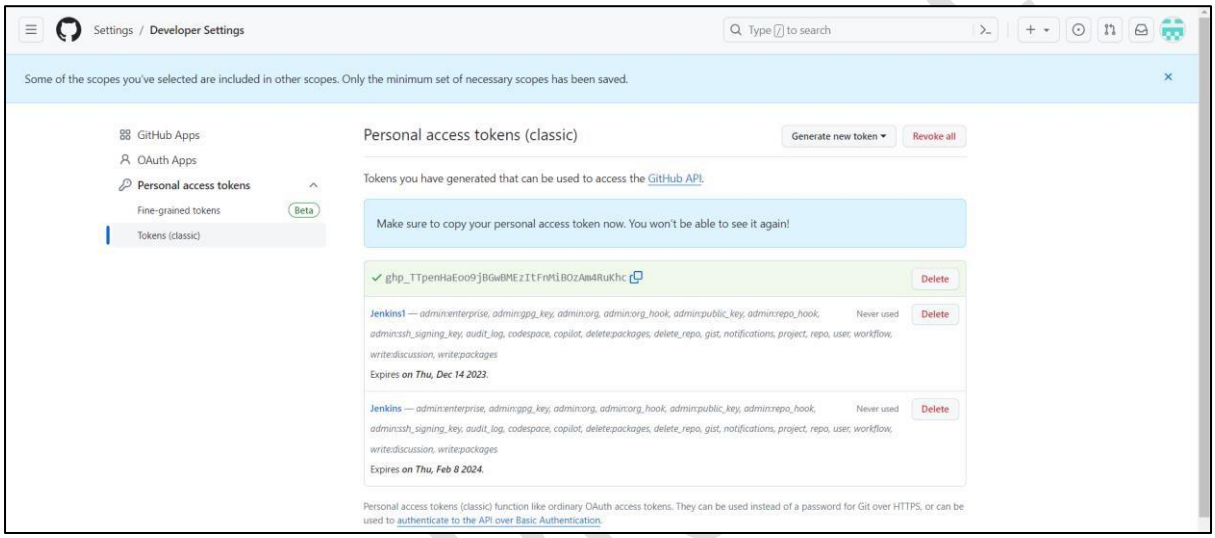
- chmod 666 /var/run/docker.sock to give the permissions to the user.

Sub Task 2:

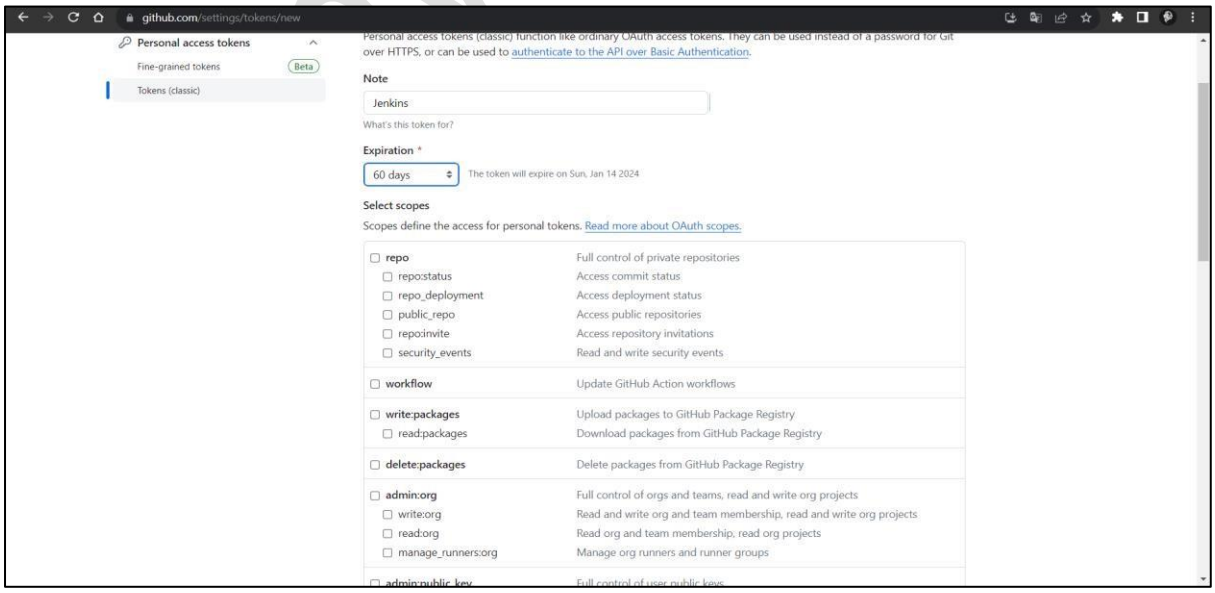
Configuring Jenkins to monitor the Git repository:

Creating Personal access token:

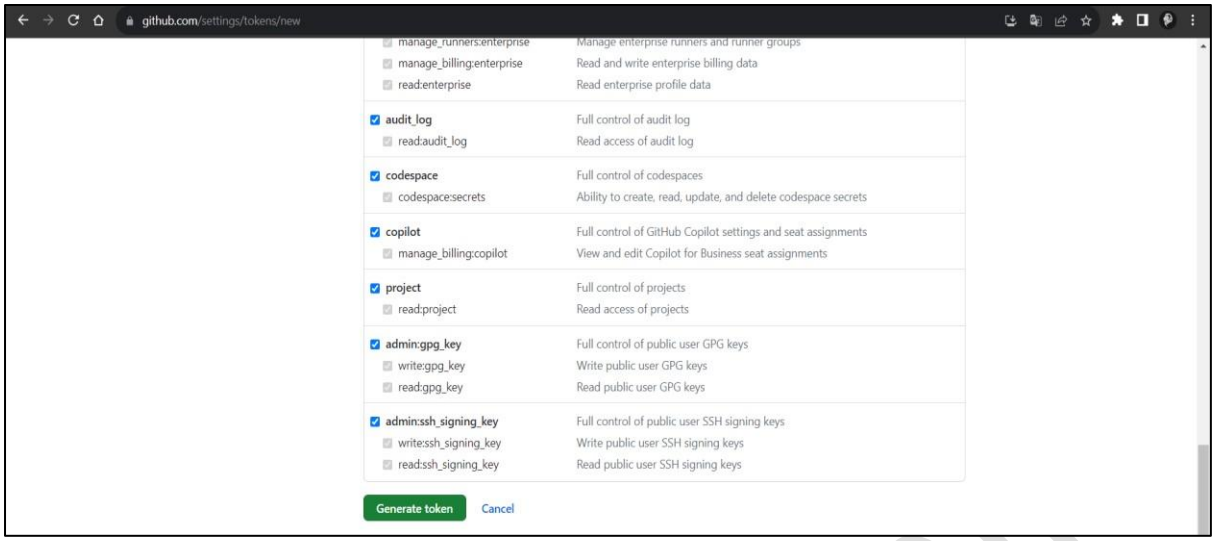
- Go to GitHub and log in to your account.
- Click on your profile picture in the top right corner and select "Settings."
- In the left sidebar, click on "Developer settings."
- Click on "Personal access tokens."



- Click the "Generate new token" button. Enter a name for your token.



- Choose the desired scopes or permissions for your token based on your requirements.



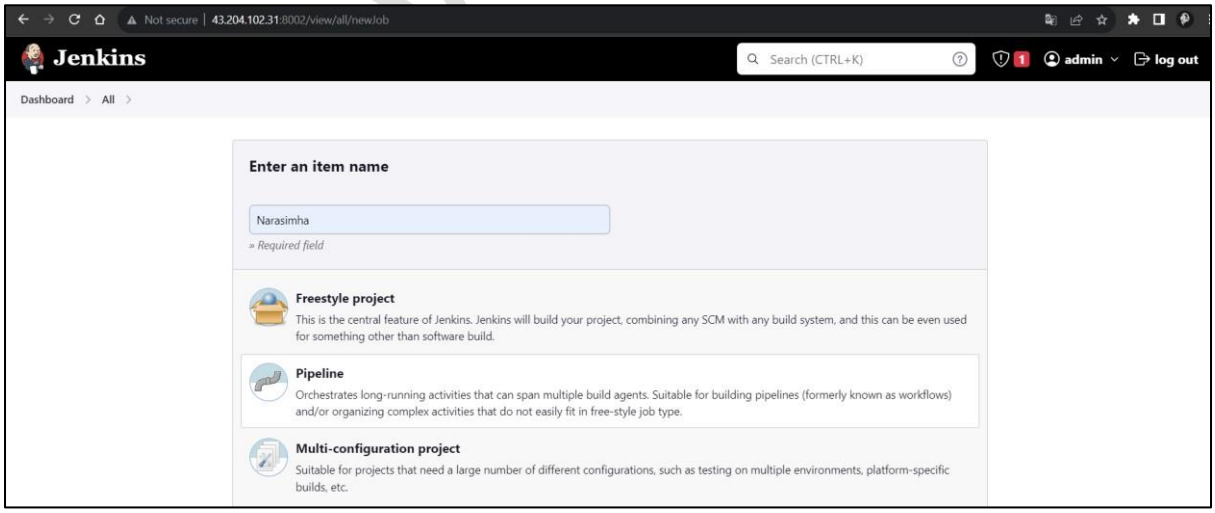
- Scroll down and click the "Generate token" button.
- Copy the generated token and store it in a secure place. Once you navigate away, you won't be able to see it again.

Webhook:

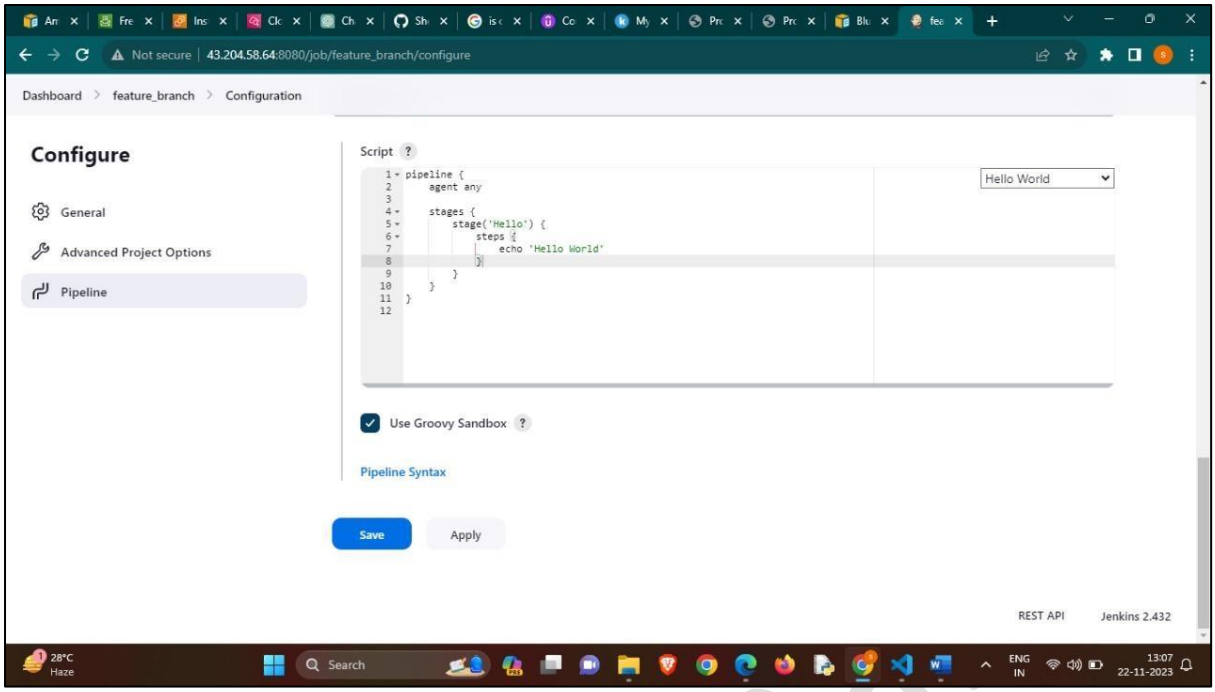
A webhook is a mechanism to automatically trigger the build of a Jenkins project upon a commit pushed in a Git repository.

In order for builds to be triggered automatically by PUSH and PULL REQUEST events, a Jenkins Web Hook needs to be added to each GitHub repository. You need admin permissions on that repository.

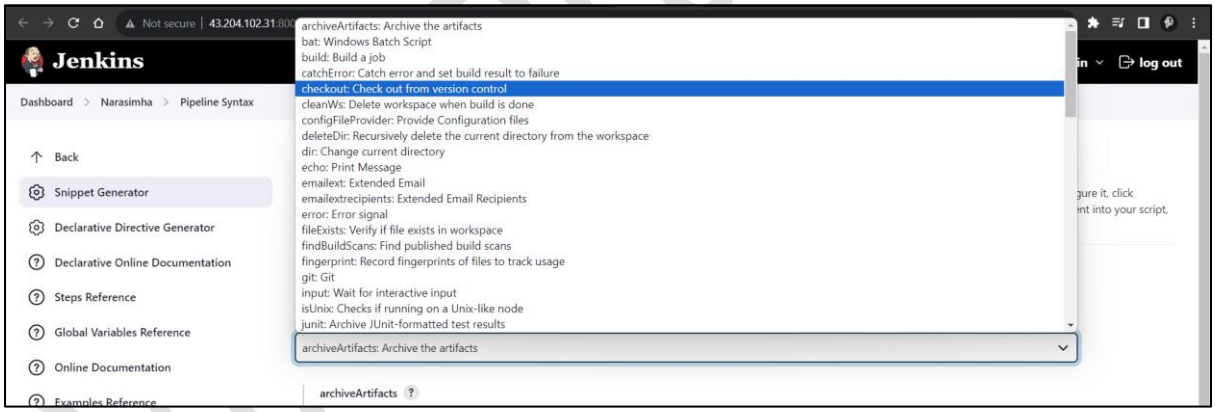
- "Github" Plugin must be installed on Jenkins.
- Create a new Jenkins job or edit an existing one with "pipeline".



- Run a sample code from pipeline script. This is the sample code you can get.

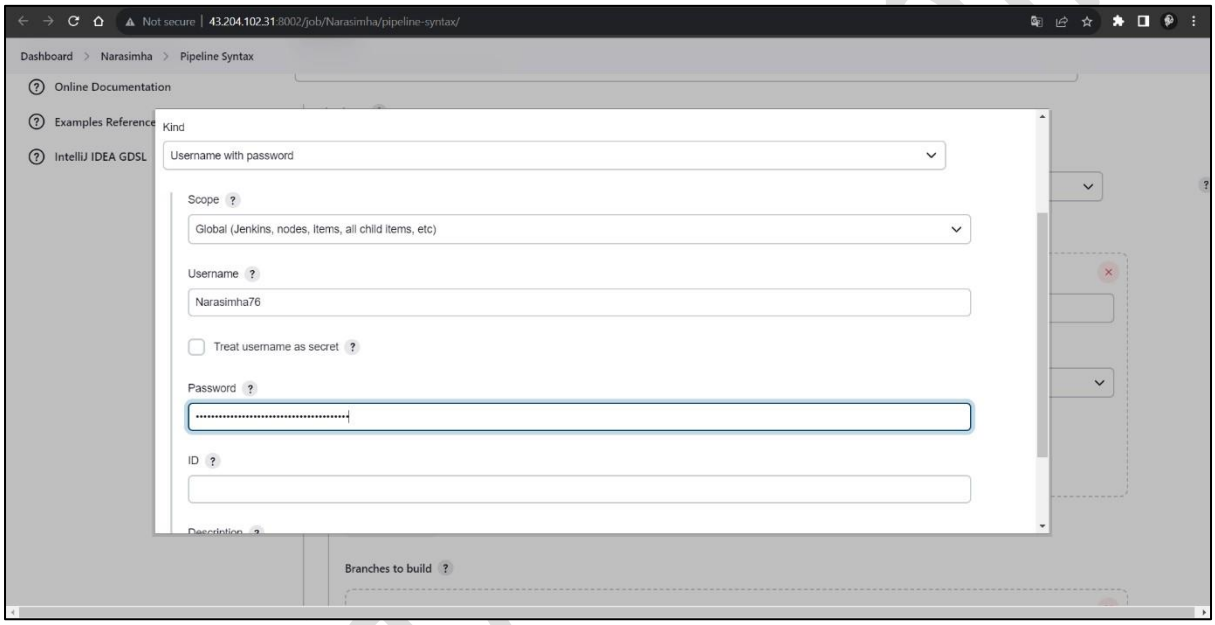


- Click on Pipeline Syntax.
- In the Pipeline syntax , Sample step select “checkout: Check out from version control”.

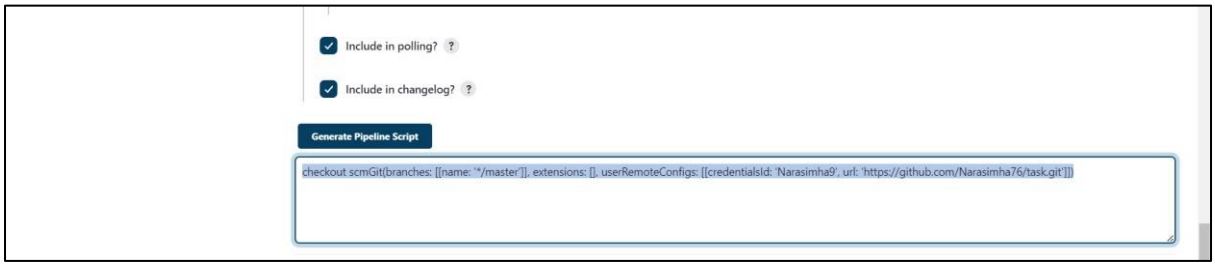


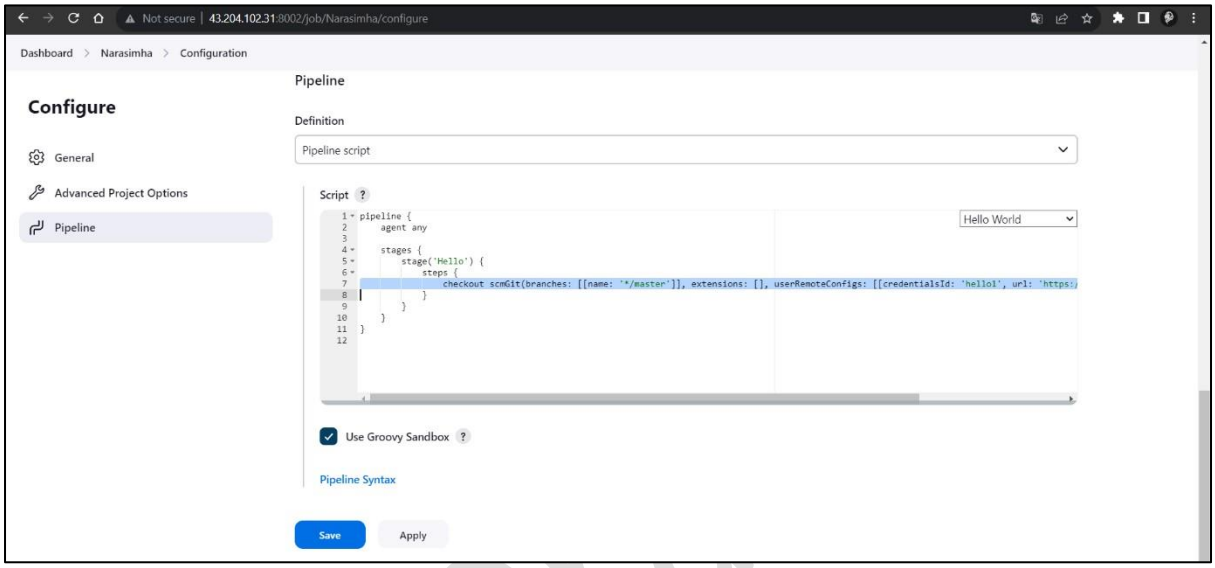
- Add GitHub repository URL here.

- In the credentials input add GitHub Username and add the Personal access token that we have created in the previous step.



- Click on Generate Pipeline script. You will get a Script copy it.

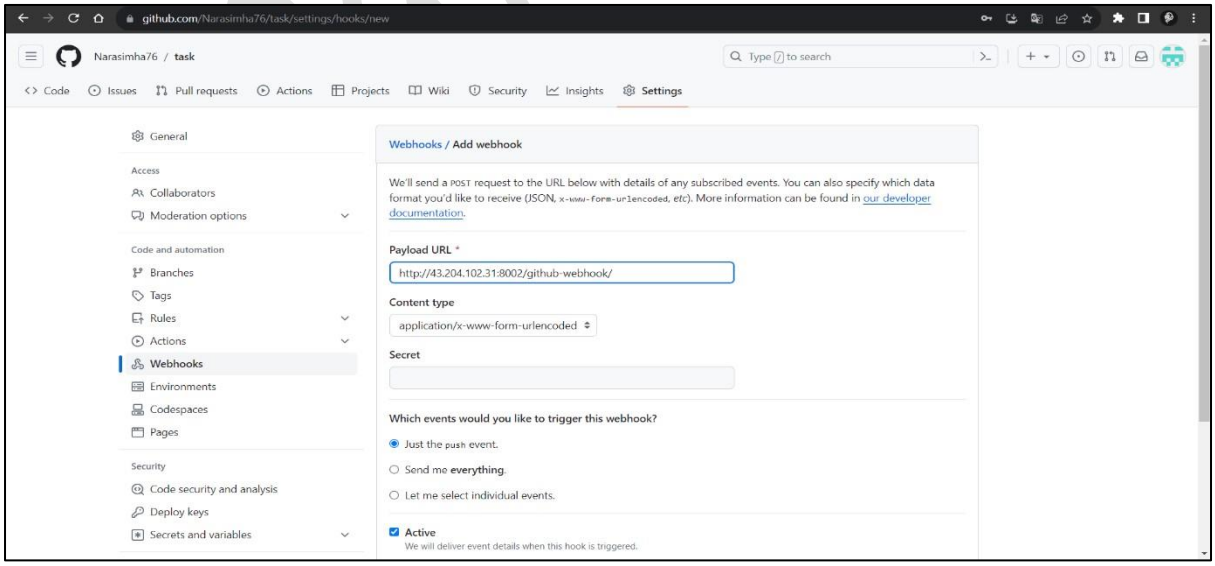




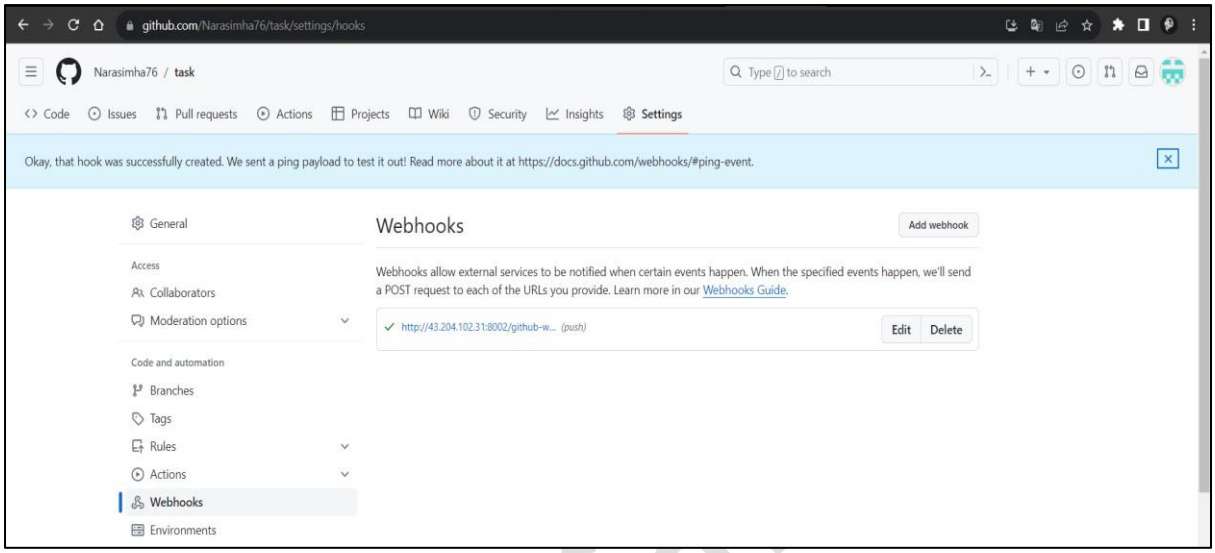
- Paste it in the pipeline script definition and click on save.

Creating a webhook:

- Go to the GitHub repository where you want to set up the webhook.
- Click on the "Settings" tab of your repository. In the left sidebar, click on "Webhooks & Services." Click the "Add webhook" button.



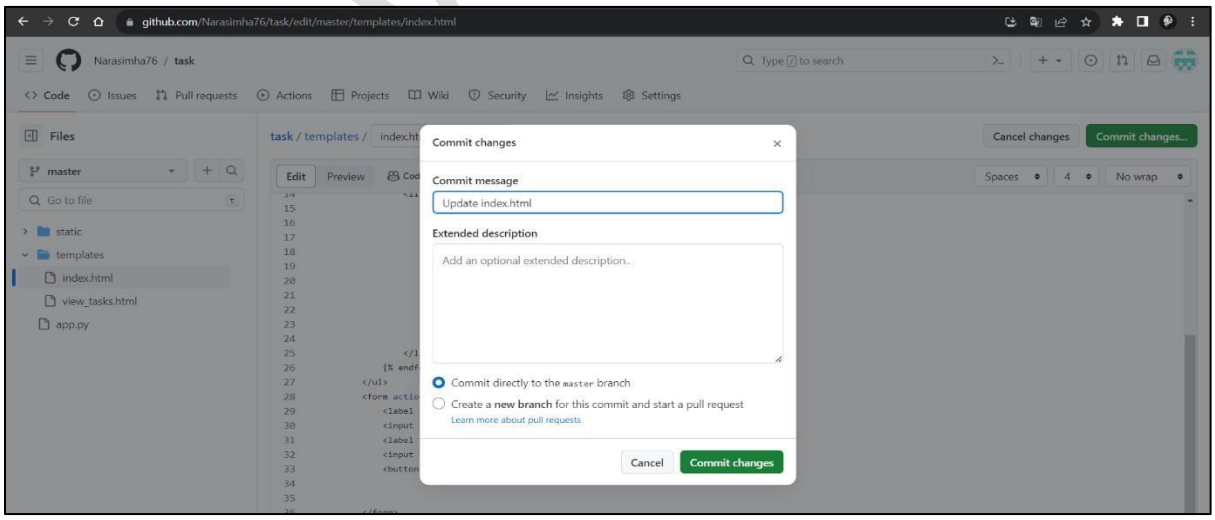
- In the "Payload URL" field, enter the Jenkins webhook URL. This should be the URL where Jenkins can receive GitHub push events.
  Format: http://jenkins-server/github-webhook/
- Replace **jenkins-server** with the actual address of your Jenkins server.
- Choose the events that should trigger the webhook. For CI/CD, you typically want to select at least "Just the push event."
- Click the "Add webhook" button to save the webhook.
- Ensure the webhook is set to be active



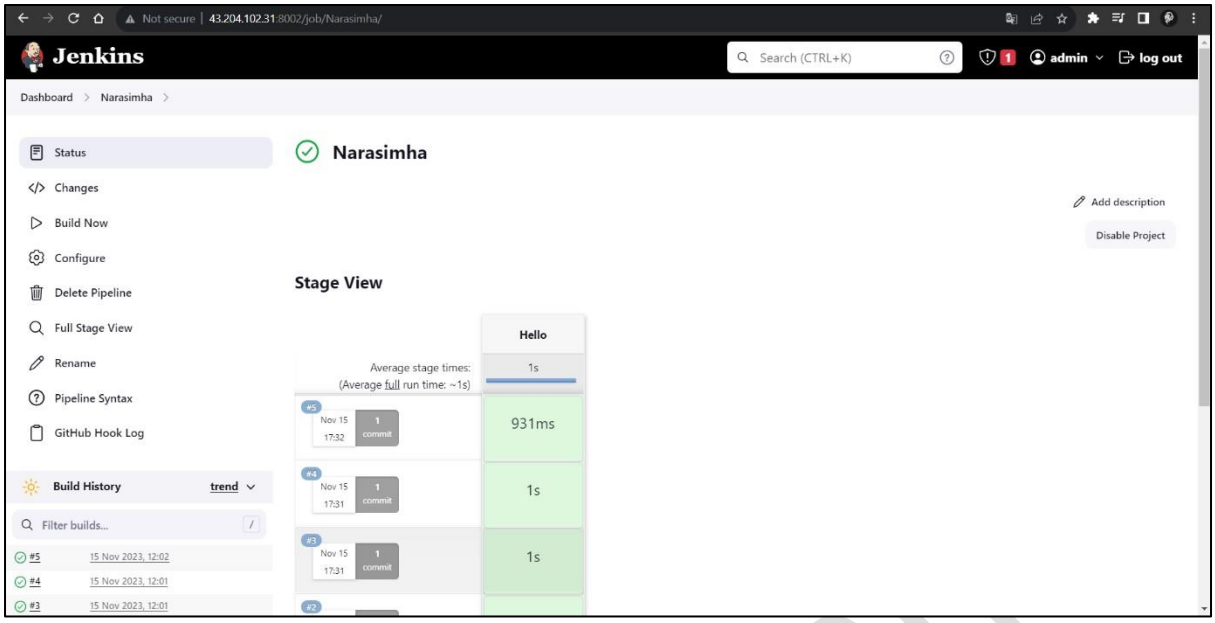
- Ensure the webhook is set to be active.

Test the setup:

- Make a small change (e.g., push a new commit) to your GitHub repository.



- Go to your Jenkins dashboard.
- You should see your job triggered by the GitHub webhook.



Set up build jobs for different branches.

Setting Up Jenkins Build Jobs for Feature Branches

Prerequisites

Jenkins installed and configured.

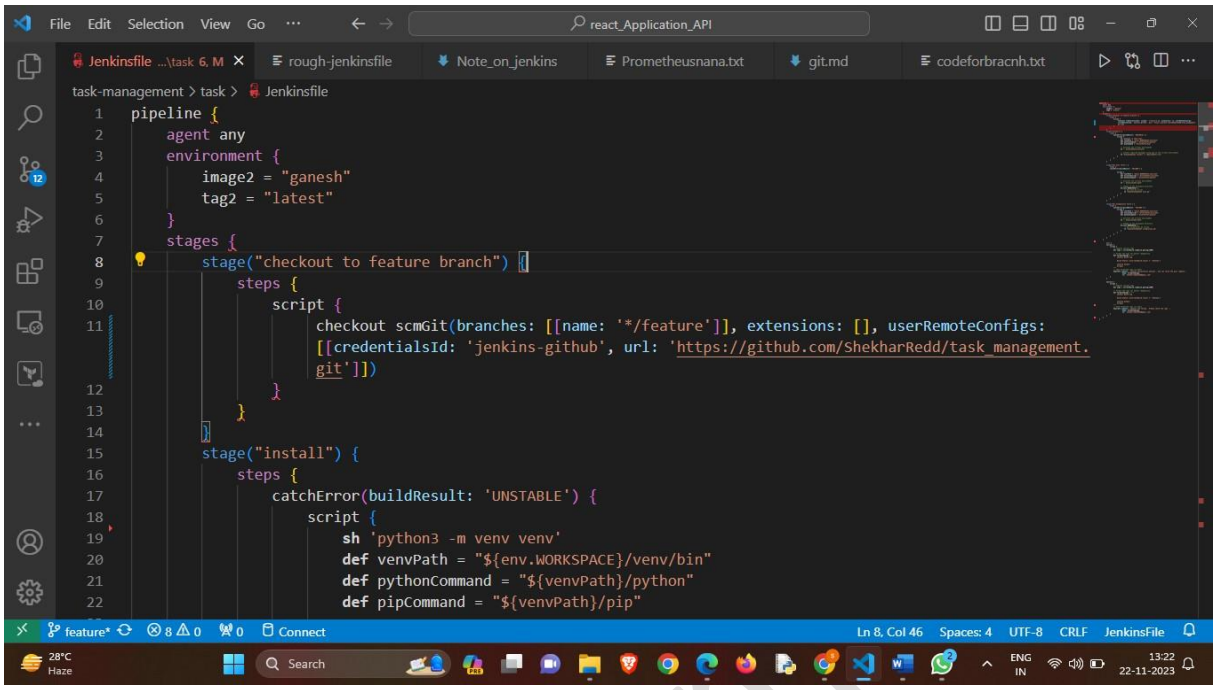
Jenkins plugins: GitHub, Email Extension

Python and required dependencies installed on the Jenkins agent.

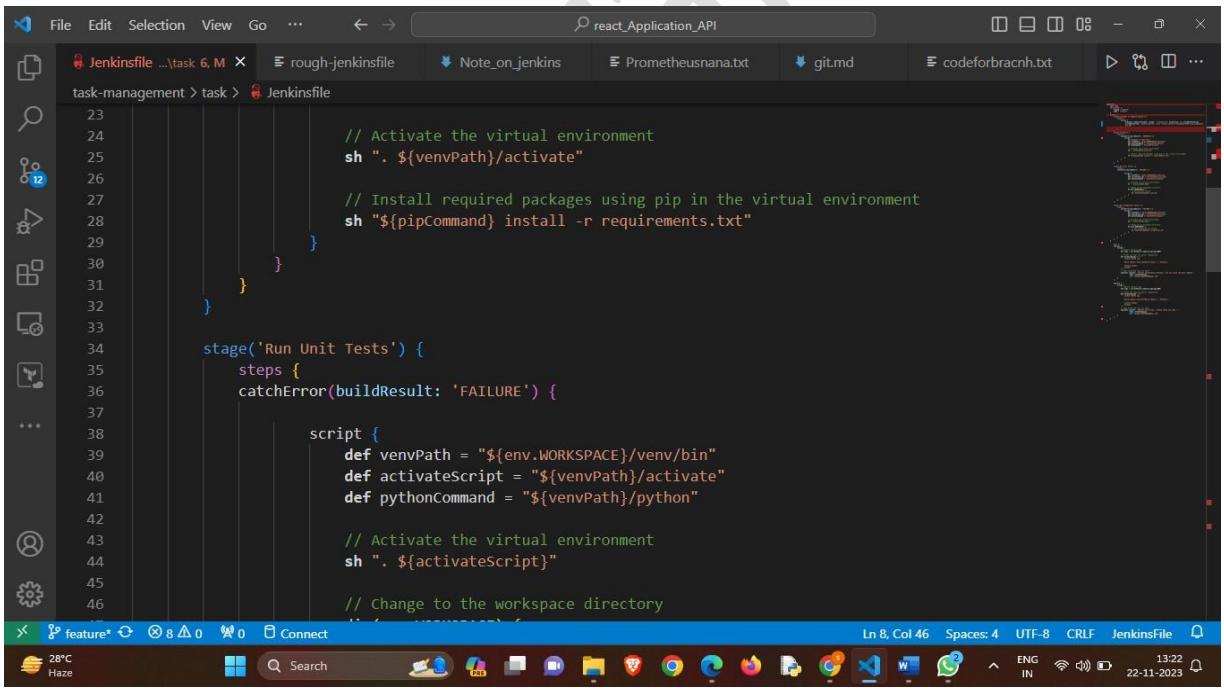
- **Configure Feature Branch Job**

1. Open Jenkins and navigate to the Jenkins dashboard.
2. Click on "New Item" to create a new Jenkins job.
3. Enter a name for the job (e.g., " feature_branch").
4. Select the "Pipeline" project type.
5. In Build Trigger section, select the “GitHub hook trigger for GITScm polling” option.
6. In the Pipeline section, choose "Pipeline script from SCM" as the Definition.
7. Choose "Git" as the SCM.
8. Set the Repository URL to your GitHub repository (e.g., 'https://github.com/ShekharRedd/task_management.git').
9. Under "Branches to build," add the branch specifier '*/feature' to focus on feature branches.
10. In the Script Path section, Select the Jenkinsfile which is in the git repo.
11. Save the job configuration.

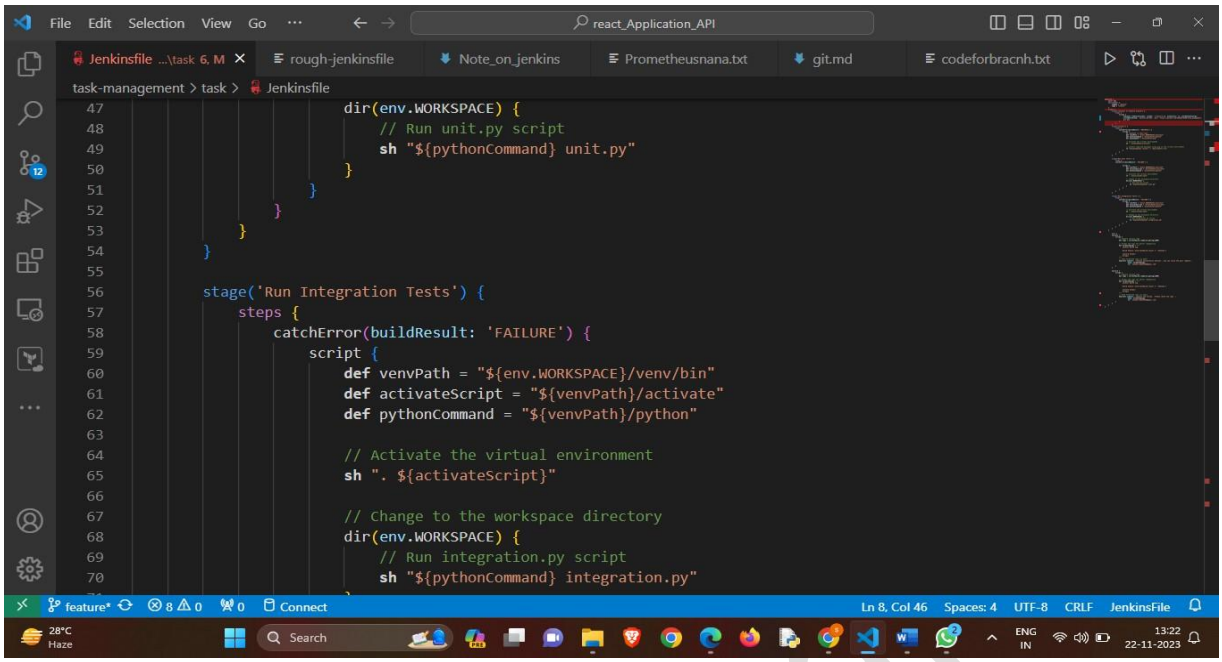
- **Jenkins Pipeline Script**



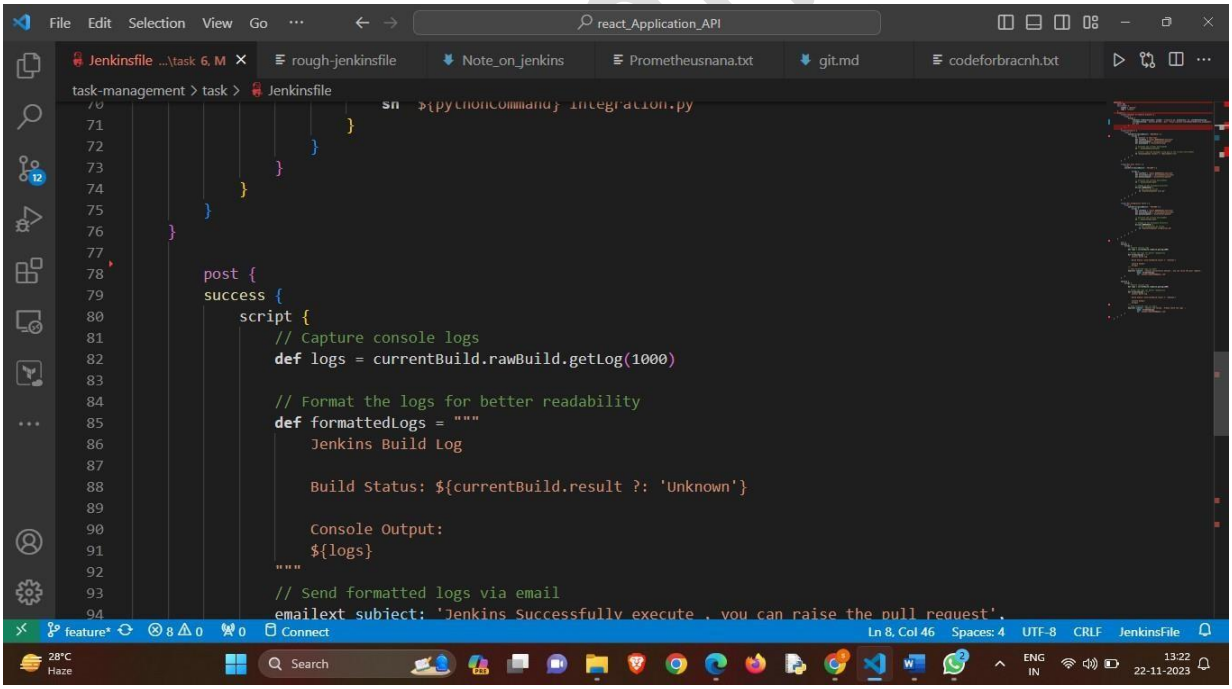
```
pipeline {
    agent any
    environment {
        image2 = "ganesh"
        tag2 = "latest"
    }
    stages {
        stage("checkout to feature branch") {
            steps {
                script {
                    checkout scmGit(branches: [[name: '*/feature']], extensions: [], userRemoteConfigs:
                    [[credentialsId: 'jenkins-github', url: 'https://github.com/ShekharRedd/task_management.
                    git']])
                }
            }
        }
        stage("install") {
            steps {
                catchError(buildResult: 'UNSTABLE') {
                    script {
                        sh 'python3 -m venv venv'
                        def venvPath = "${env.WORKSPACE}/venv/bin"
                        def pythonCommand = "${venvPath}/python"
                        def pipCommand = "${venvPath}/pip"
```



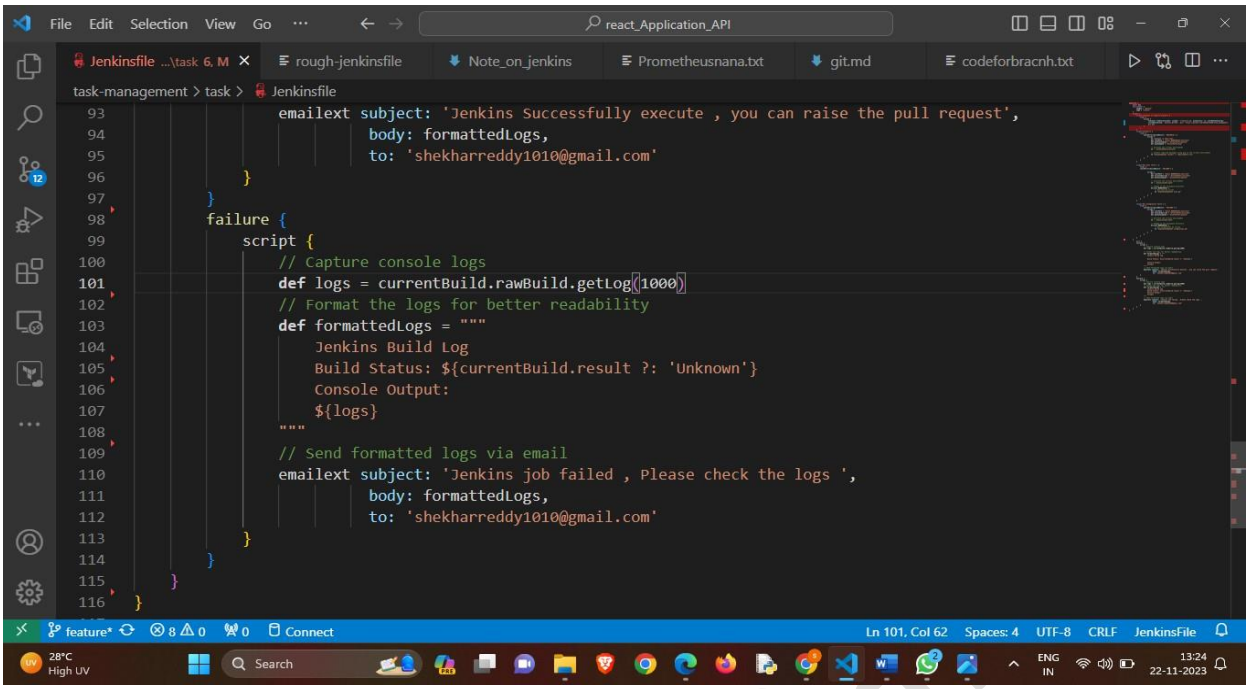
```
                        // Activate the virtual environment
                        sh ". ${venvPath}/activate"

                        // Install required packages using pip in the virtual environment
                        sh "${pipCommand} install -r requirements.txt"
                    }
                }
            }
        }

        stage('Run Unit Tests') {
            steps {
            catchError(buildResult: 'FAILURE') {

                    script {
                        def venvPath = "${env.WORKSPACE}/venv/bin"
                        def activateScript = "${venvPath}/activate"
                        def pythonCommand = "${venvPath}/python"

                        // Activate the virtual environment
                        sh ". ${activateScript}"

                        // Change to the workspace directory
```



```
                        dir(env.WORKSPACE) {
                            // Run unit.py script
                            sh "${pythonCommand} unit.py"
                        }
                    }
                }
            }
        }

        stage('Run Integration Tests') {
            steps {
                catchError(buildResult: 'FAILURE') {
                    script {
                        def venvPath = "${env.WORKSPACE}/venv/bin"
                        def activateScript = "${venvPath}/activate"
                        def pythonCommand = "${venvPath}/python"

                        // Activate the virtual environment
                        sh ". ${activateScript}"

                        // Change to the workspace directory
                        dir(env.WORKSPACE) {
                            // Run integration.py script
                            sh "${pythonCommand} integration.py"
```

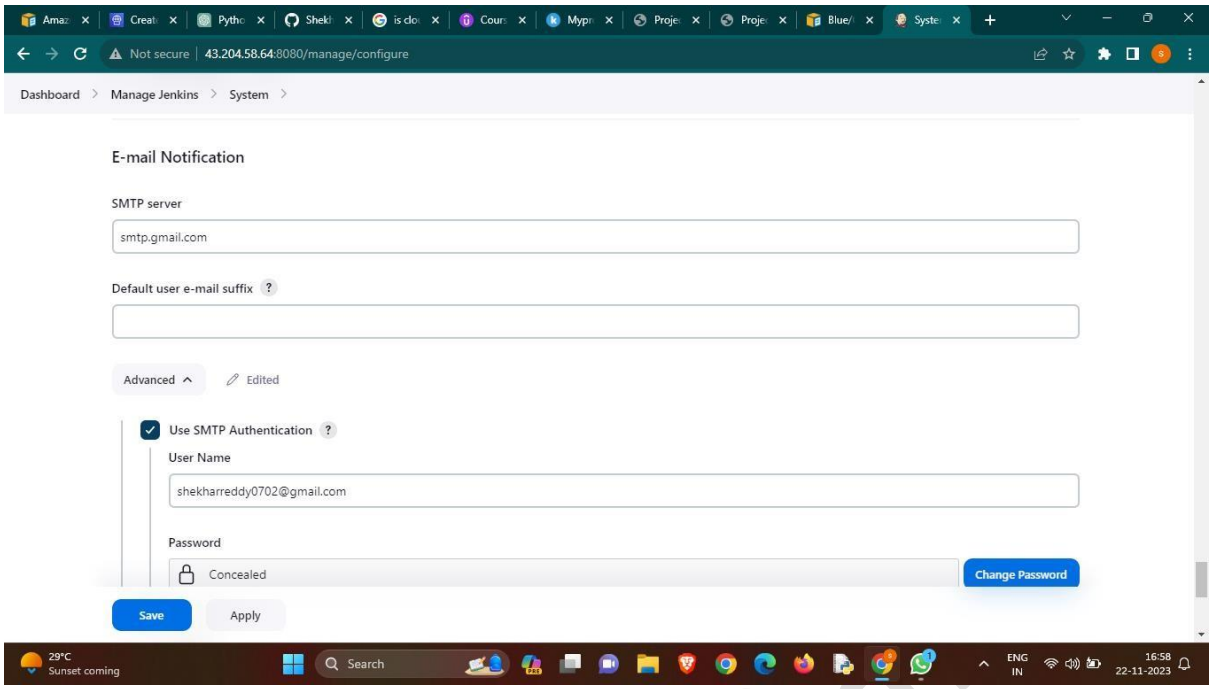


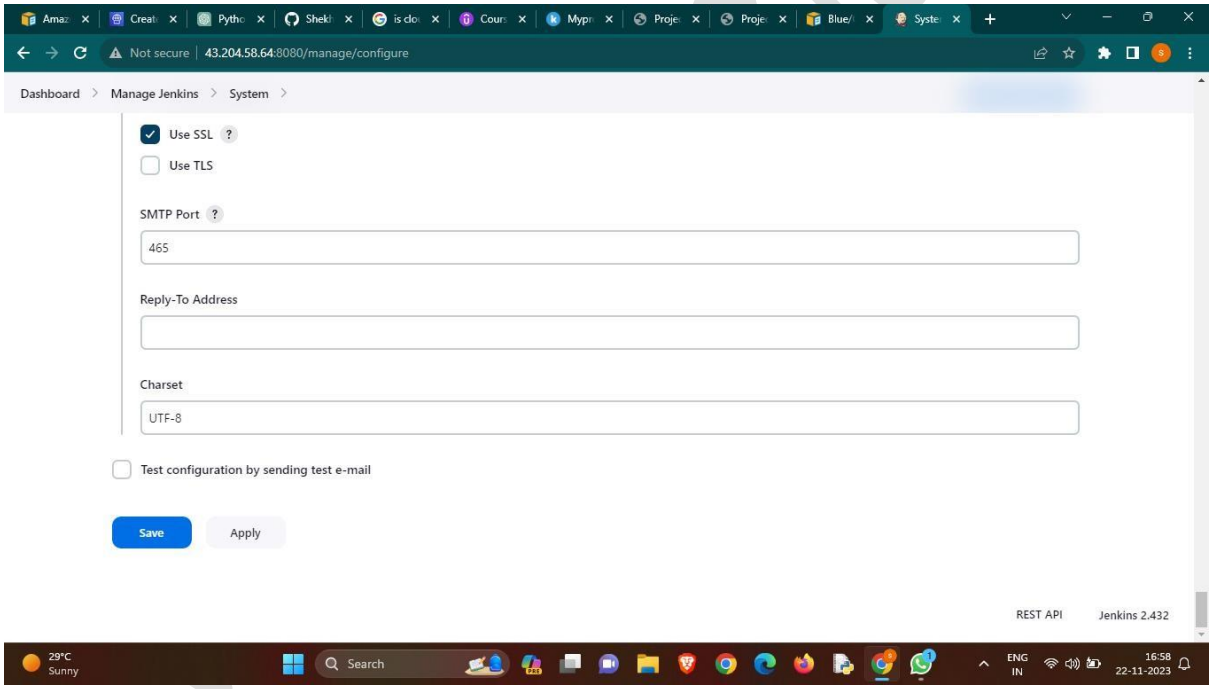
```
                        sh "${pythonCommand} integration.py"
                    }
                }
            }
        }
    }

    post {
        success {
            script {
                // Capture console logs
                def logs = currentBuild.rawBuild.getLog(1000)

                // Format the logs for better readability
                def formattedLogs = """
                    Jenkins Build Log

                    Build Status: ${currentBuild.result ?: 'Unknown'}

                    Console Output:
                    ${logs}
                """
                // Send formatted logs via email
                emailext subject: 'Jenkins Successfully execute , you can raise the pull request',
```

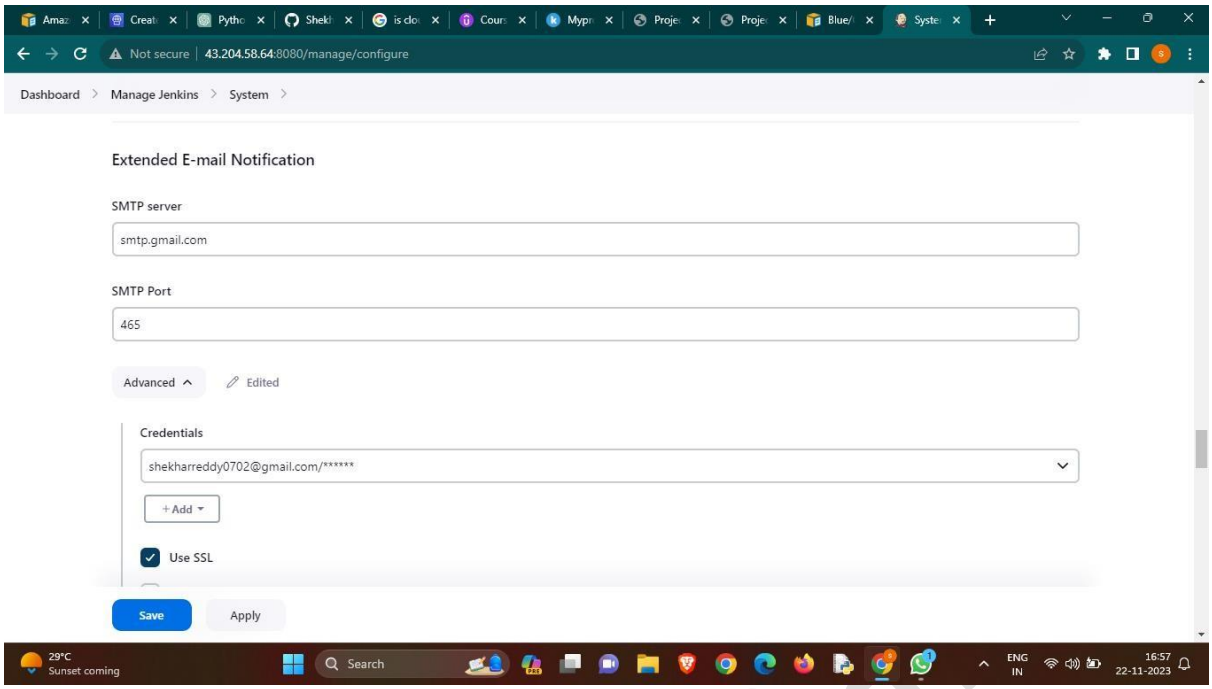


EMAIL Configuration

- Install E-mail Notification and Extended E-mail Notification plugins E-mail Notification: This plugin Mailer, allows users to send basic emails.
- Extended E-mail Notification: This plugin Email Extension Plugin, allows users to customize when emails are sent, who receives them, and the content of the emails.
- Goto Dashboard >> Manage Jenkins >> System
- Configure the settings as shows in below pictures.

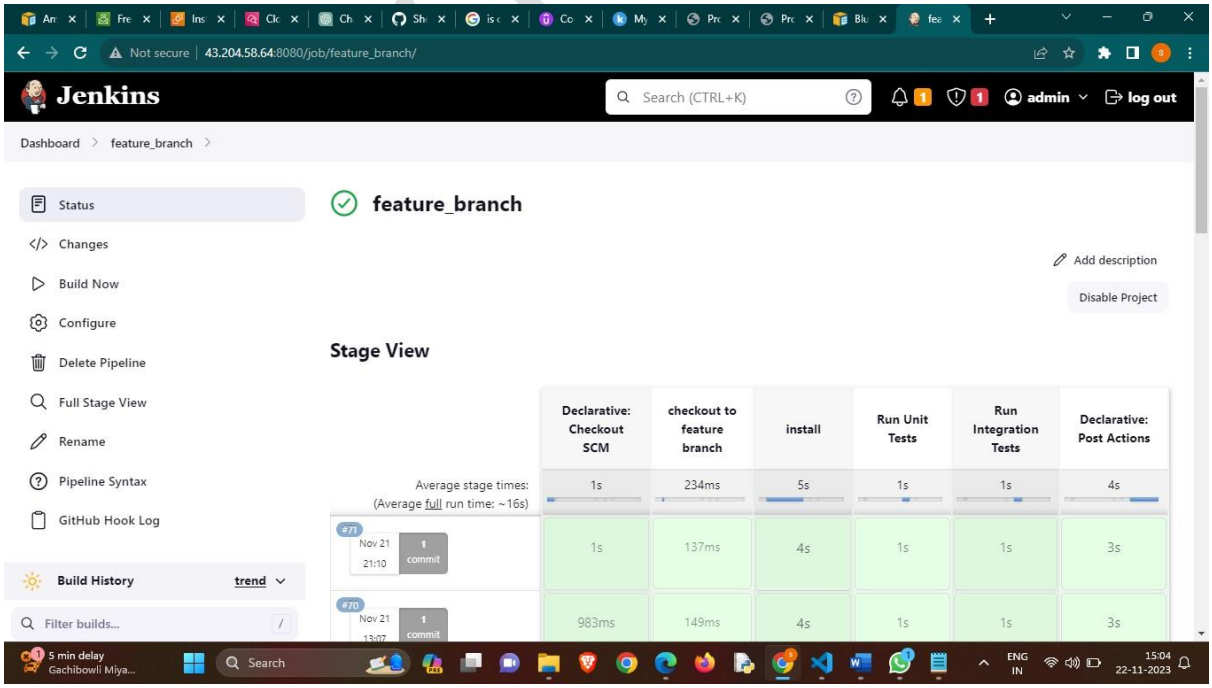




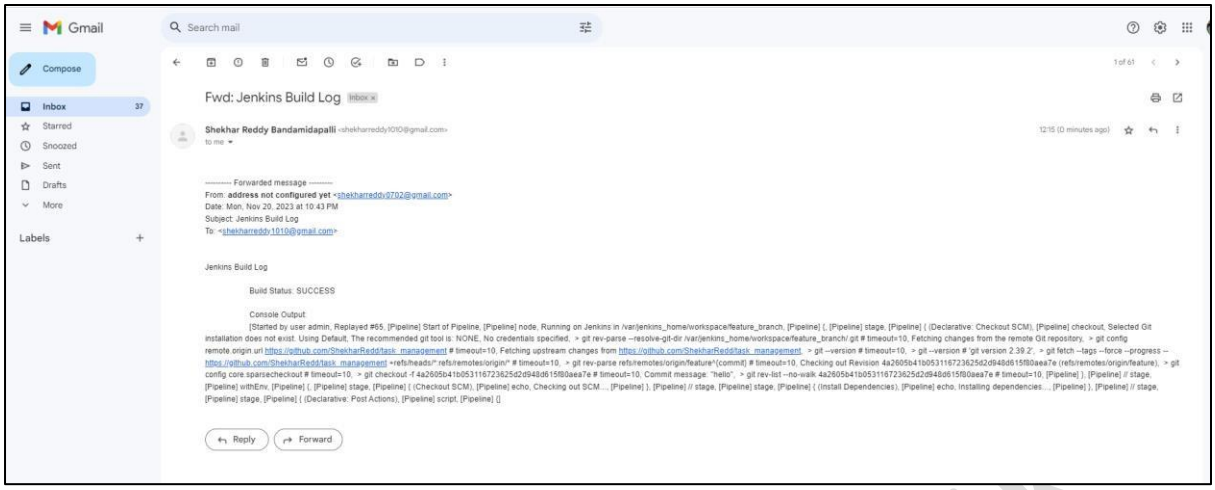


Run the Feature Branch Job

- Trigger a build manually or configure webhooks for automatic triggering on code changes.
- Monitor the build progress on the Jenkins dashboard.
- Review the console output and logs for each stage.
- Receive email notifications on build success or failure.



- Jenkins sends an email notification when a build completes, whether it's a successful build, unstable build, or a failed build.



UNIT TEST :

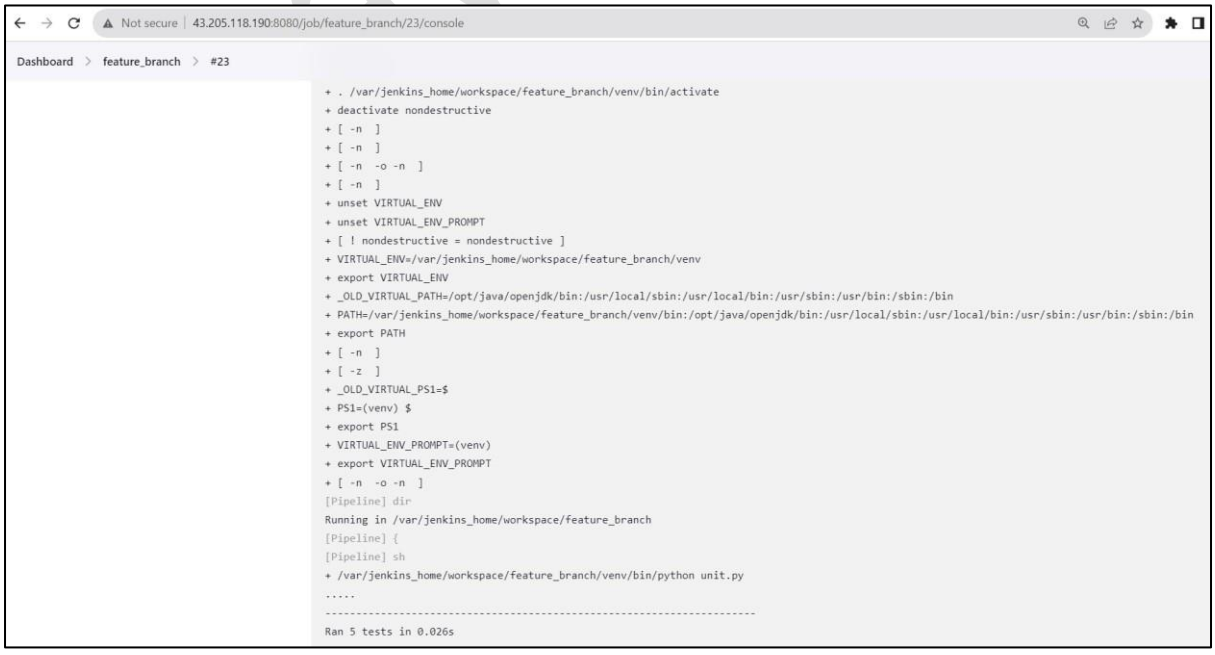
- Unit testing in Python is a crucial step in making sure each part of your code works correctly. It involves checking the smallest units, or components, of your program to ensure they function as expected.
- Python's built-in "unittest" module helps you create and run these tests. This process is essential for building reliable and error-free software

Usecase:

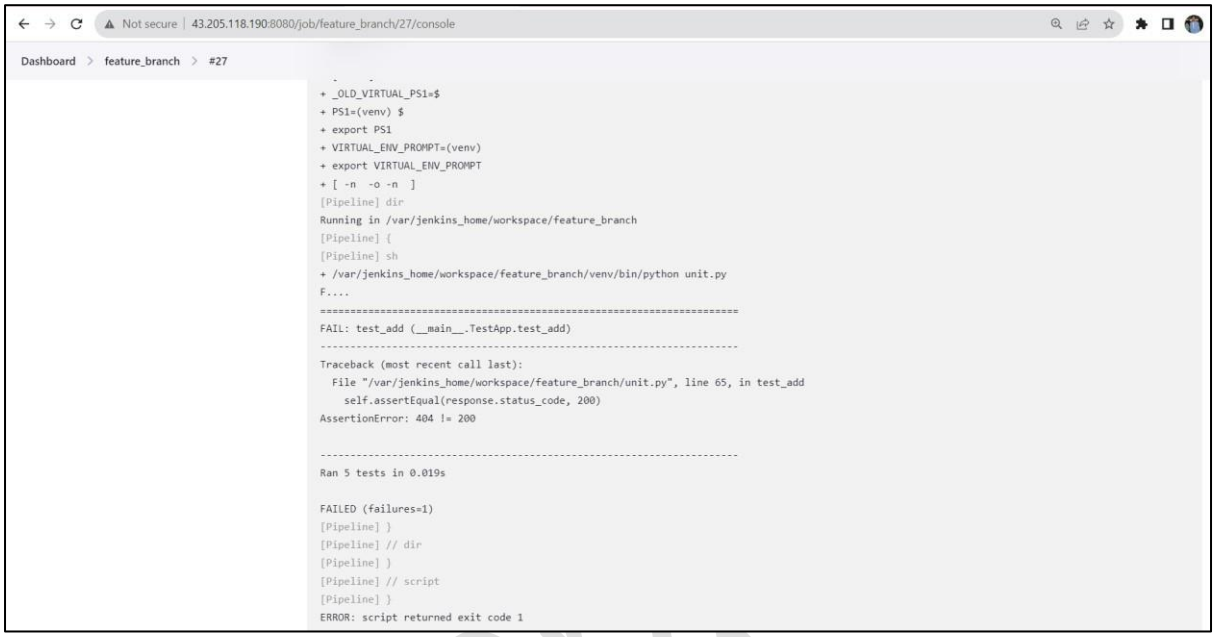
- Unit testing ensures that each isolated part (unit) of a software application functions correctly in isolation, catching bugs early in the development process.

```python
#     unittest.main()

from datetime import datetime
import unittest
from app import app, tasks

class TestApp(unittest.TestCase):

    def setUp(self):
        self.app = app.test_client()
        self.app.testing = True
        tasks.clear()  # Clear tasks before each test

    def test_index(self):
        response = self.app.get('/')
        self.assertEqual(response.status_code, 200)

    def test_add(self):
        task_data = {'task': 'Test Task', 'due_date': '2023-11-30'}
        response = self.app.post('/add', data=task_data, follow_redirects=True)

        self.assertEqual(response.status_code, 200)
        self.assertIn({'task': 'Test Task', 'due_date': datetime(2023, 11, 30)}, tasks)

    def test_delete_valid_id(self):
        tasks.append({'task': 'Test Task', 'due_date': datetime.now()})
        response = self.app.get('/delete/0')
        self.assertEqual(response.status_code, 302)
        self.assertEqual(len(tasks), 0)

    def test_delete_invalid_id(self):
        response = self.app.get('/delete/0')
        self.assertEqual(response.status_code, 302)
        self.assertEqual(len(tasks), 0)

    def test_view_tasks(self):
        response = self.app.get('/view_tasks')
        self.assertEqual(response.status_code, 200)

if __name__ == '__main__':
    unittest.main()
```

- If all assertions within a test case are true, the test case is considered to have passed.
- The testing framework will typically output a message indicating that the test has passed, and the overall result of the test run is considered successful.



- If any assertion within a test case is false, the test case is considered to have failed.
- The testing framework will output information about which specific assertion failed, along with the expected and actual values.

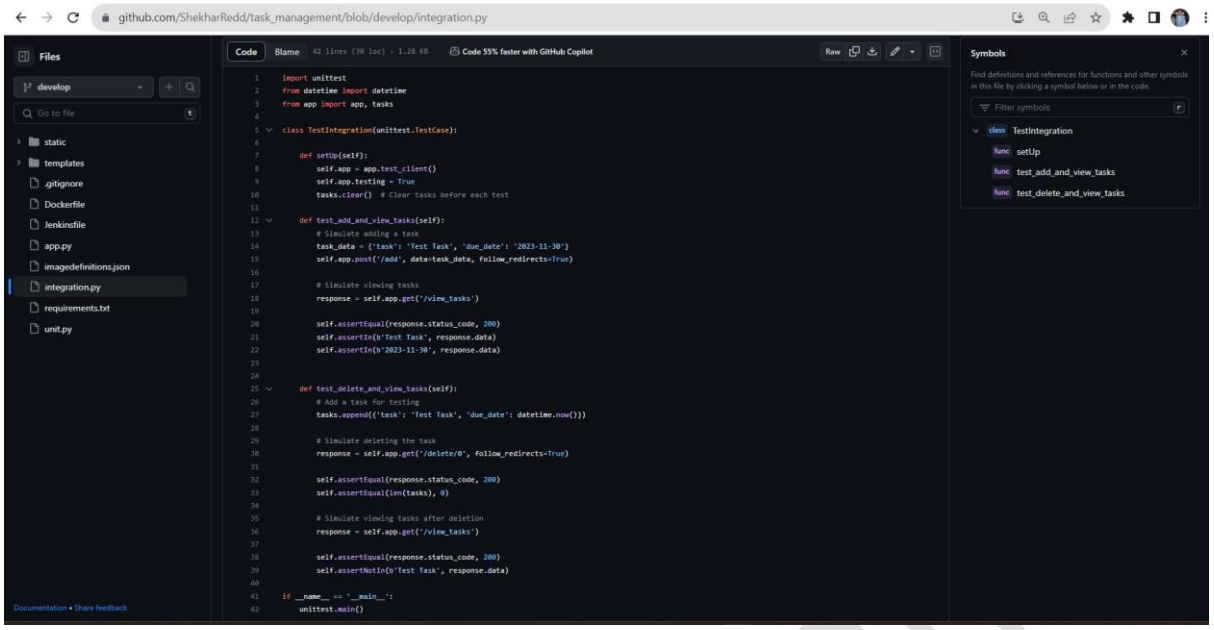


INTEGRATION TEST :

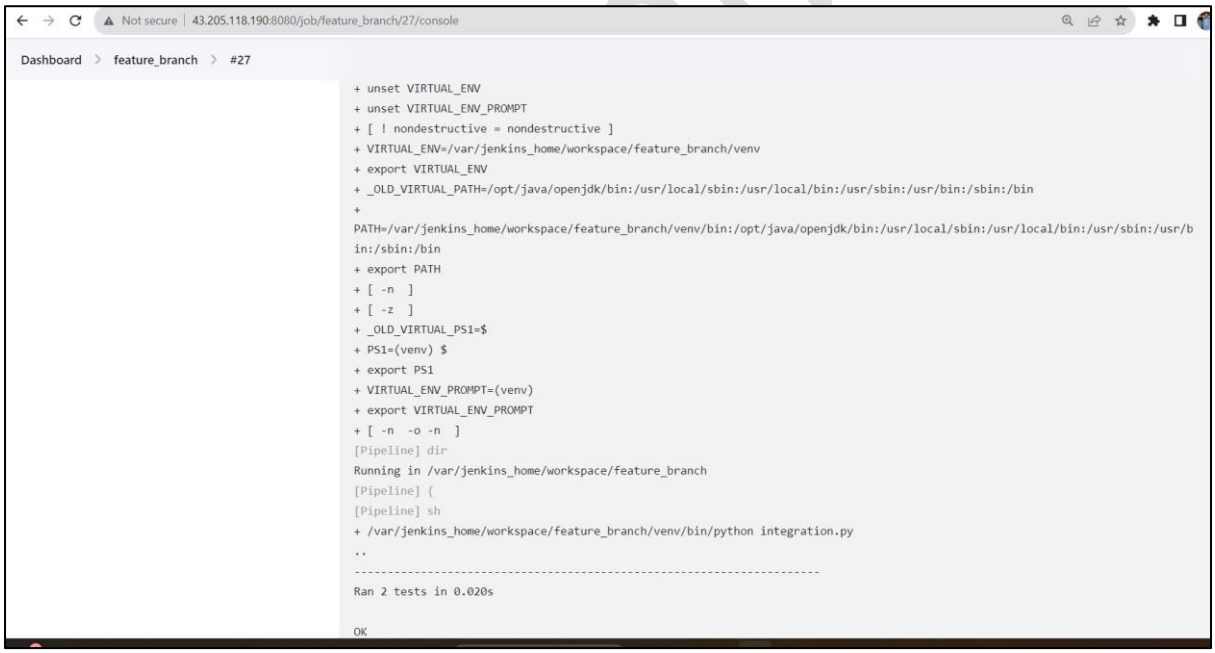
- Integration testing is a level of software testing where individual units or components of a system are combined and tested as a group to ensure that they work together seamlessly.

Usecase:

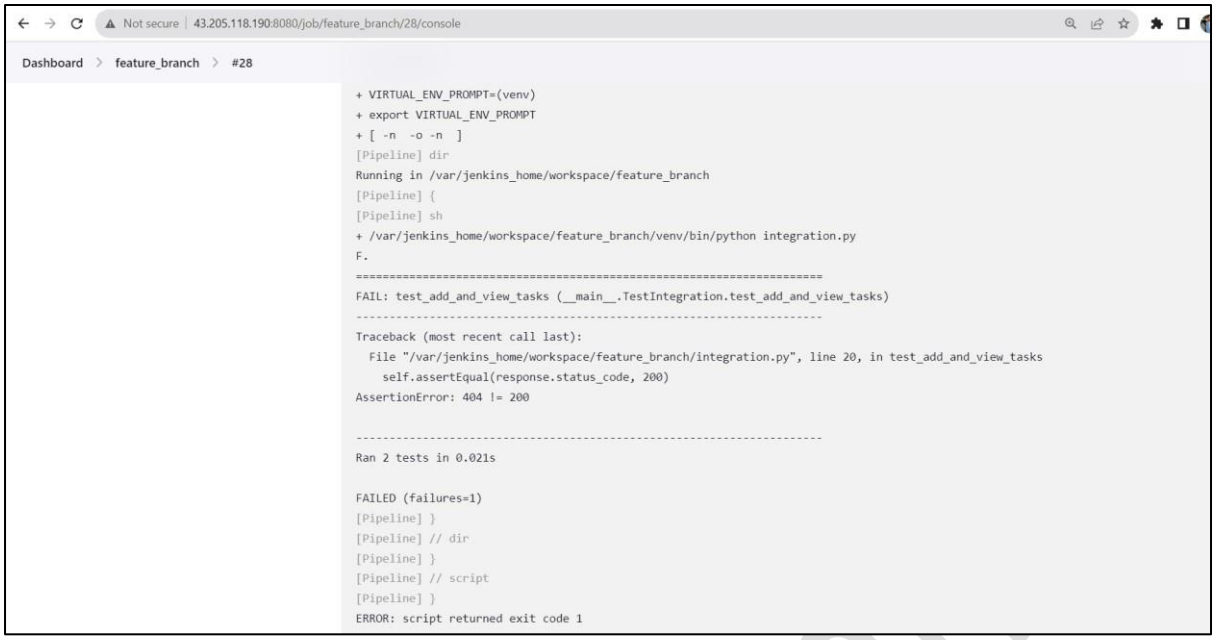
- Integration testing helps catch issues related to the combination of components early in the development process, reducing the risk of problems in the production environment.



- If all the test cases in your integration test suite pass, it indicates that the integrated components are working together as expected.
- The testing framework or test runner will typically output a message indicating that the integration tests have passed.



- If any test case fails during integration testing, it indicates that there is an issue with the interaction between components.
- The testing framework will output information about which specific test case failed, along with the expected and actual results.

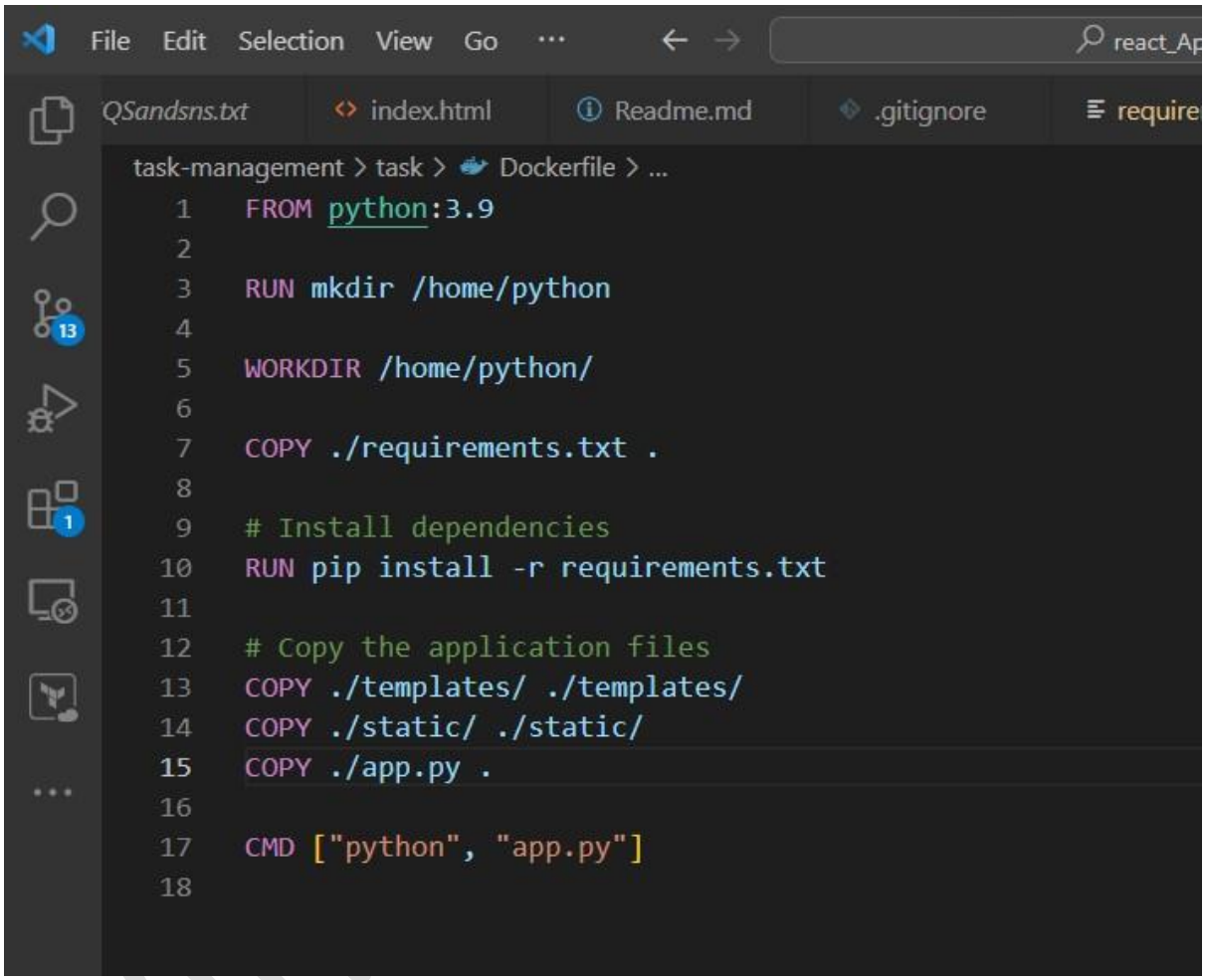


- The build status will be marked as "Success" or "Stable." This indicates that all the defined stages, including unit and integration tests, have completed successfully.

Sub Task 3:

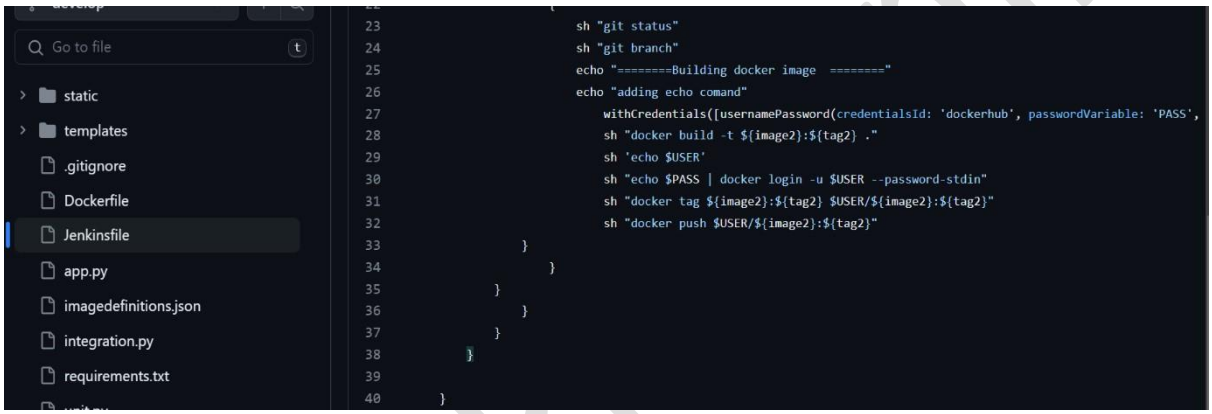
Dockerfile:

- Writing a Dockerfile for a web application involves specifying the necessary steps to create a Docker image that can run your web application. Here's a basic example of a Dockerfile for a web application using a Python-based web framework (Flask)

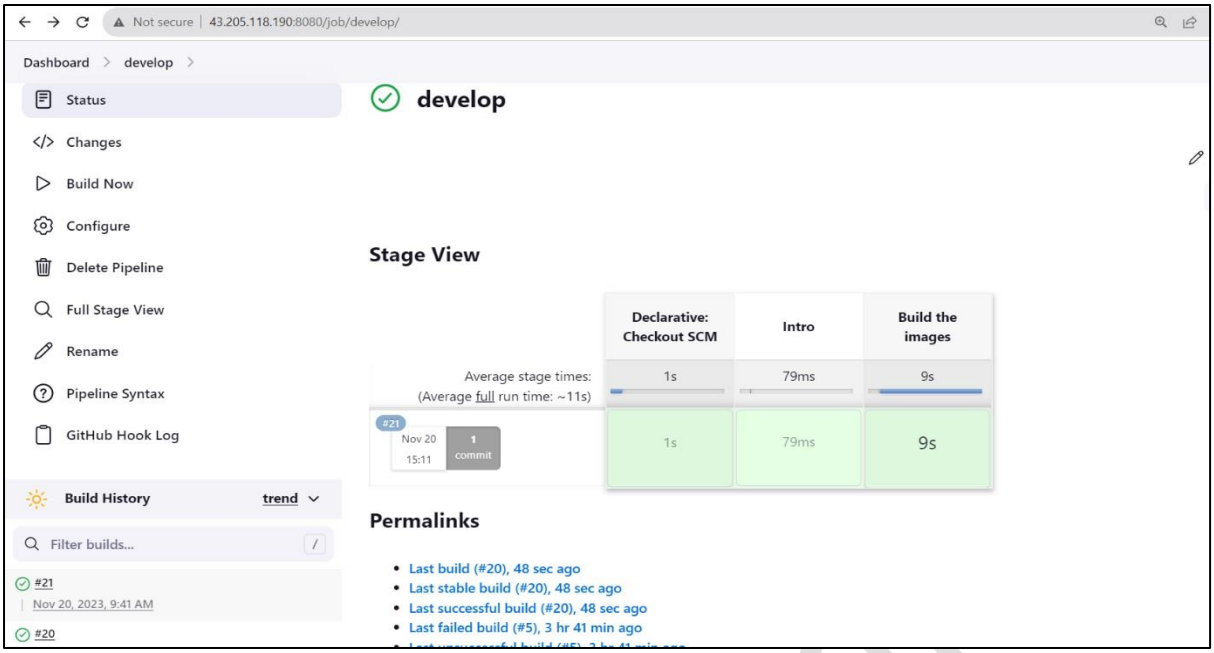


Build docker image and store in docker hub :

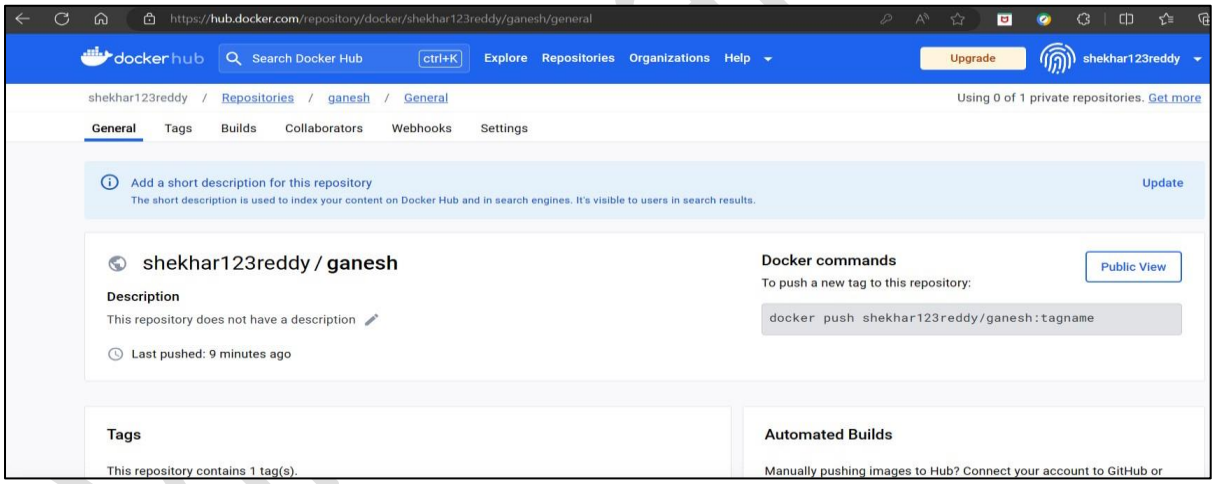
- Jenkinsfile that we can use to build Docker images for different components of a web application (frontend and backend) and push those images to Docker Hub.



- If all stages in the Jenkinsfile are executed successfully, and there are no errors, the pipeline will be considered a success.

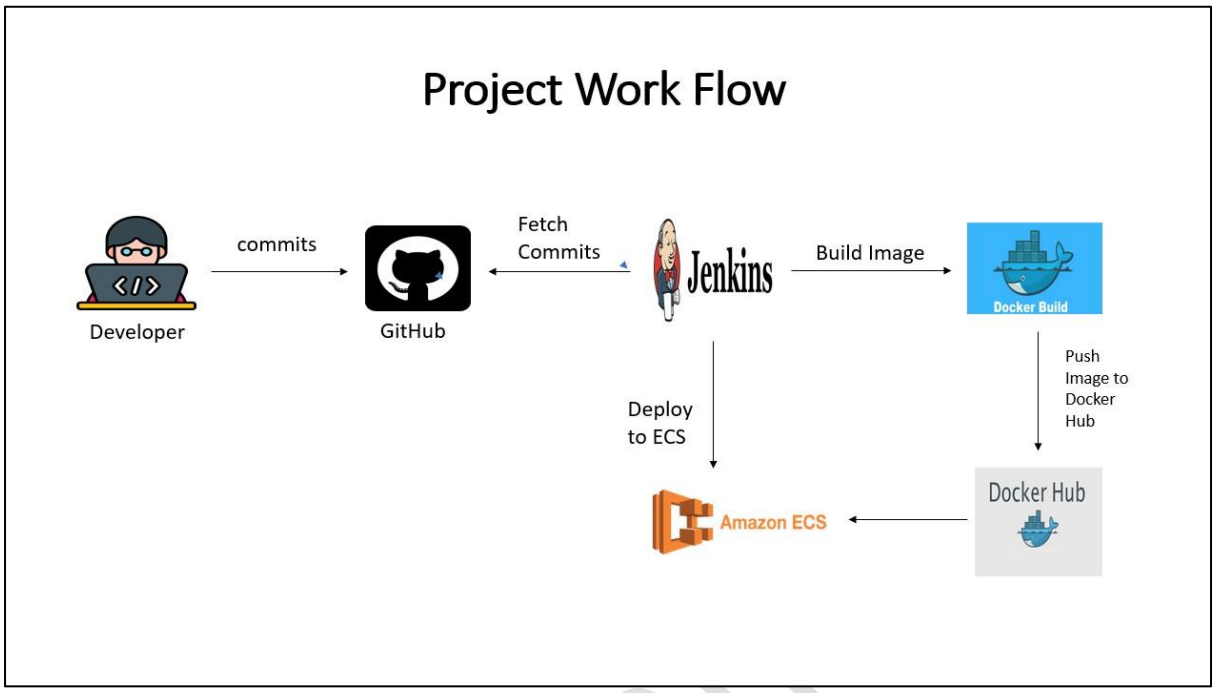


- If the Docker image is successfully built, the next step in a typical CI/CD pipeline is to push the Docker image to a container registry, such as Docker Hub.



Sub Task 4:

Setting up a Code Pipeline with source (Git), build (Jenkins), and deploy ECS stages.



Prerequisites:

2. **AWS Account:**

   - Ensure you have an AWS account with the necessary permissions to create and configure AWS resources.

3. **Git Repository:**

   - Have a Git repository containing your application code.

4. **Jenkins Server:**

   - Set up a Jenkins server that can be accessed by AWS Code Pipeline.

5. **Docker and ECS:**

   - Ensure you have Docker installed for building containers.
   - Here an ECS cluster, task definition, and service ready for deploying your application.

Procedure:

Step 1: Create IAM Role for Code Pipeline:

In AWS CodePipeline, an IAM (Identity and Access Management) role is used to define permissions for the pipeline to interact with AWS services and resources. This role specifies what actions the

pipeline can perform and what resources it can access during its execution. Here are the general steps to create an IAM role for AWS CodePipeline.

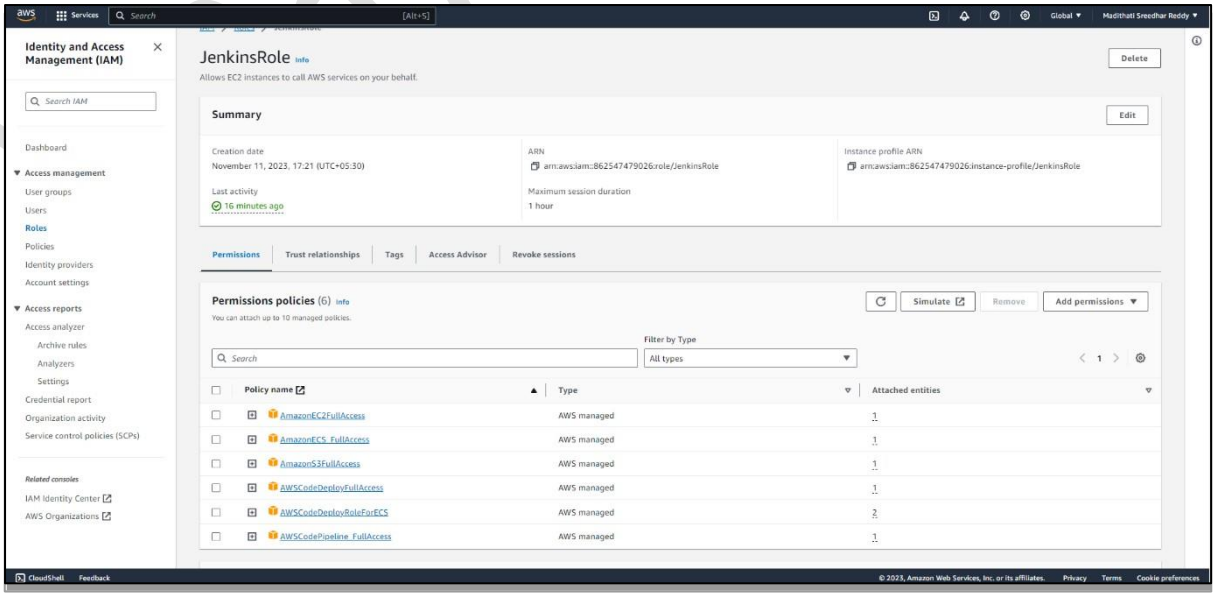
1. **Open the IAM Console:**
   - Log in to the AWS Management Console and navigate to the IAM service.

2. **Create a New Role:**
   - Click on "Roles" in the left navigation pane.
   - Click on the "Create role" button.

3. **Choose the Service that Will Use the Role:**
   - In the "Select type of trusted entity" section, choose "AWS service" as the type of trusted entity.
   - In the "Choose a use case" section, find and select "CodePipeline."

4. **Attach Permissions Policies:**
   - In the "Attach permissions policies" step, you can either choose existing policies or attach policies later. Policies define what actions the role can perform. At a minimum, the role should have permissions to access the artifacts stored in Amazon S3 and perform actions on the CodePipeline itself.
   - Common policies to attach include `AWSCodePipelineFullAccess` and `AmazonS3ReadOnlyAccess`.

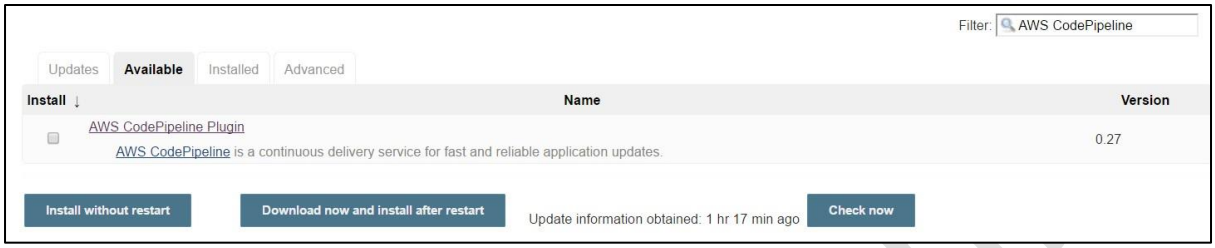
5. **Review:**
   - Provide a meaningful name for the role and optionally add a description. Review your choices and click "Create role."



Step 2: Configure Jenkins for Code Pipeline Integration:

1. Install the Jenkins AWS CodePipeline plugin

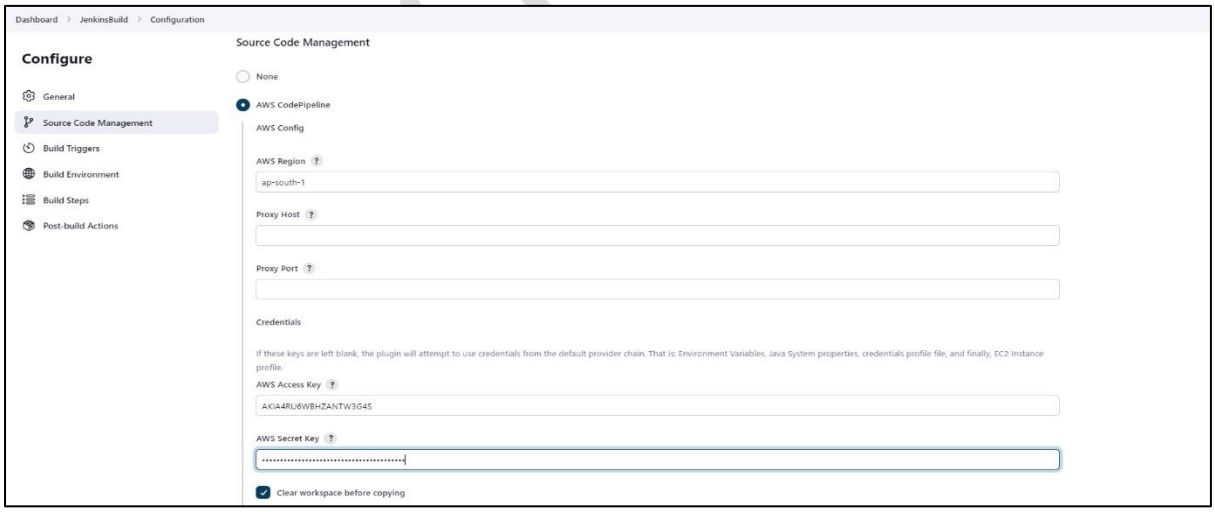
- Click **Manage Jenkins**.
- Select **Manage Plugins**.
- Choose the **Available** tab, then search for the **AWS CodePipeline** plugin in the **Filter** search box.



- Select the plugin, then click **install without restart**

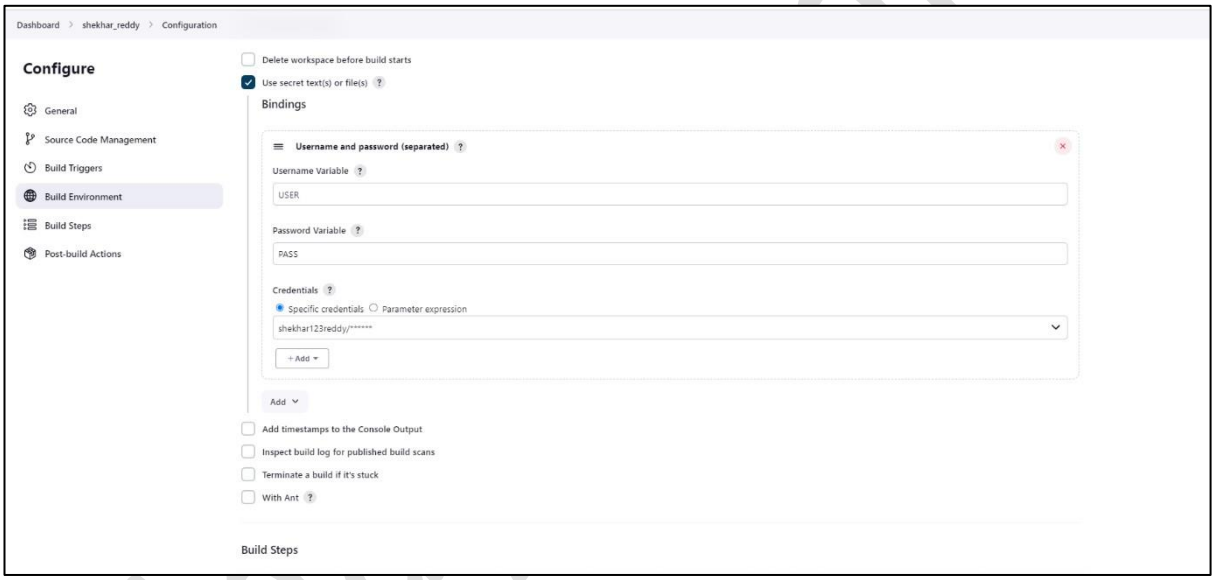
2. Create the "JenkinsBuild" Jenkins project:

- Click **New Item.**
- Enter an Item Name: **"JenkinsBuild"**
- Choose **Freestyle Project**, then click **OK**
- In **Source Code Management** section, select the AWS CodePipeline, then configure the values as shown below:

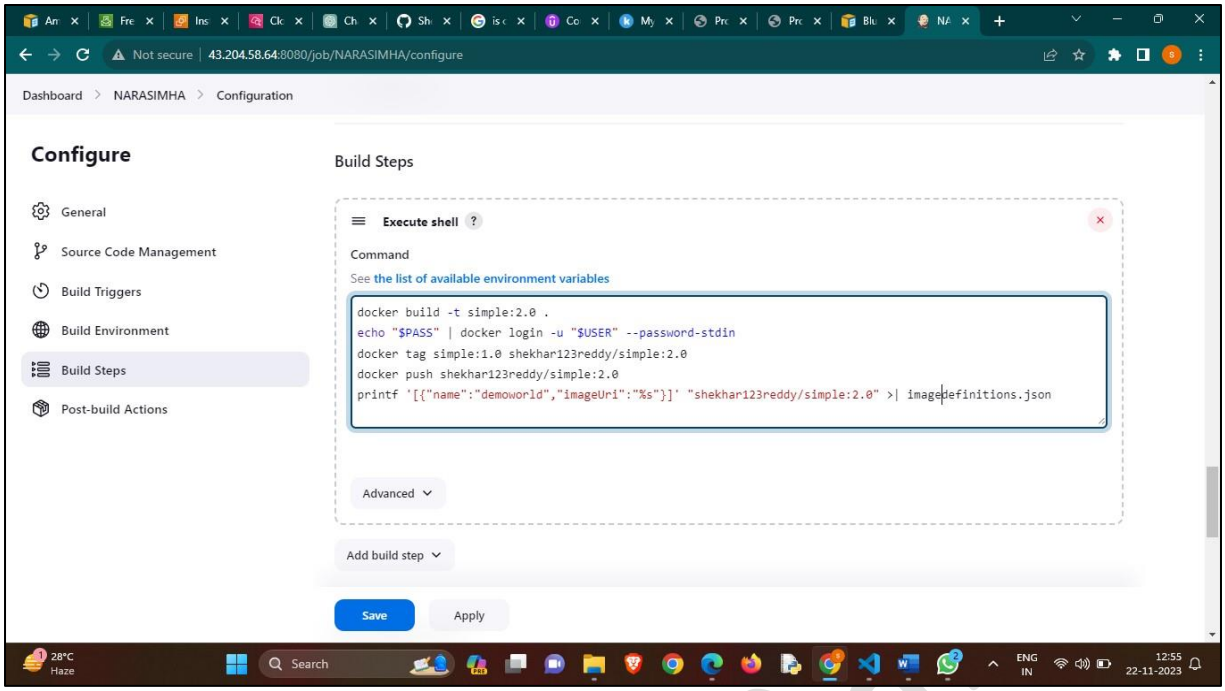


- In the **Build Triggers** section, select **Poll SCM** and configure the schedule to poll the SCM as often as you want, for example every minute.

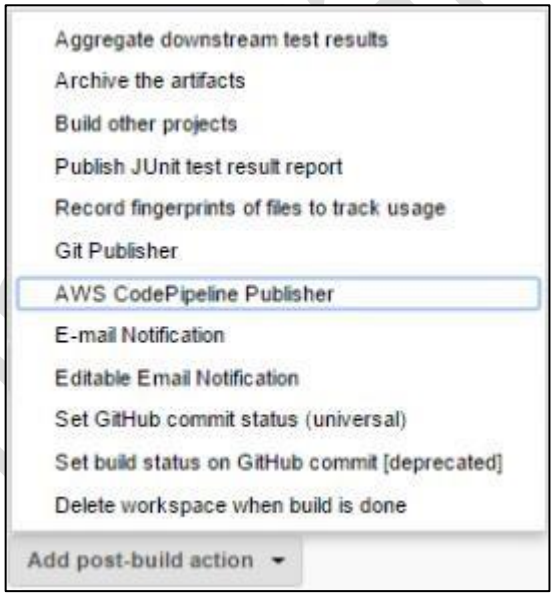
- In **Buil Environment section,** select the “Use secret text(s) or file(s)”



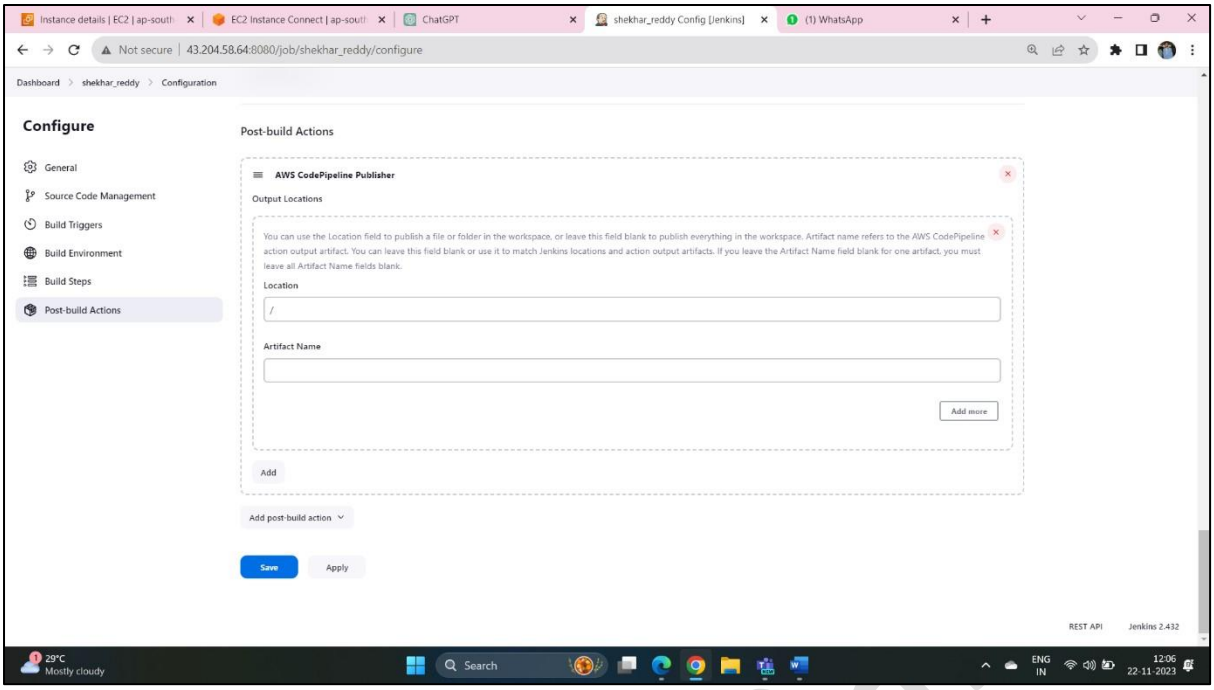
- In **Build steps** section, select add Execute shell commands.



- On the **Post-build Actions** tab, add a post-build action.
- Choose **AWS CodePipeline Publisher**.



- Do not add any output locations.



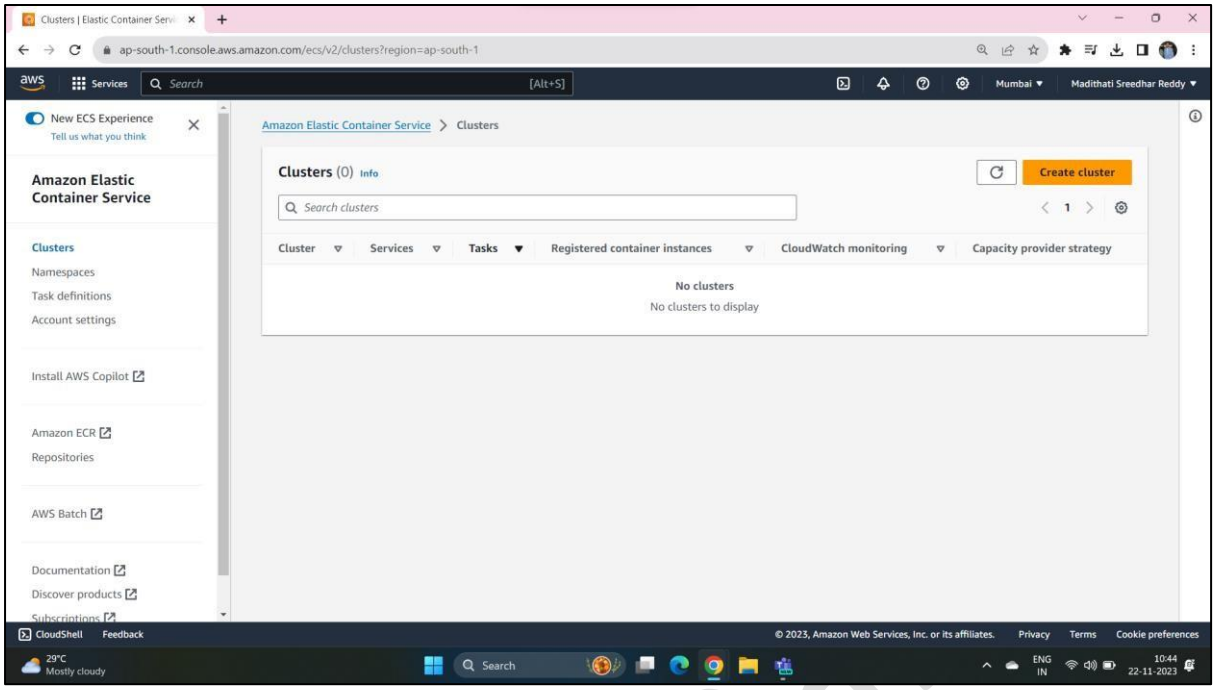
- Click **Save**.

Step-3: Creating an ECS Cluster, Task definition, Service in AWS:

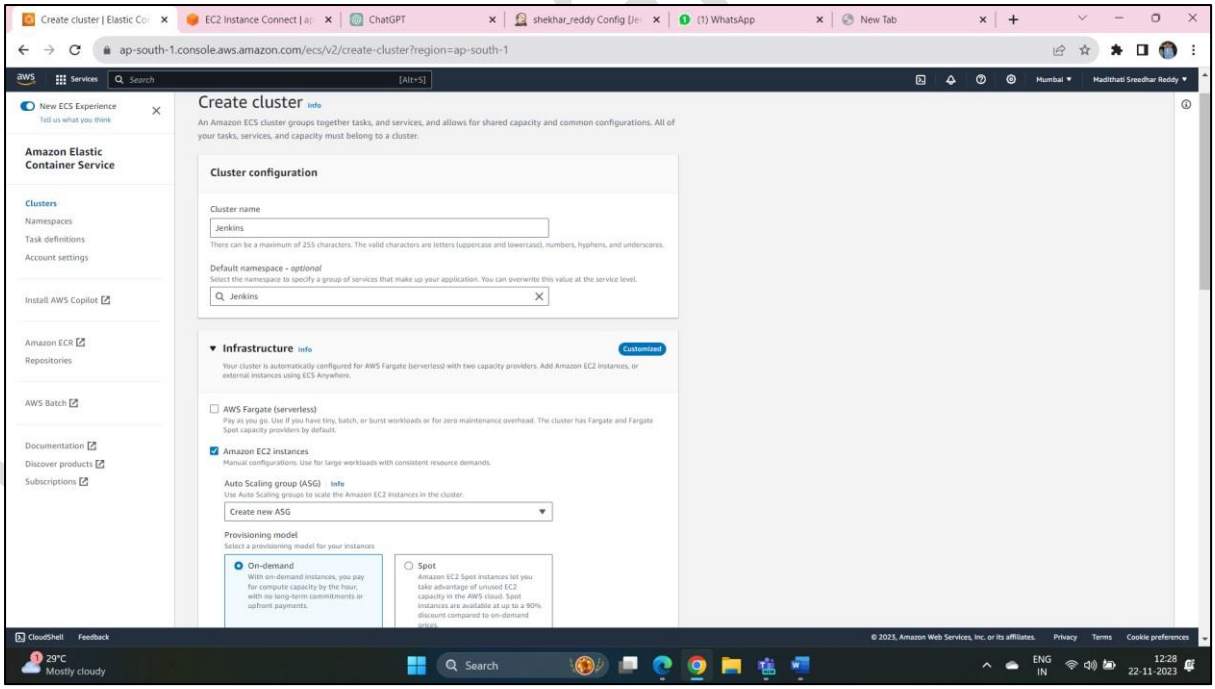
1.Creating ECS Cluster:

Creating an Amazon ECS (Elastic Container Service) cluster involves several steps. ECS is a fully managed container orchestration service that allows you to run, stop, and manage Docker containers on a cluster. Here are the general steps to create an ECS cluster:

- **Open the AWS Console >>>** Navigate to the Amazon ECS console.
- **Create a Cluster**
- In the ECS console, click on "Clusters" in the left navigation pane. Click the "Create Cluster" button.

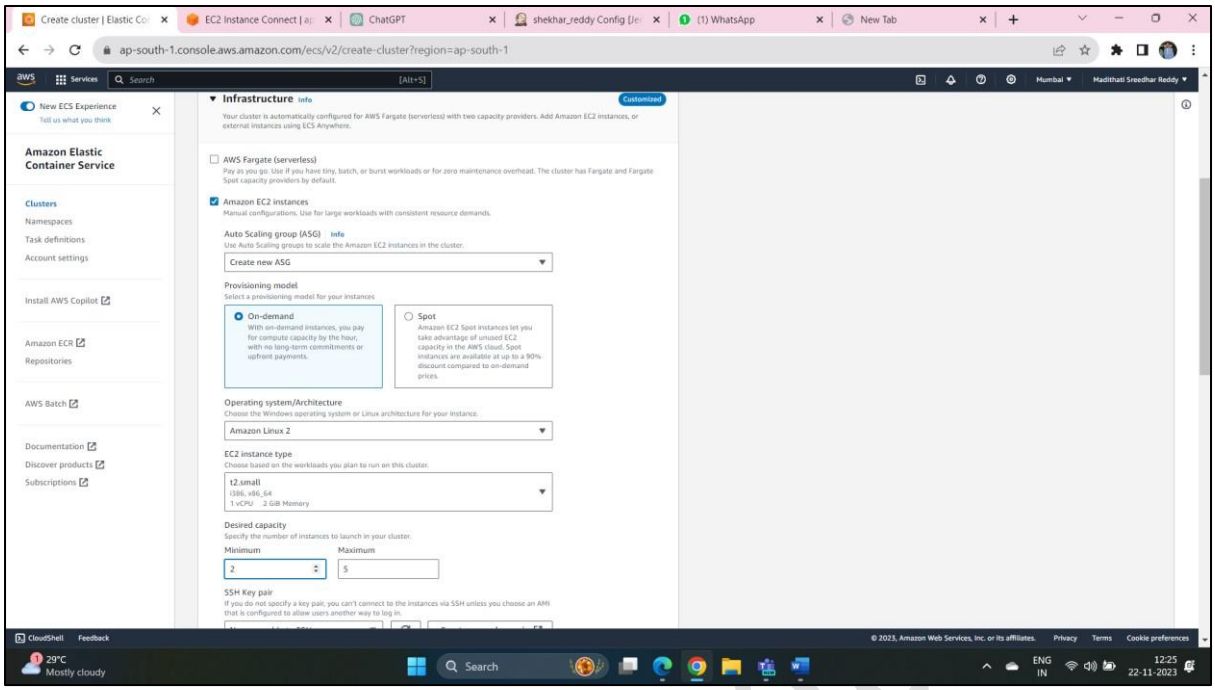


- Configure the cluster settings:
  Cluster Name: Give your cluster a unique name.

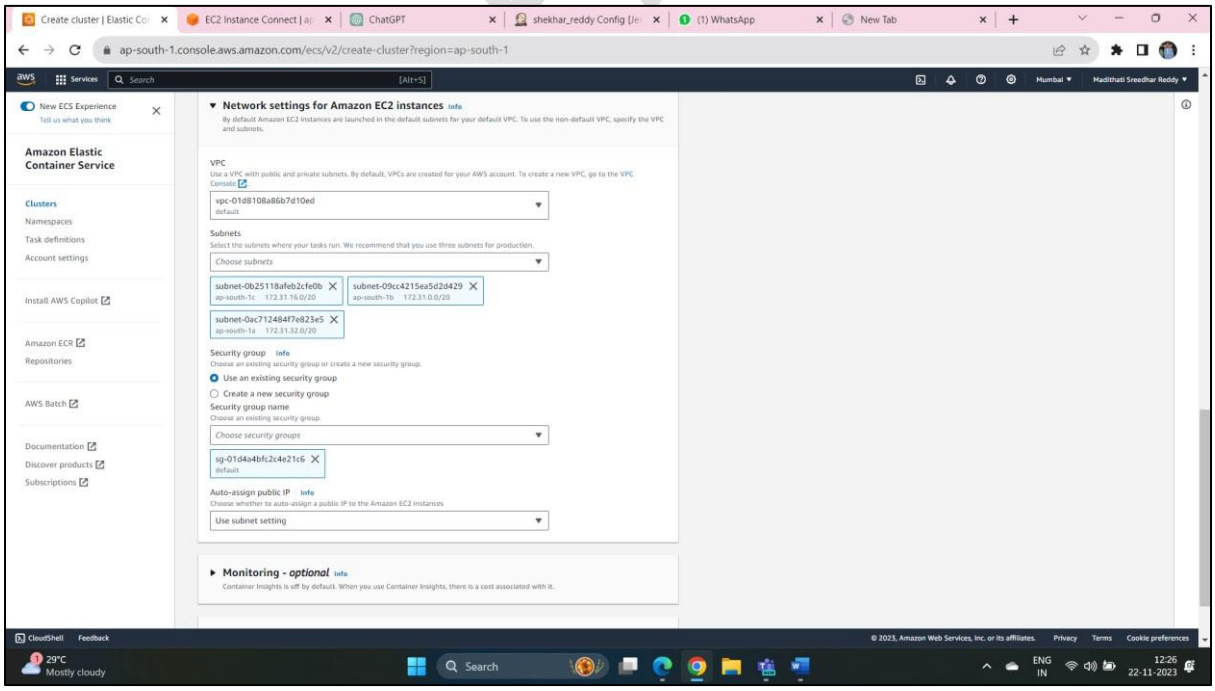


- Choose a cluster Infrastructure based on your requirements:
- **Amazon EC2 Instance:** For EC2 instances using Linux and manual networking.
- **Fargate:** For serverless containers without the need to manage EC2 instances.

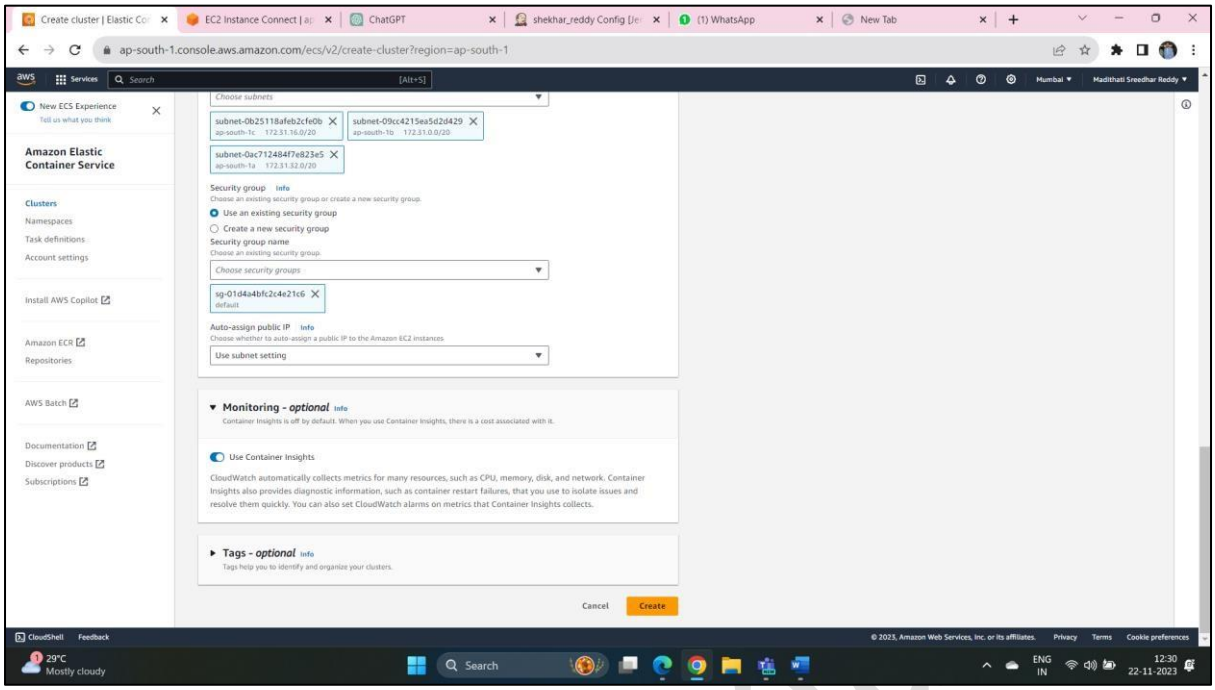
- Create an EC2 Instance based on your requirements.



- Number of Instances (if using EC2 instances): Set the initial number of instances in your cluster.
- VPC, Subnets, and other networking settings.



- Click on Monitoring and then turn on it for the CloudWatch Monitoring

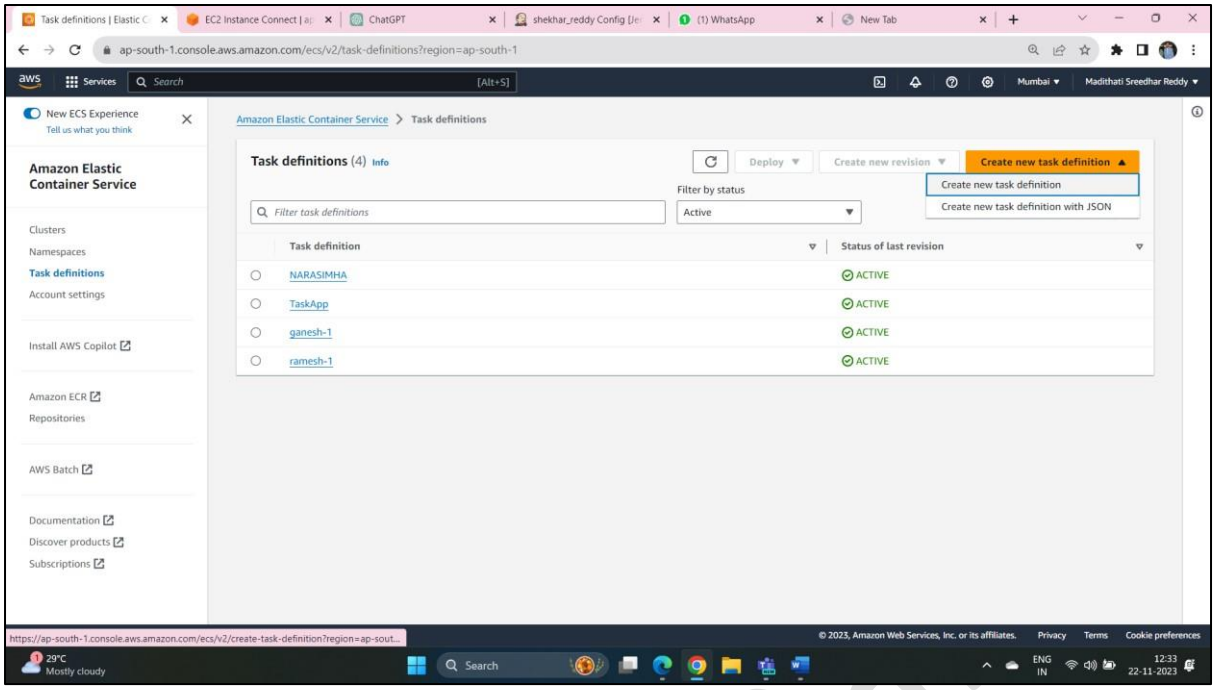


- Click "Create" to create the cluster.

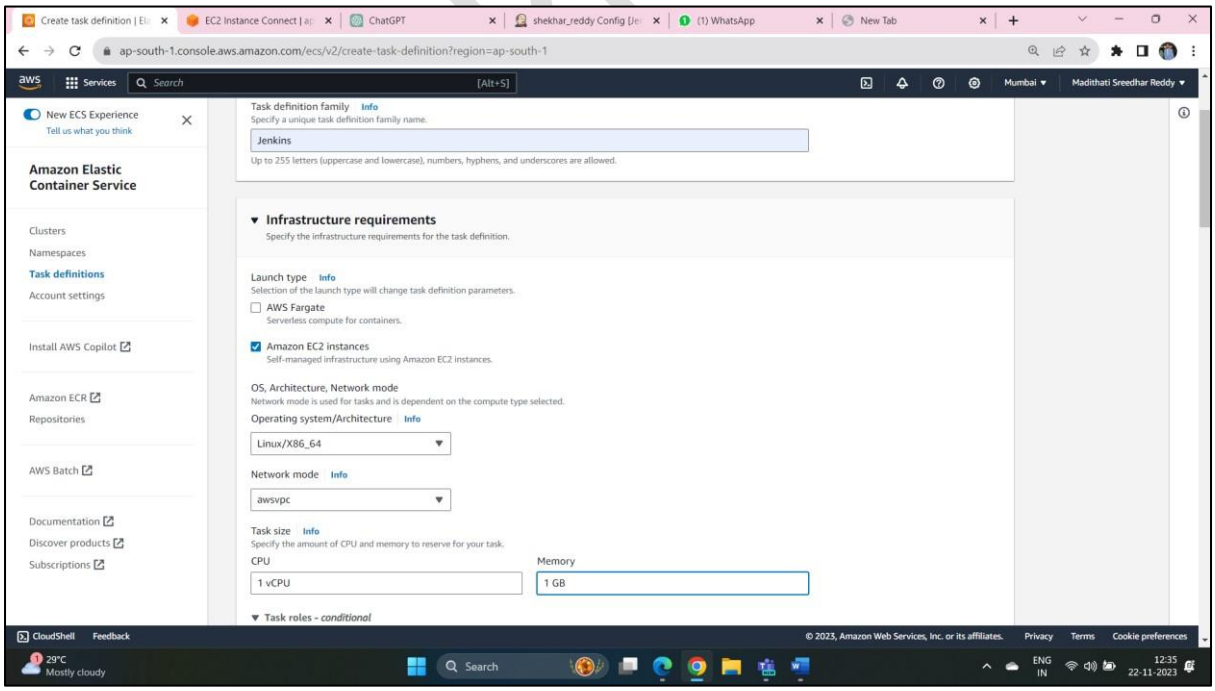
Creating ECS Task definition:

Creating an Amazon ECS (Elastic Container Service) Task Definition is a crucial step before launching containers on ECS. A task definition is a blueprint for your application, specifying various parameters such as the Docker image, CPU and memory requirements, environment variables, networking information, and more. Here how you can create an ECS Task Definition.

- Open the AWS Management Console and navigate to the ECS service.
- In the ECS console, click on "Task Definitions" in the left navigation pane.



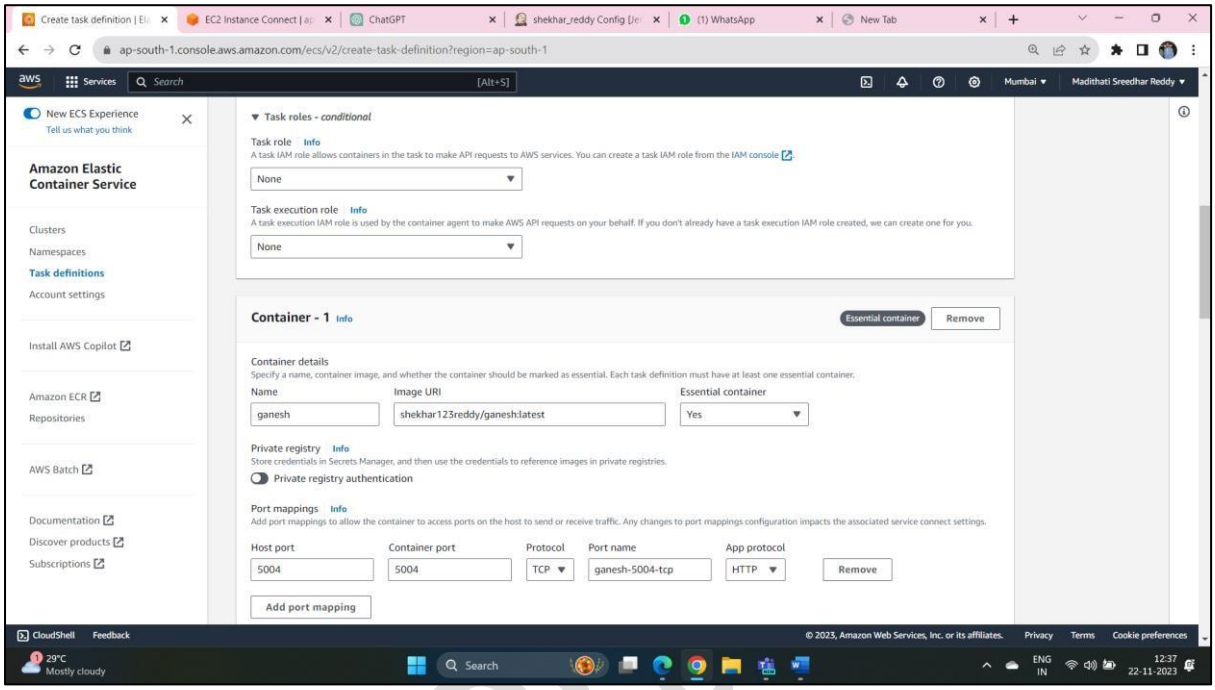
- Click the "Create new Task Definition" button.
- Specify a unique task definition Family name.
- Choose whether your task definition is compatible with EC2 instances, Fargate.



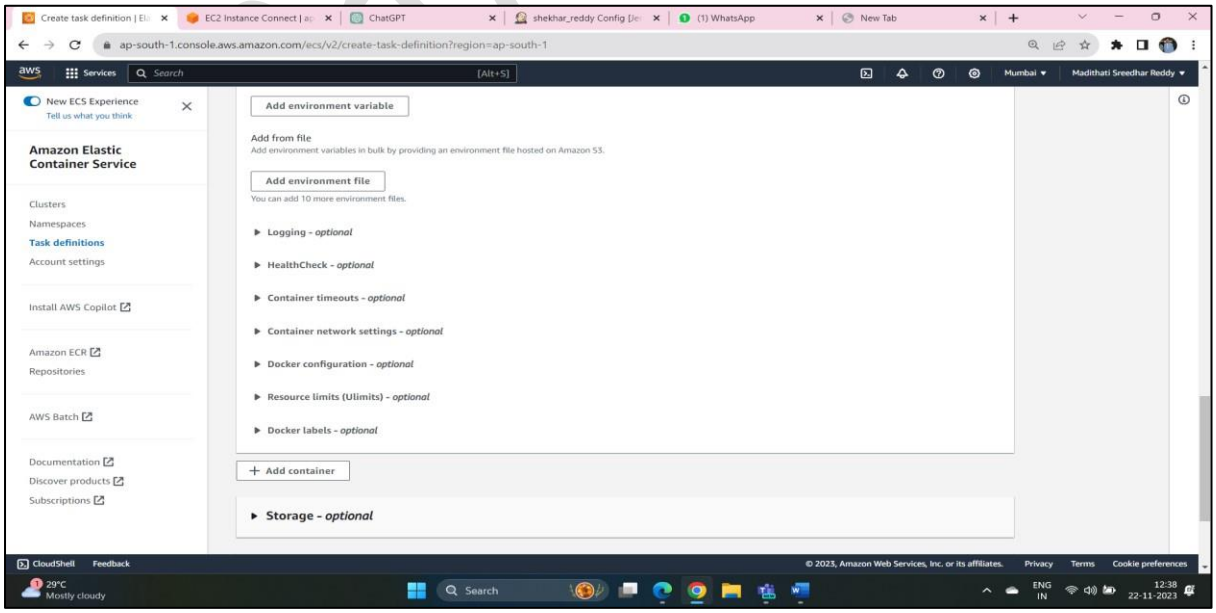
- Select Network mode as Default
- Optionally, specify an IAM role for your task execution role.

Provide details for your container, including:

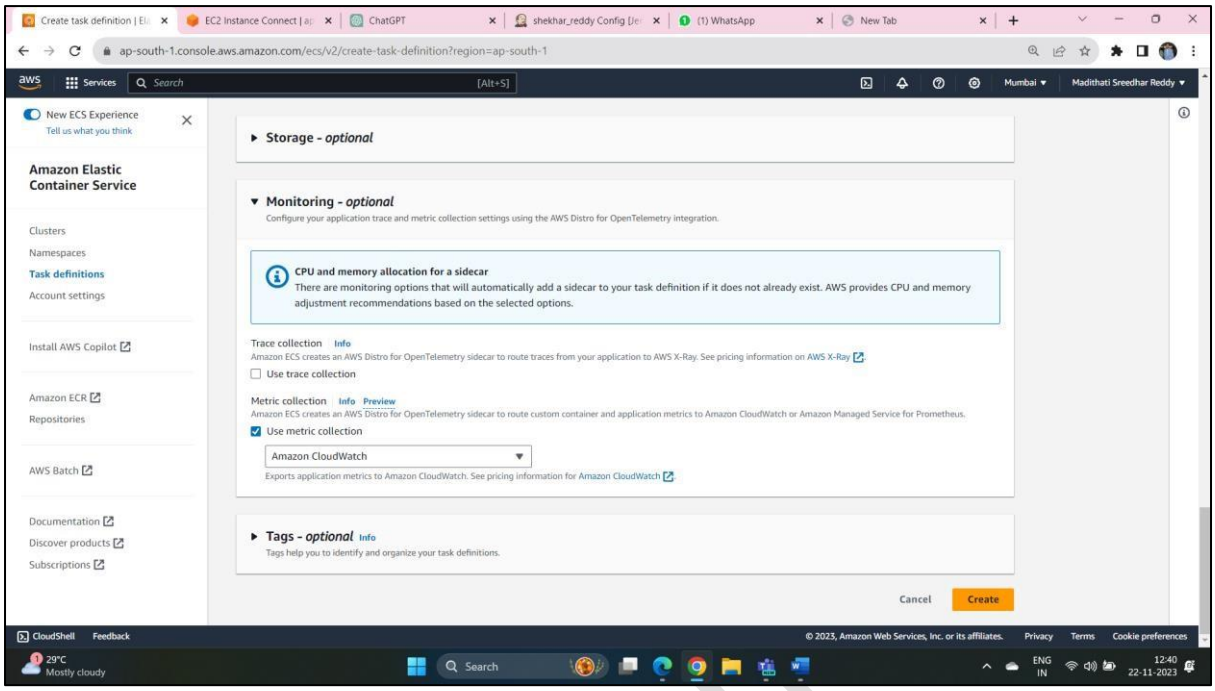
- **Container Name:** A unique name for your container.
- **Image URI:** The Docker image to use.
- **Port Mappings:** Give Host Port and Container Port



- **Environment:** Set environment variables if needed
- Configure Logging, Health Check, Docker configuration and resource limits depends on your need.



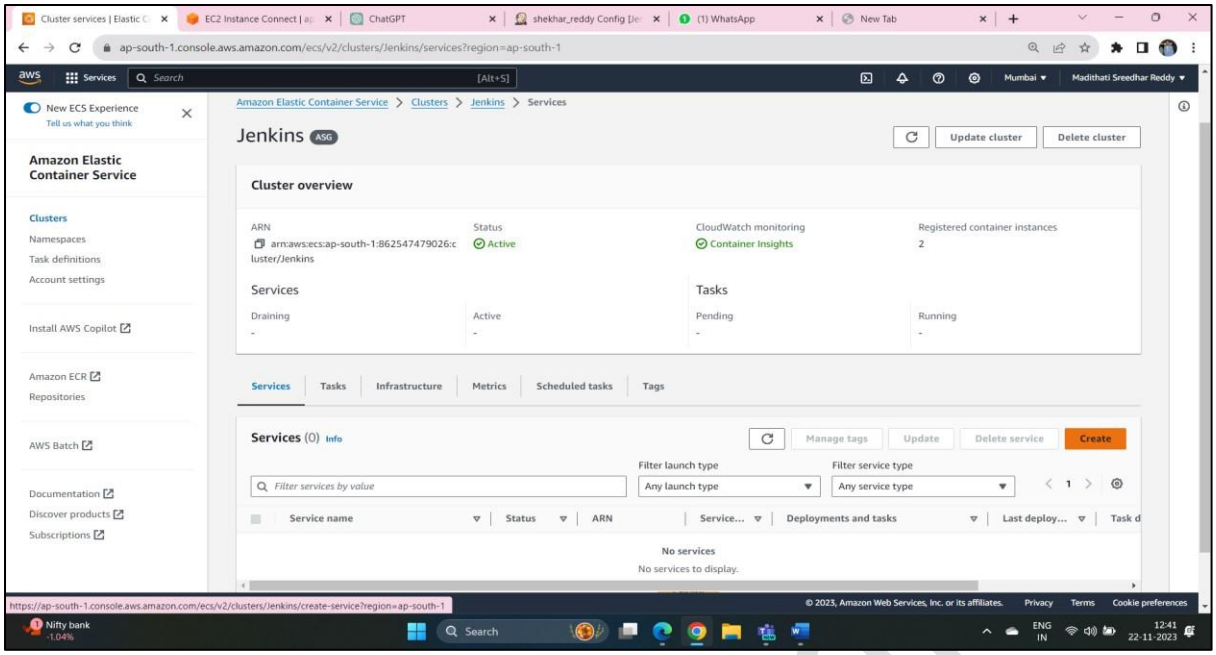
- Set up CloudWatch to monitor key metrics during deployments. For ECS, common metrics include CPU utilization, memory utilization, and the number of running tasks.



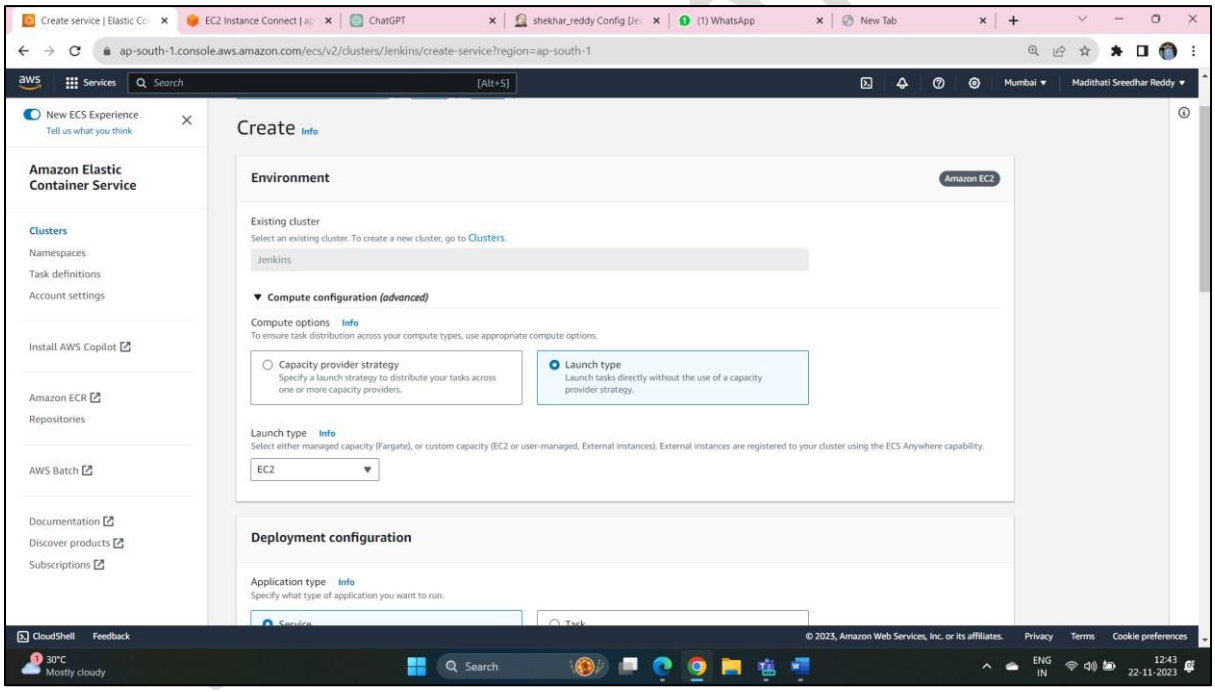
- Review the task definition configuration.
- Click the "Create" button to create the task definition.

Creating an ECS (Elastic Container Service) service in AWS:

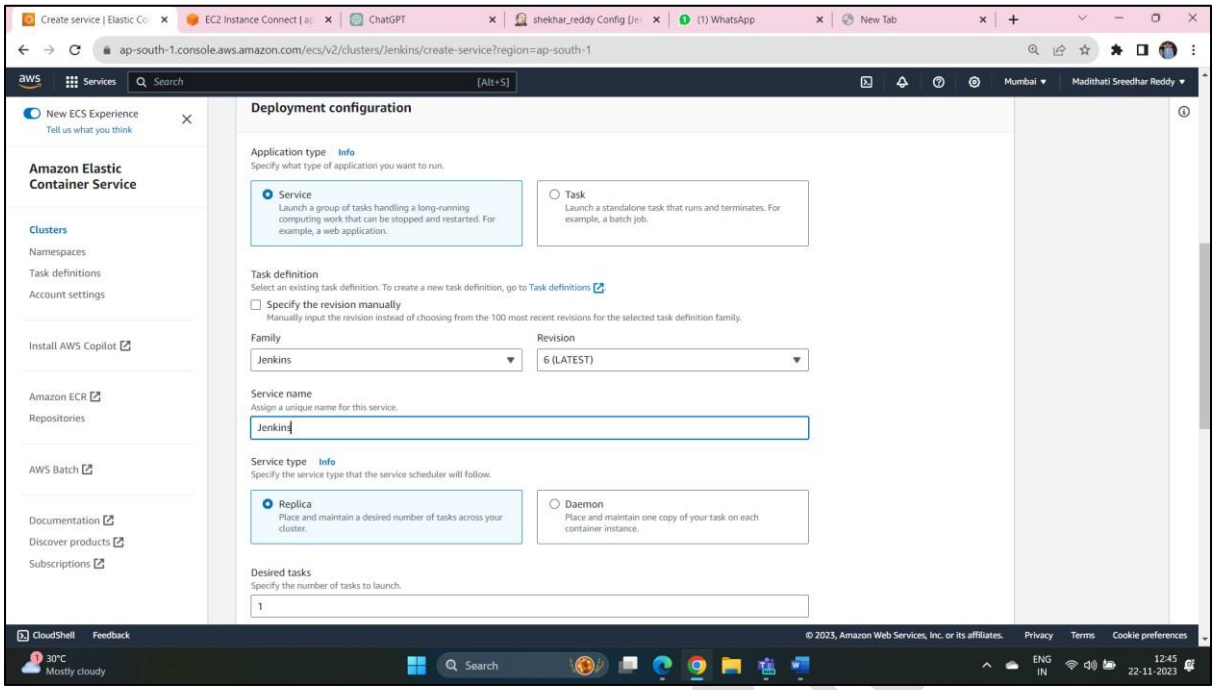
- Open the AWS Management Console and navigate to the ECS service.
- In the ECS console, click on "Clusters" in the left navigation pane.
- Click on the name of the ECS cluster where you want to create the service.
- In the cluster details page, click the "Create" button next to "Services."



- Select the Configuration.



- Configure as below.



In Amazon ECS (Elastic Container Service), there are two primary deployment types:

Blue/Green Deployment:

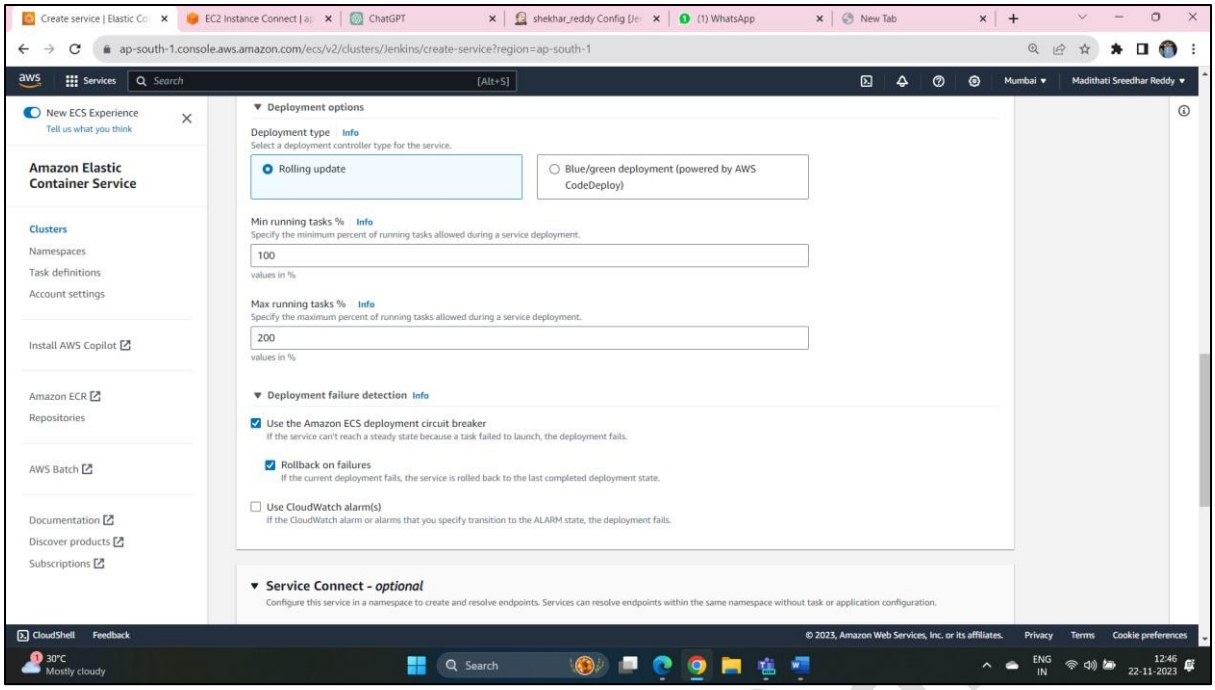
Blue/Green Deployment is Both the old ("blue") and new ("green") versions of the application are deployed in parallel. Provides minimal downtime during the deployment process. Enables thorough testing and validation of the new version before switching traffic. Allows for a quick rollback to the previous version if issues are discovered.

Rolling Deployment:

Updates are applied incrementally to a subset of tasks while maintaining a specified number of healthy tasks. Provides a continuous deployment process, gradually updating tasks without downtime. Allows for a controlled rollout of the new version across the ECS cluster. Optimizes resource usage by replacing tasks gradually.

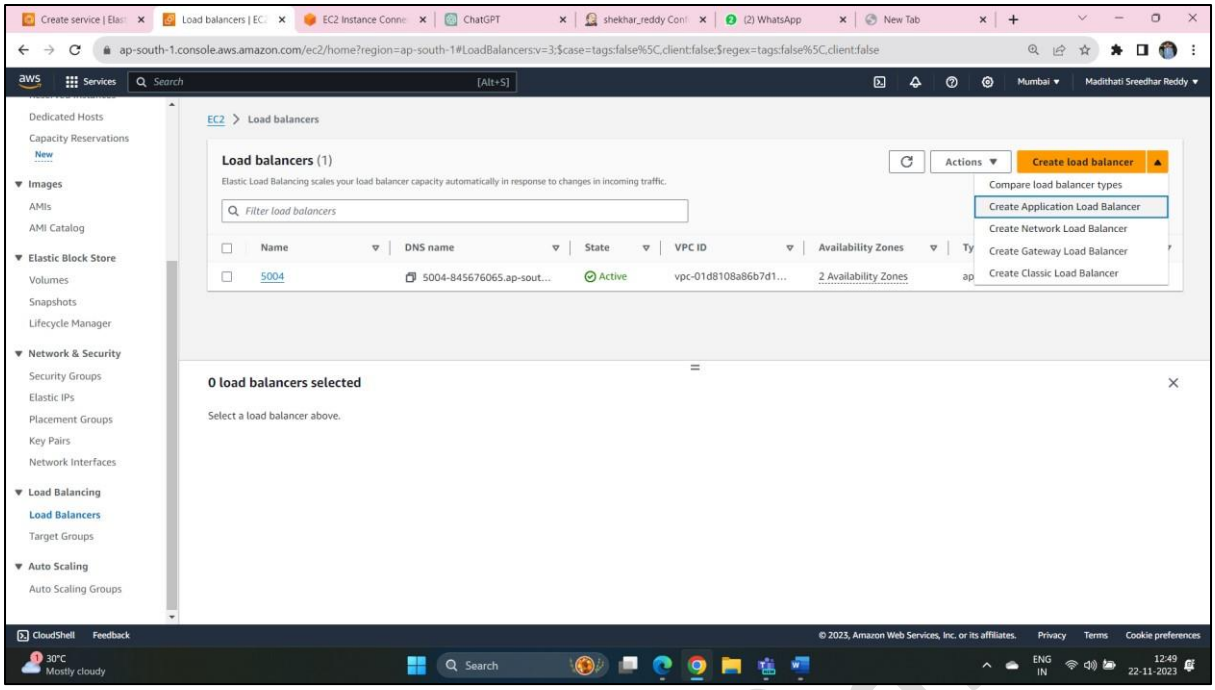
Rollback procedures in ECS involve reverting to a previous version of your application or infrastructure in case of issues.

- Here use the Rolling Update Option by configuring as below.

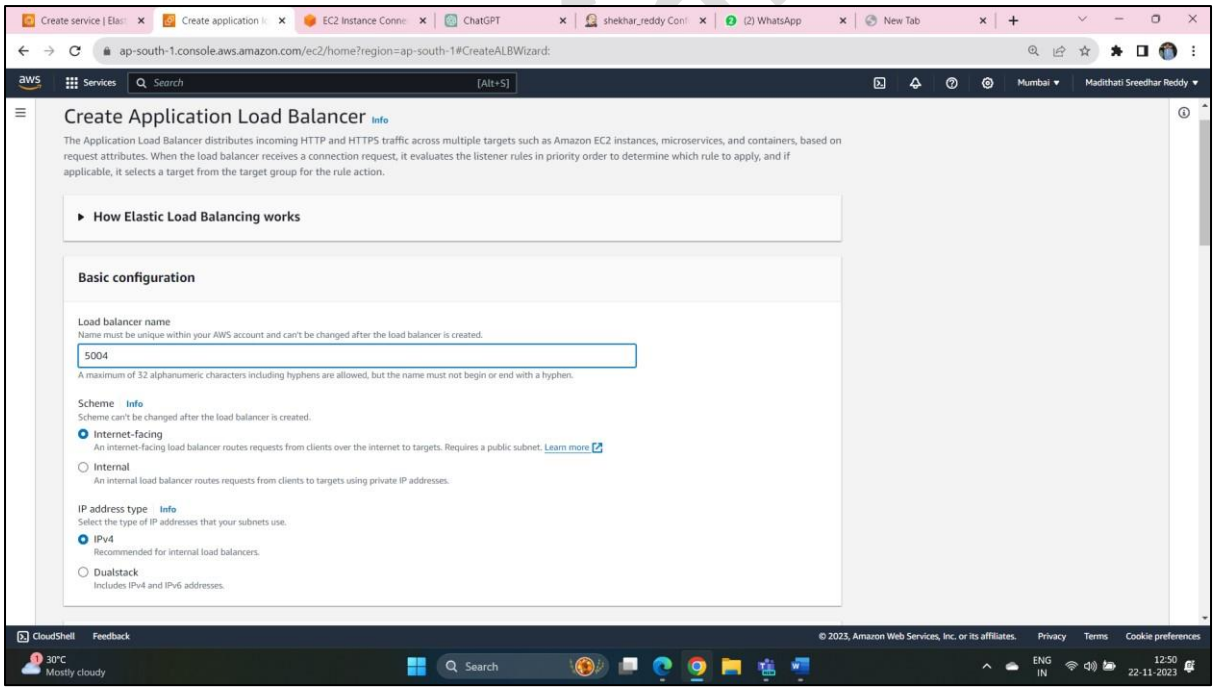


Create an Application Load Balancer (ALB)

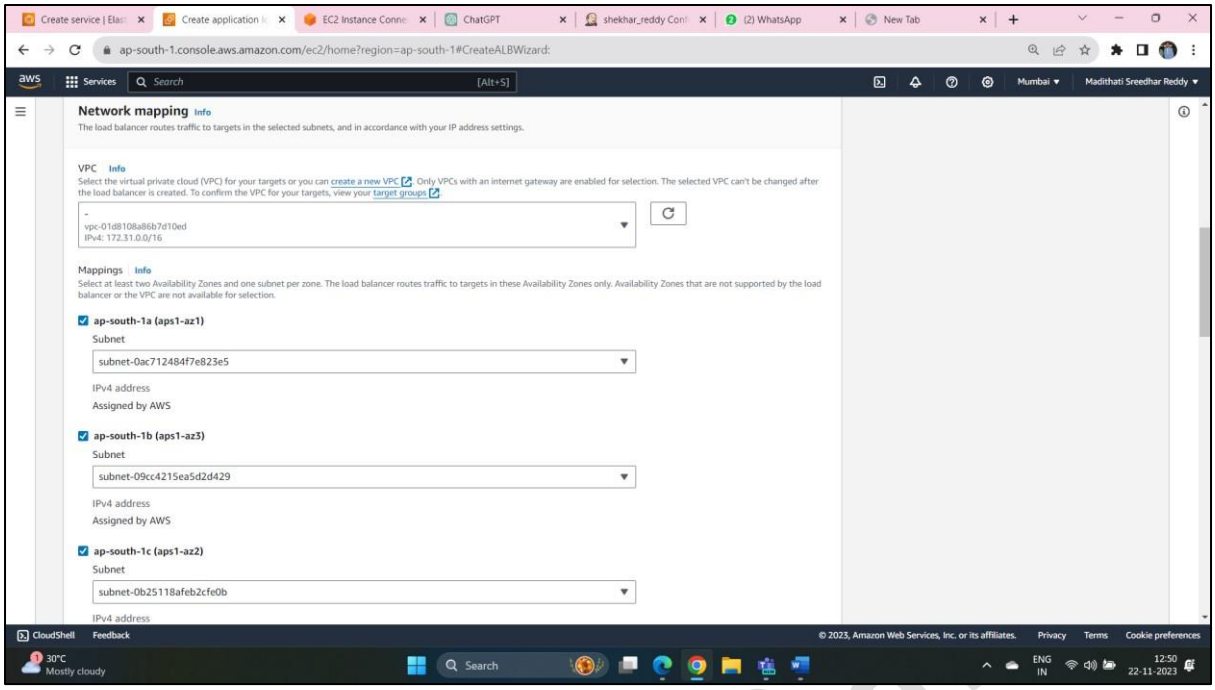
1. **Navigate to the EC2 Console:**
   - Go to the EC2 Dashboard.
2. **Create a Load Balancer:**
   - Click on "Load Balancers" in the left sidebar.
   - Click "Create Load Balancer."
   - Choose "Application Load Balancer."



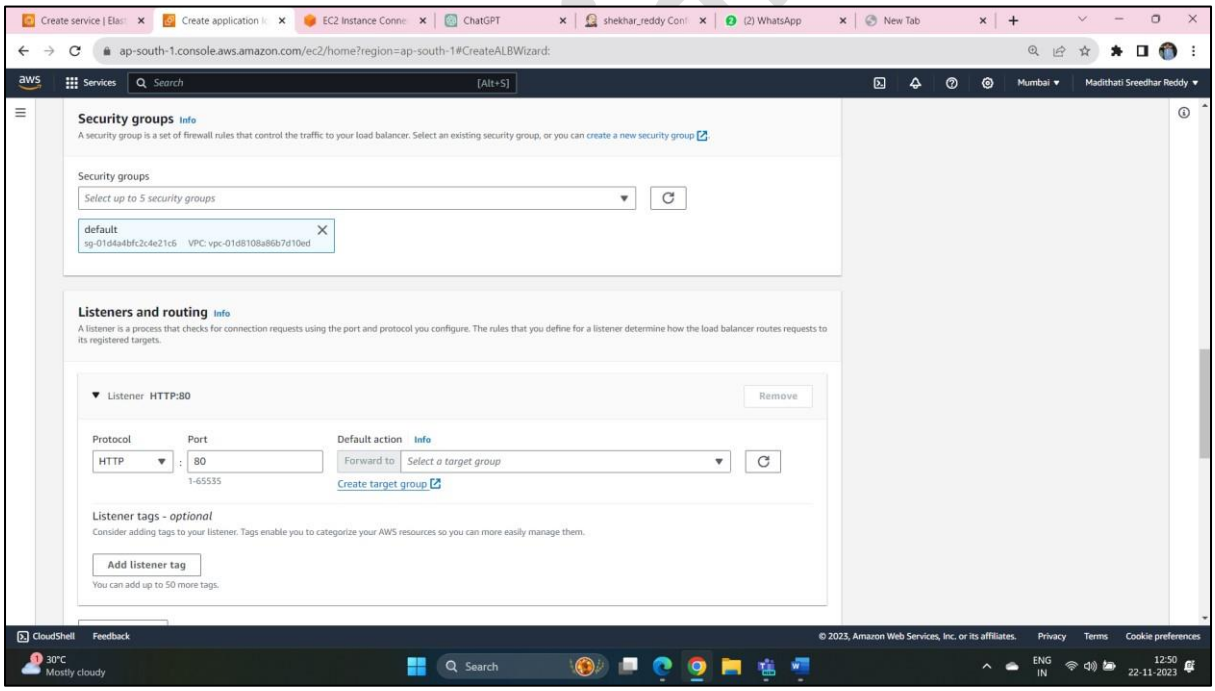
- Give a name for it.



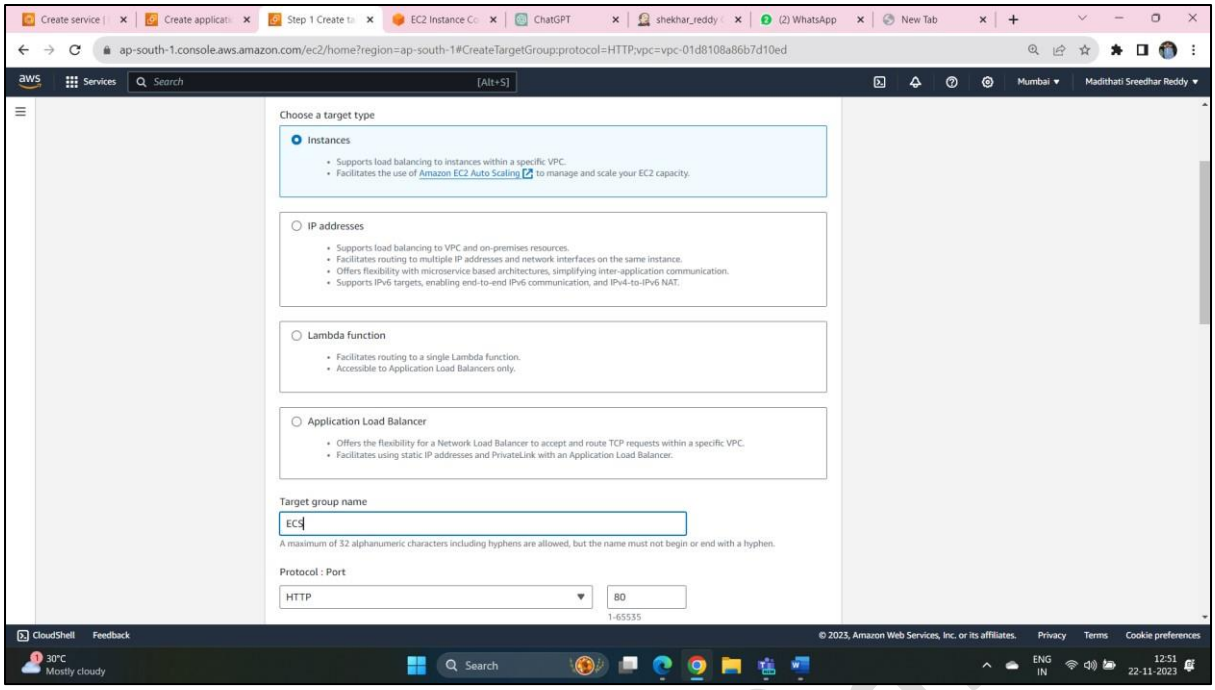
- Configure the load balancer settings (VPC, Subnets, Security Groups).



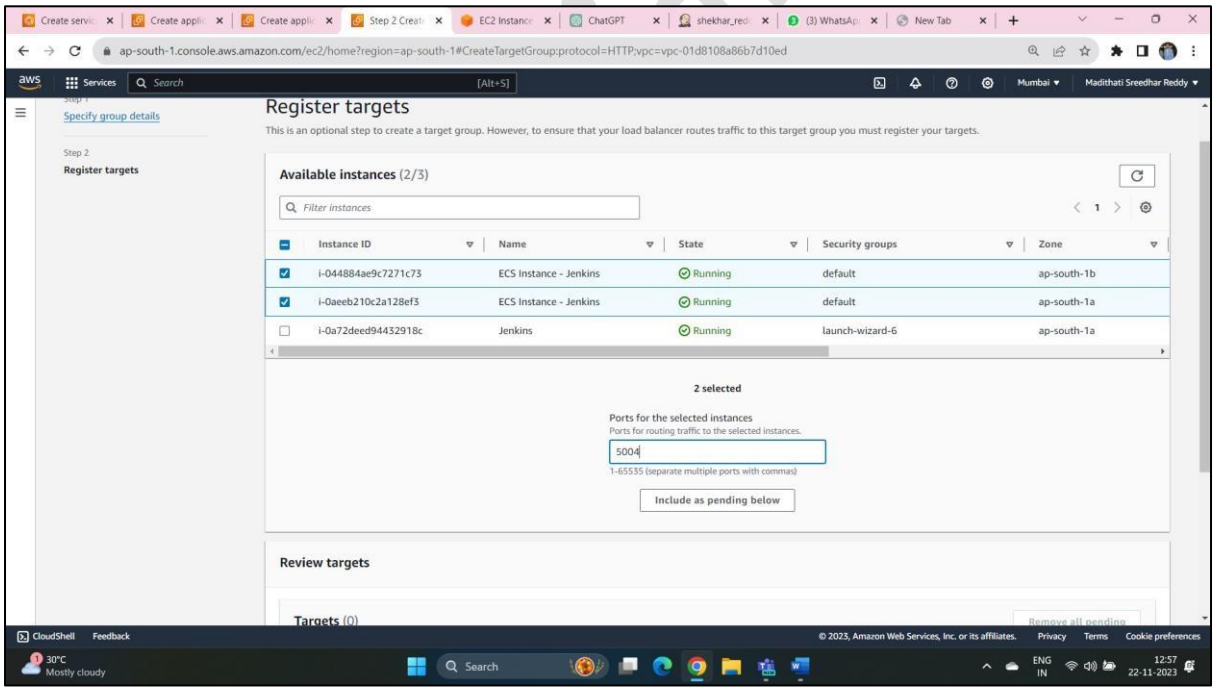
- Configure routing by adding listeners and target groups.



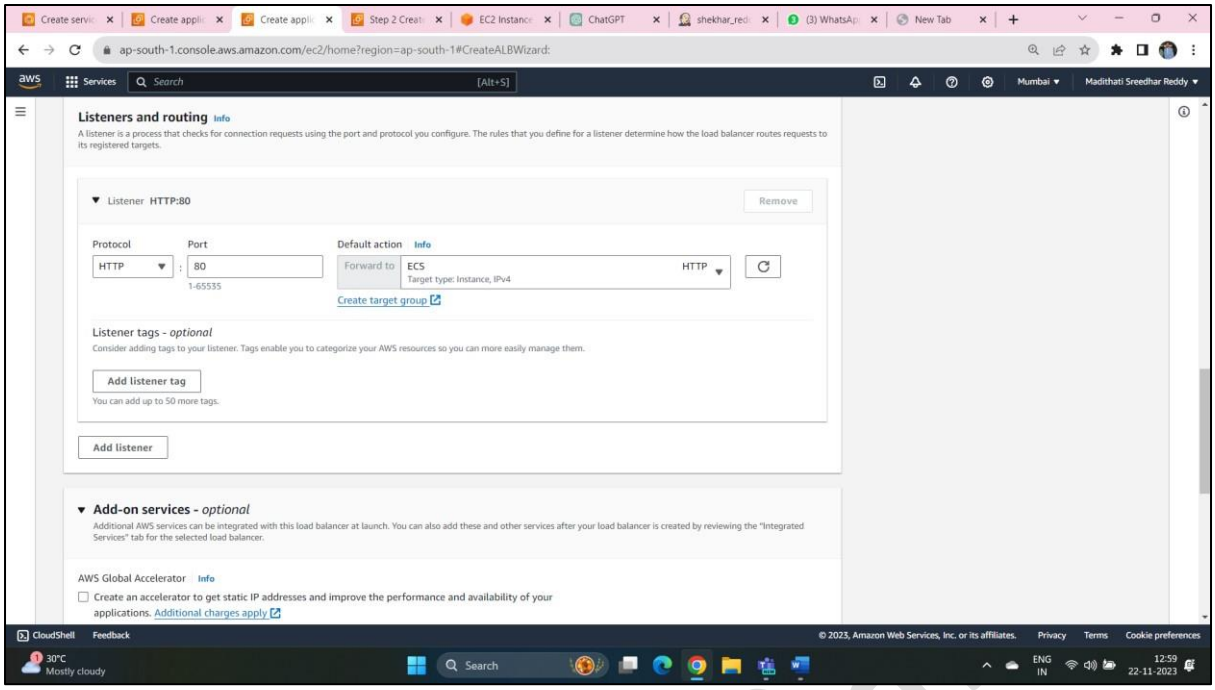
- For this Create a Target Group by clicking on the Blue Option present there



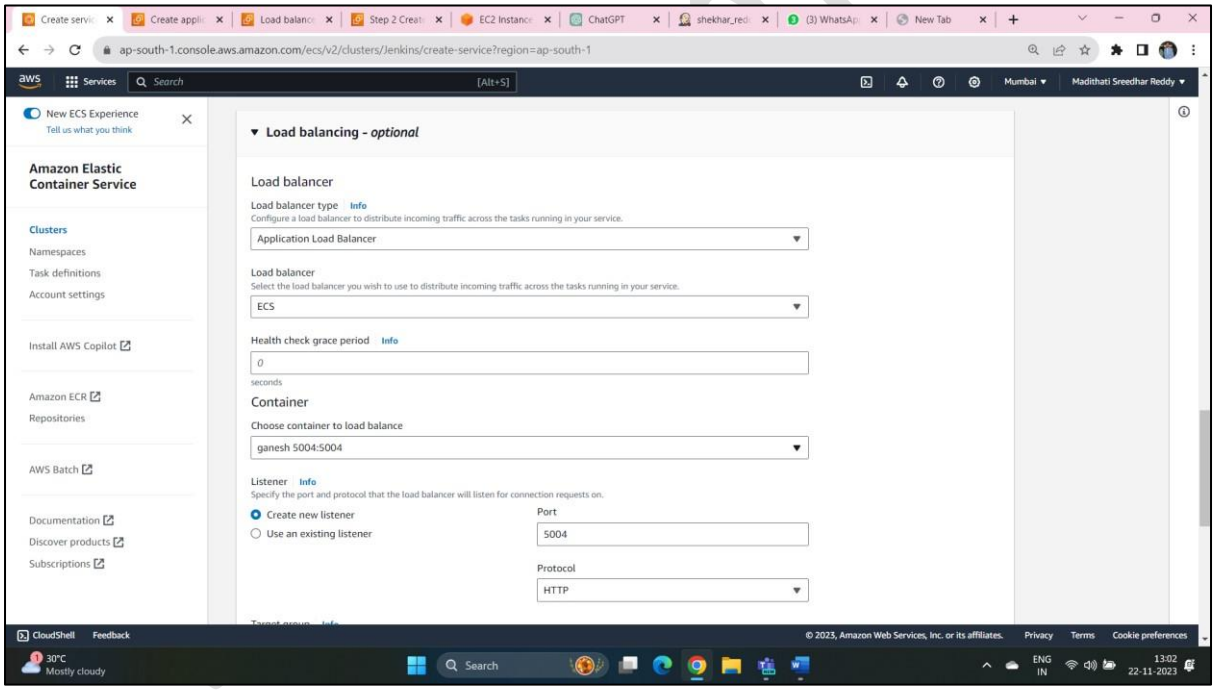
- Next Choose the targets as the registered instances of the created cluster.



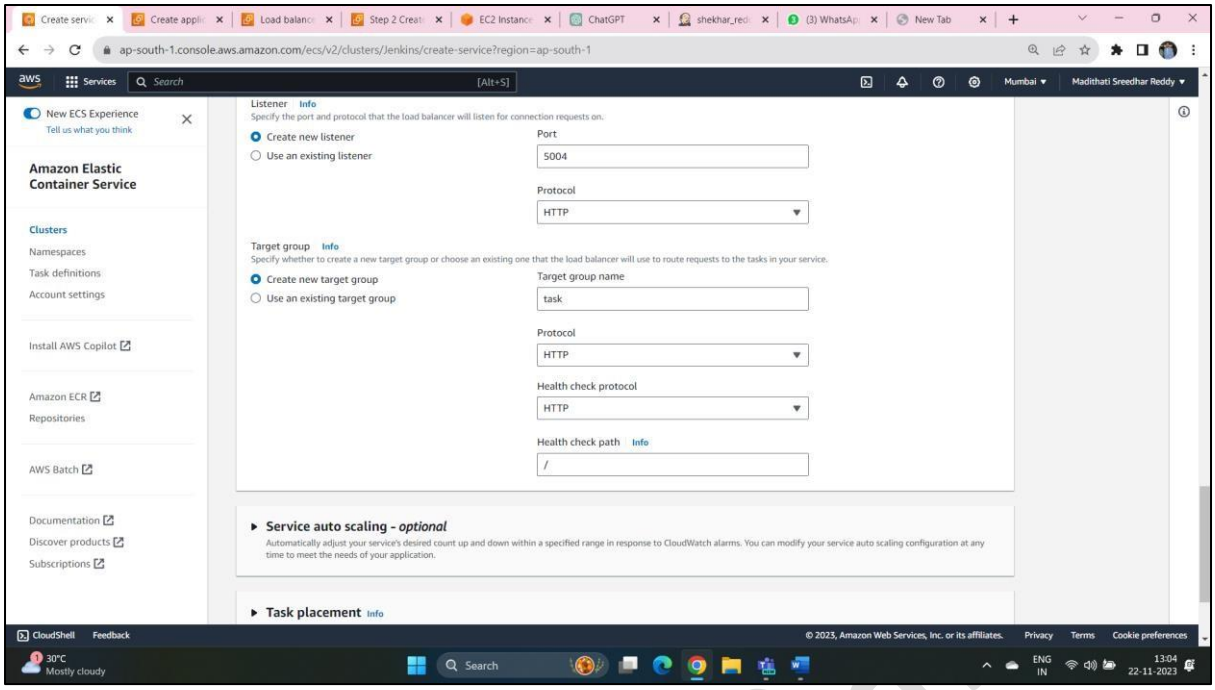
- Then return to the LoadBalancer and assign the created Register target.



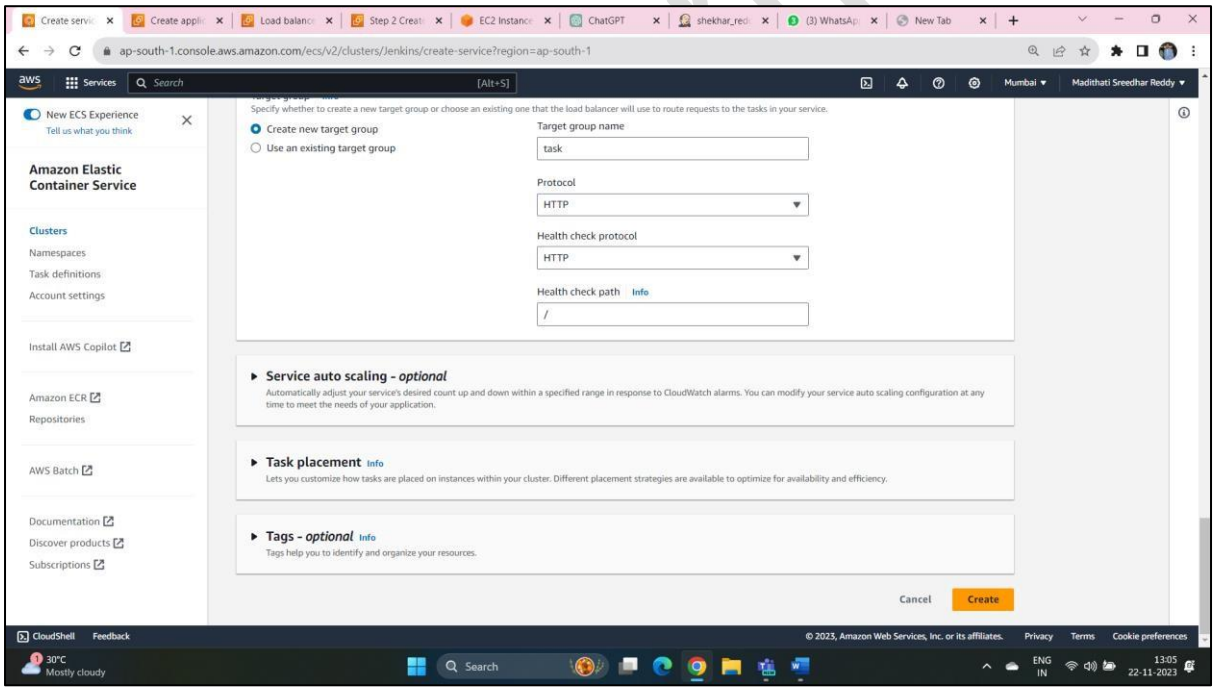
- Give that LoadBalancer under the LoadBalancing Section.



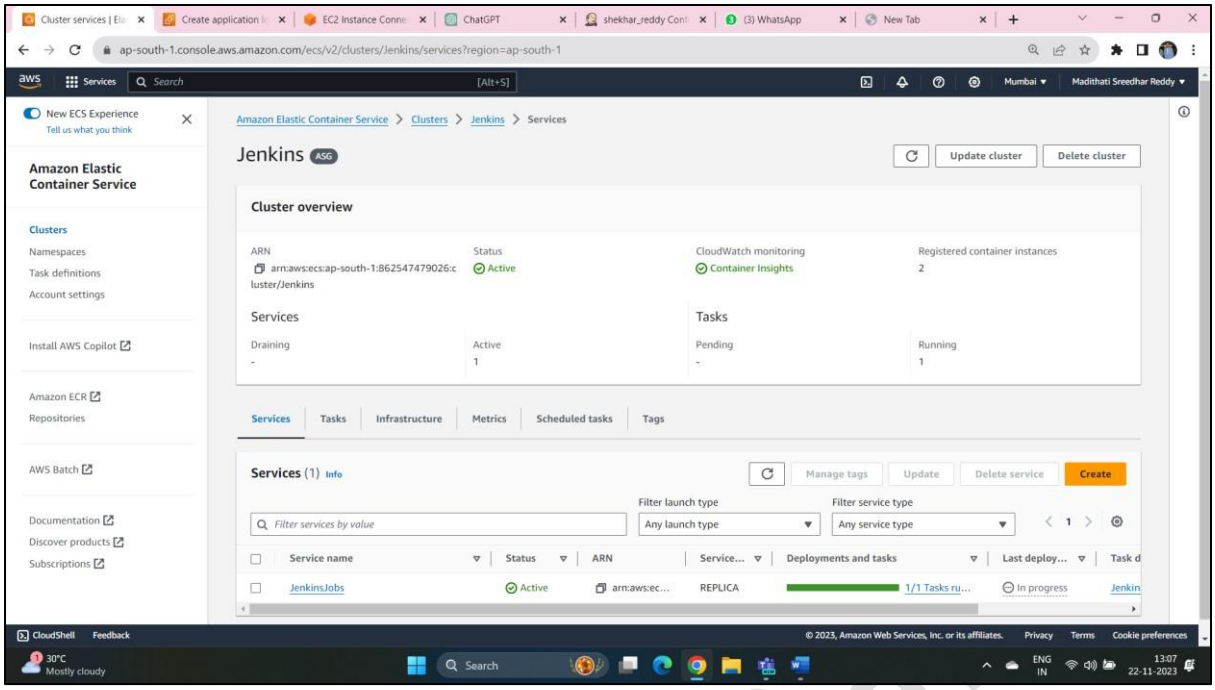
- Configure other Options as Below.



- Leave the Other Options as Optional.

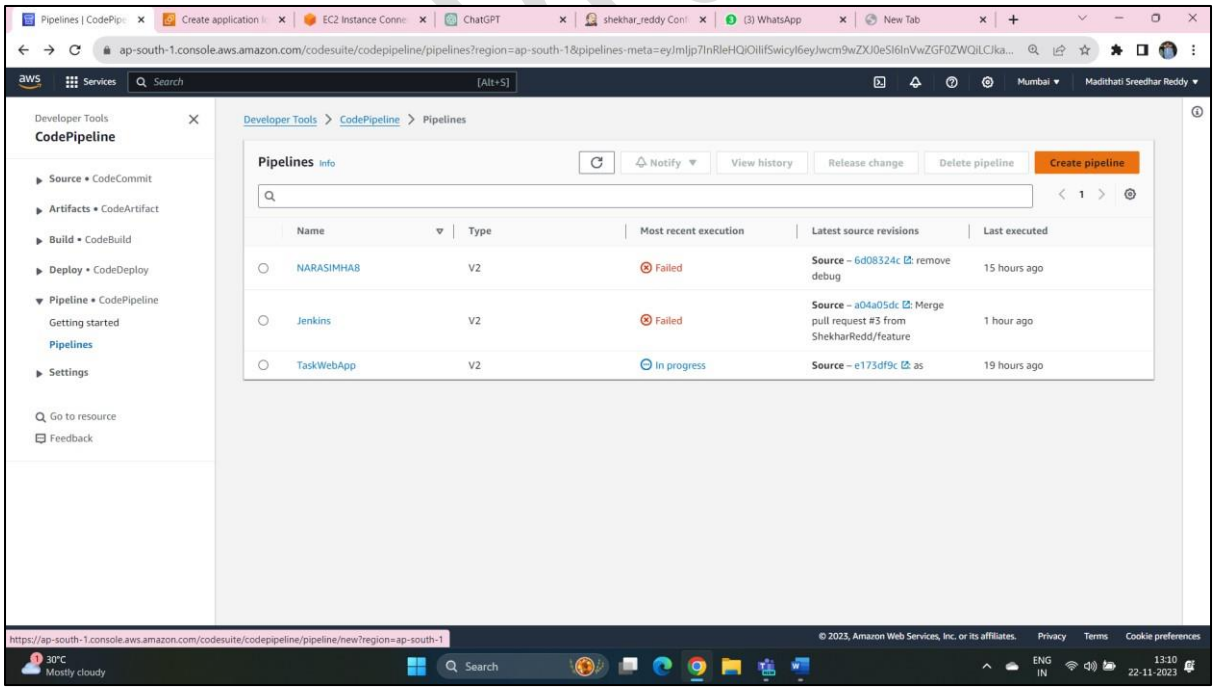


- Review all configurations.
- Click the "Create" button to create the ECS service.
- Once created, you can view the details of your service, including task status, events, and more
- If Everything goes right Once you open the cluster, It will Look As below.



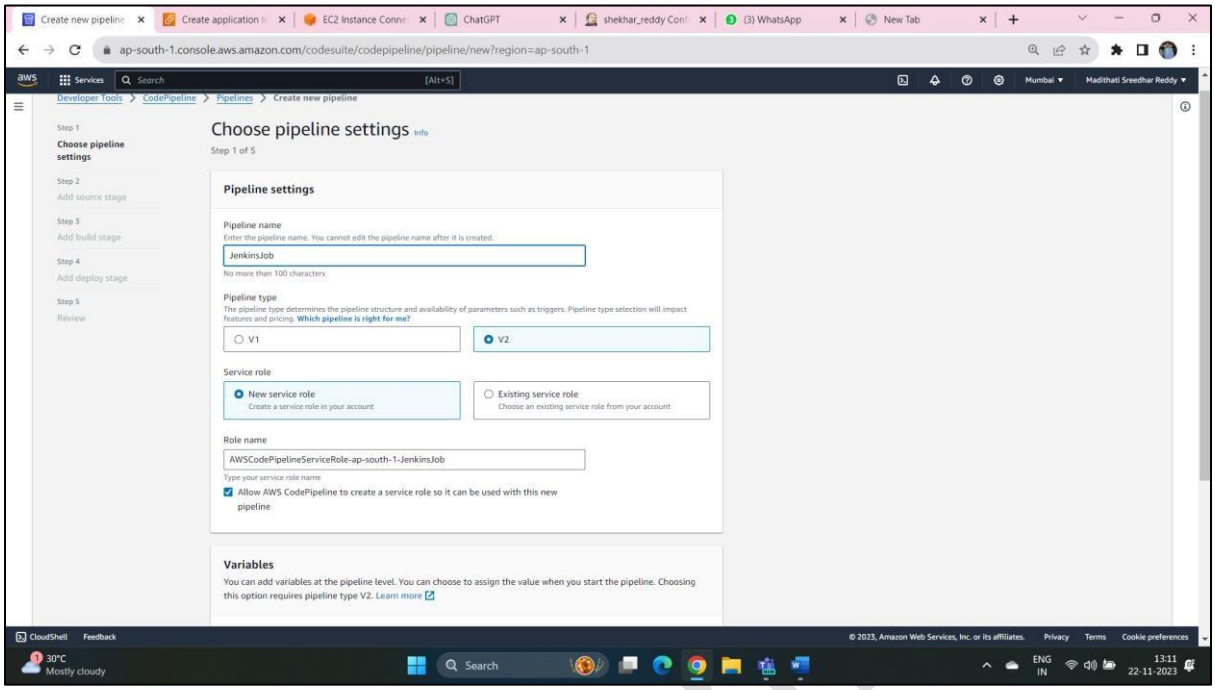
Create CodePipeline:

- In the AWS Management Console, navigate to CodePipeline.
- Click "Create pipeline."

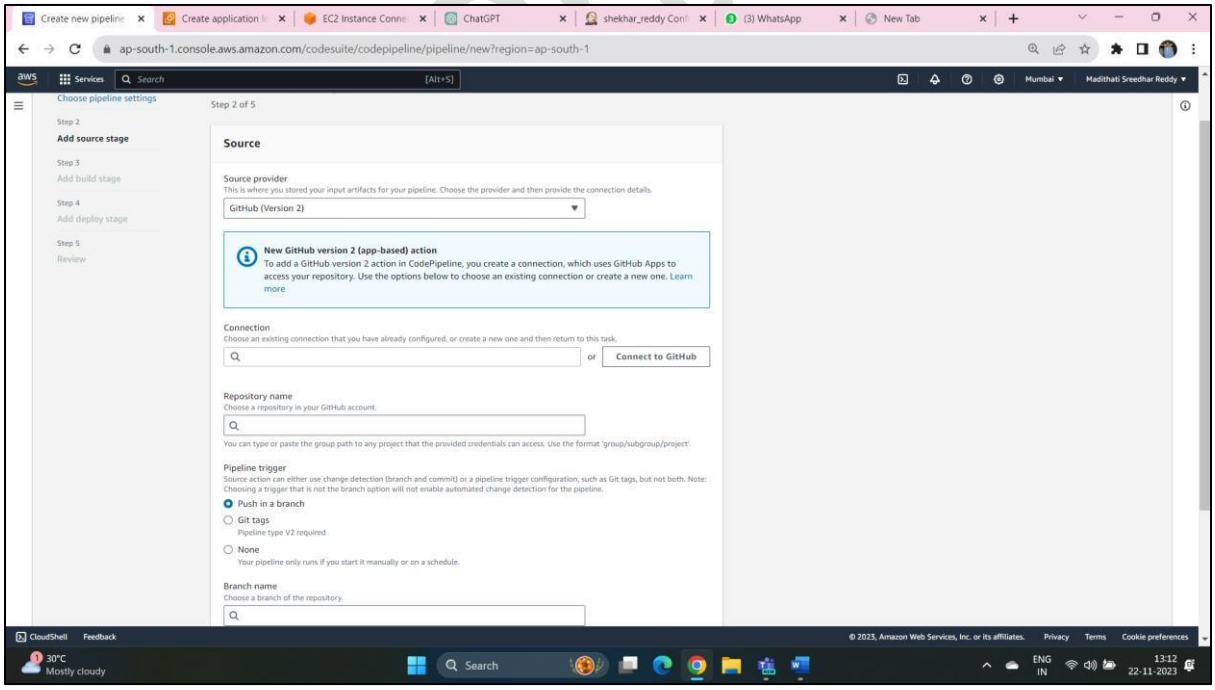


- Configure your pipeline with the following stages:
- **Source Stage:** Connect to your “GitHub” repository.
- **Build Stage:** Use “Jenkins” as the build provider.

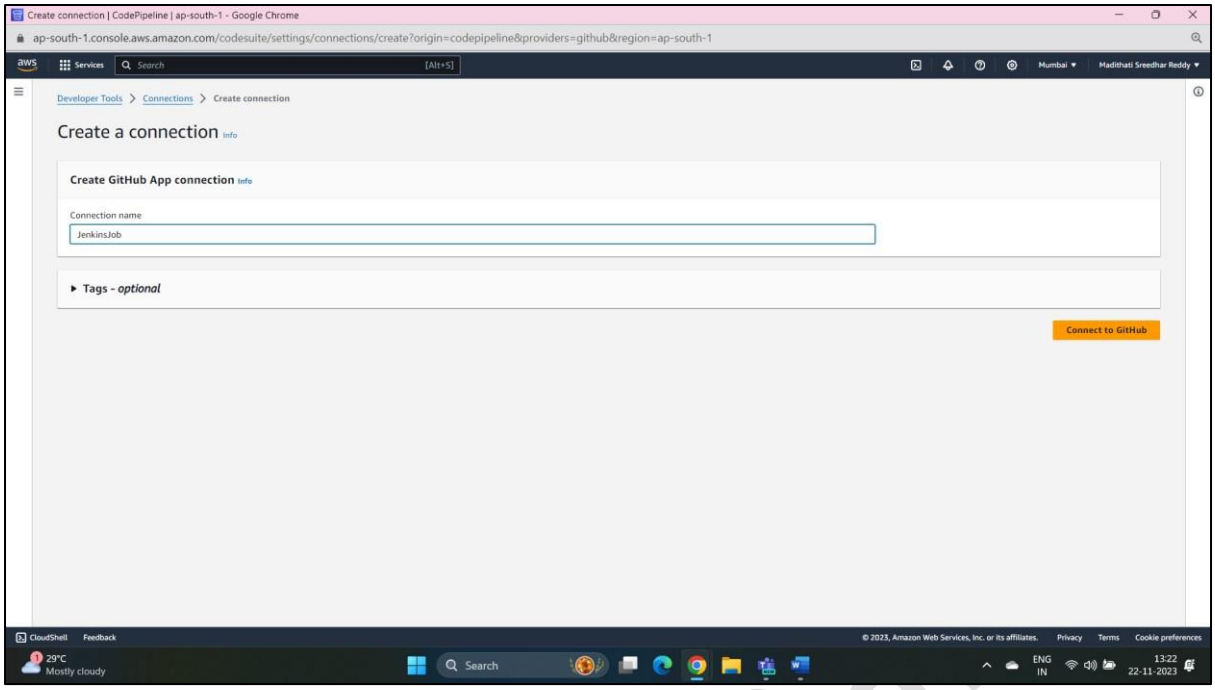
- **Deploy Stage:** Choose “ECS” as the deployment provider.
- Enter the Unique Pipeline name.



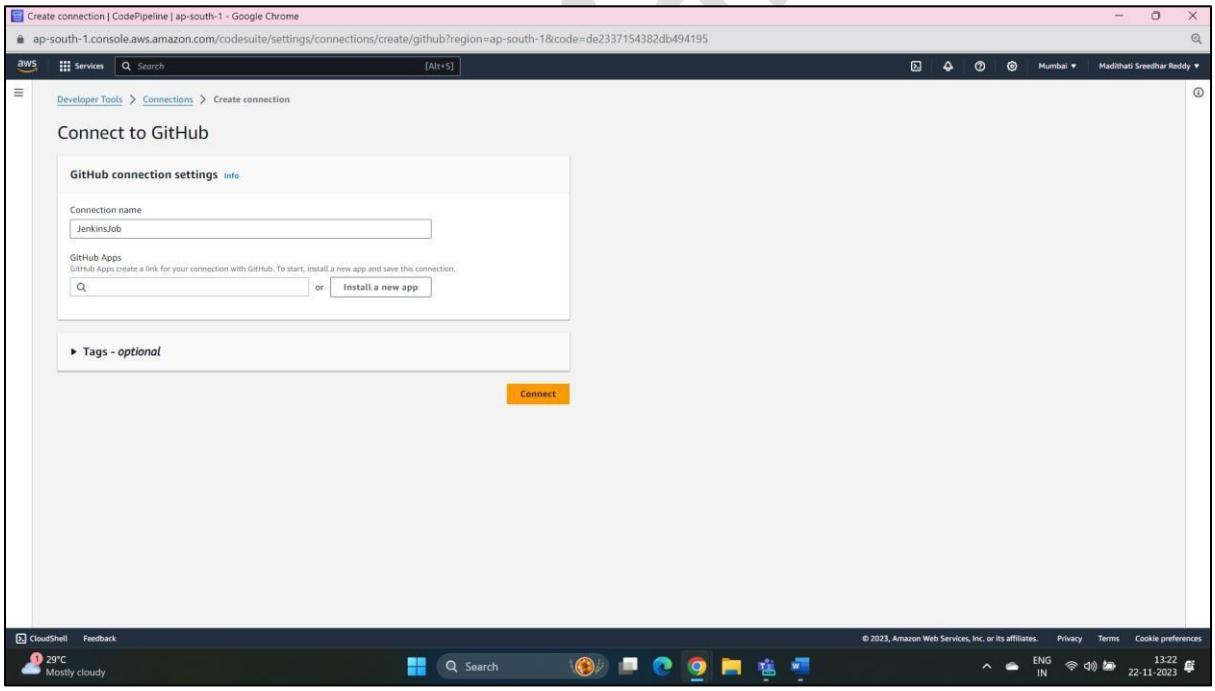
- Click next
- Choose the source provider (e.g., GitHub)



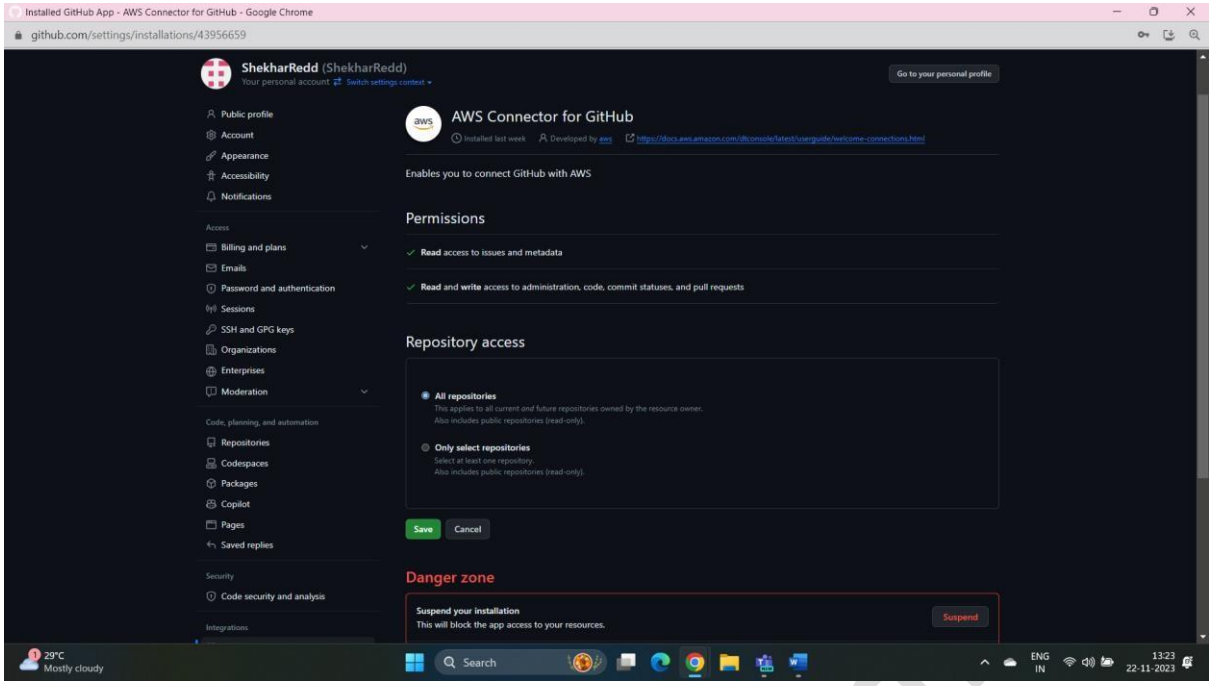
- Click on “Connect to GitHub”



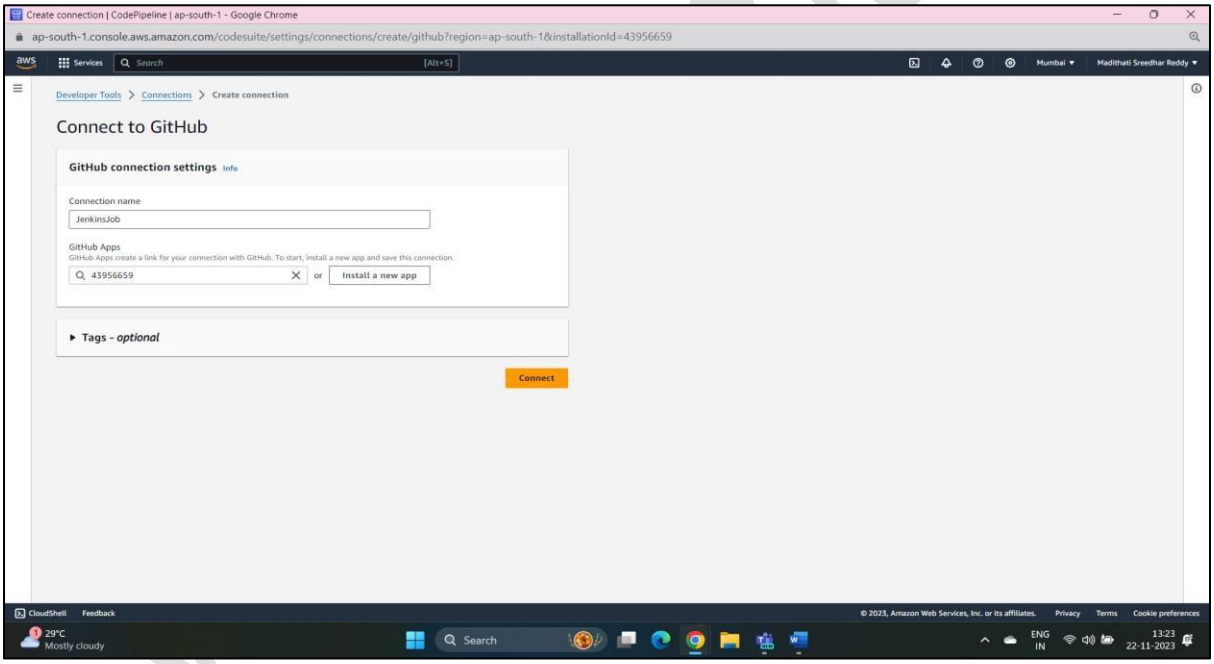
- Click on the tab mentioned there.



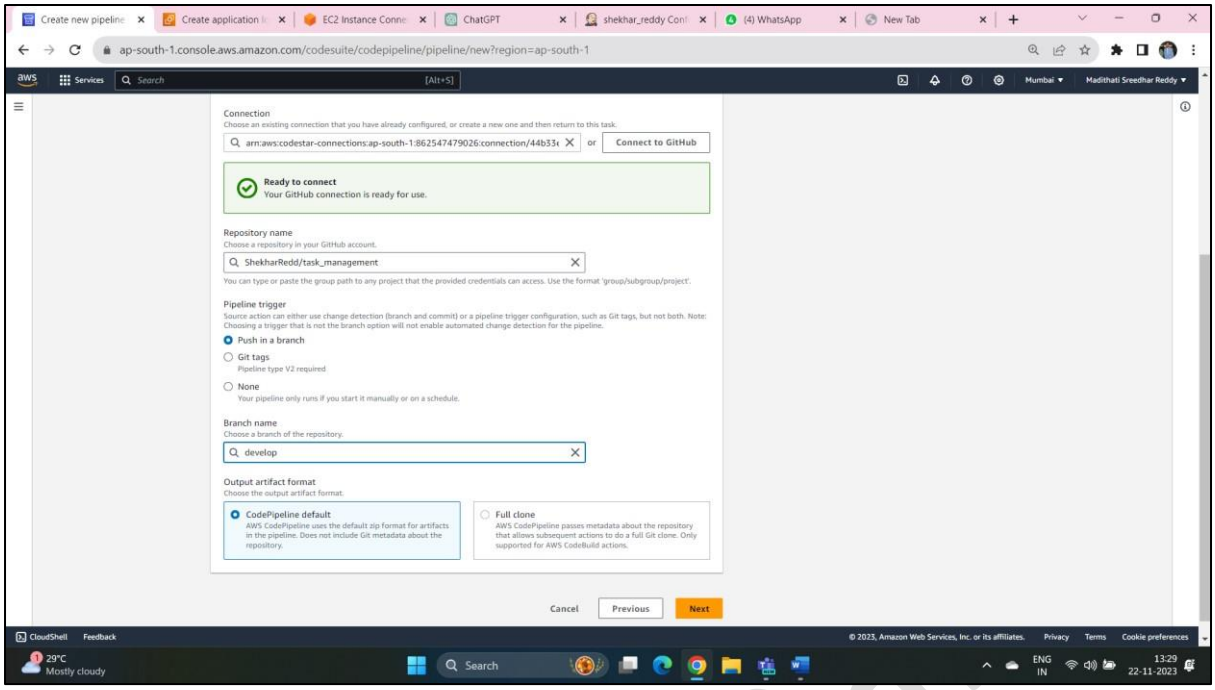
- Click on Install New App
- Next Enter the credentials of your GitHub
- Then Choose the Repo where your SourceCode relies.



- Click on save and then return to the AWS Console.



- Connect to your repository and select the branch to monitor.

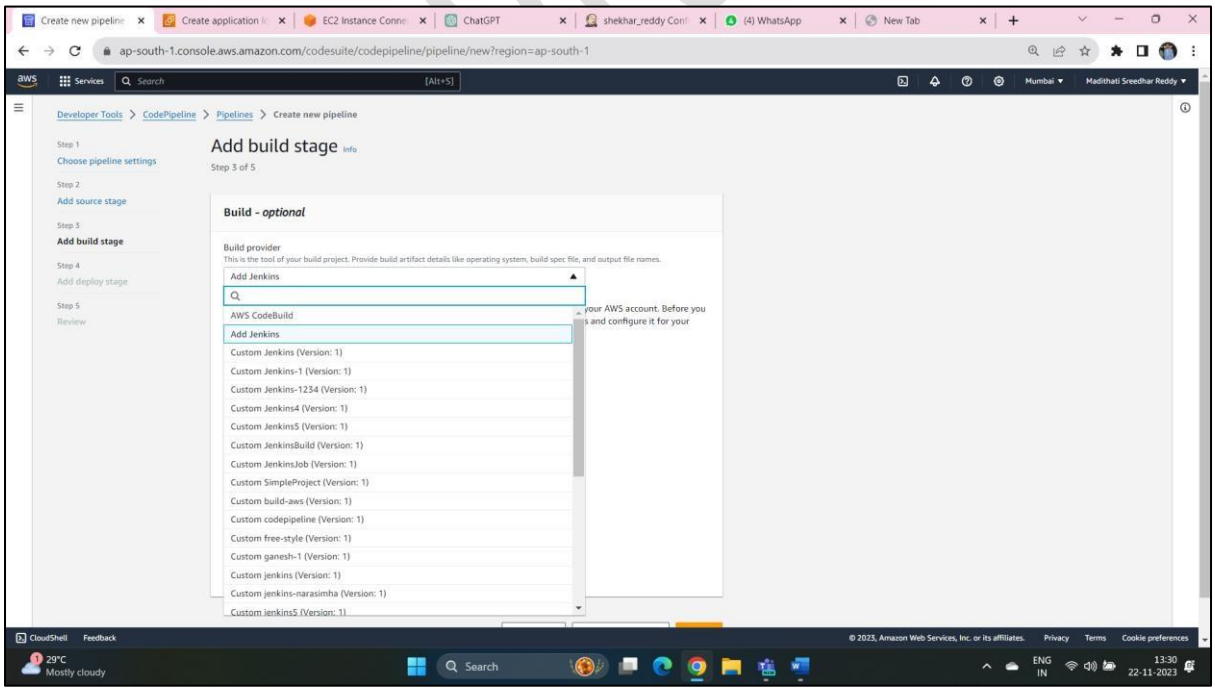


- Click on next

Configure Build Stage :

Configure the build settings

- Build Provider: Choose Build provider (Jenkins)

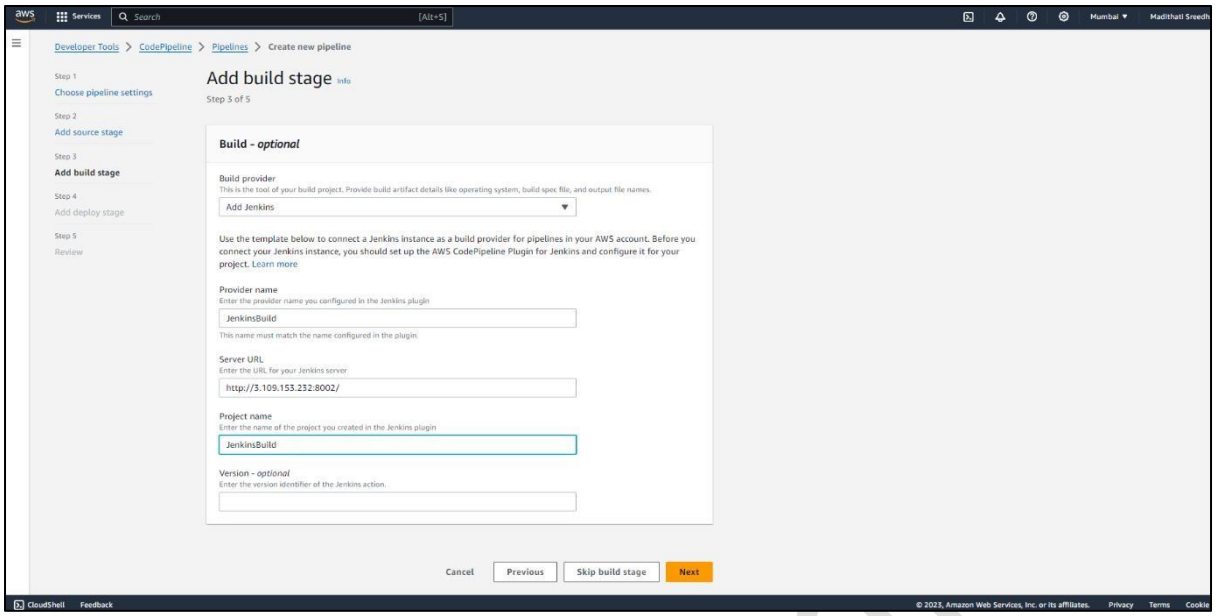


- Provide Name: Enter the provider name you configured in the Jenkins plugin

(Note: This name must match the name configured in the Pipeline)

- Server URL: Enter the URL of your Jenkins server

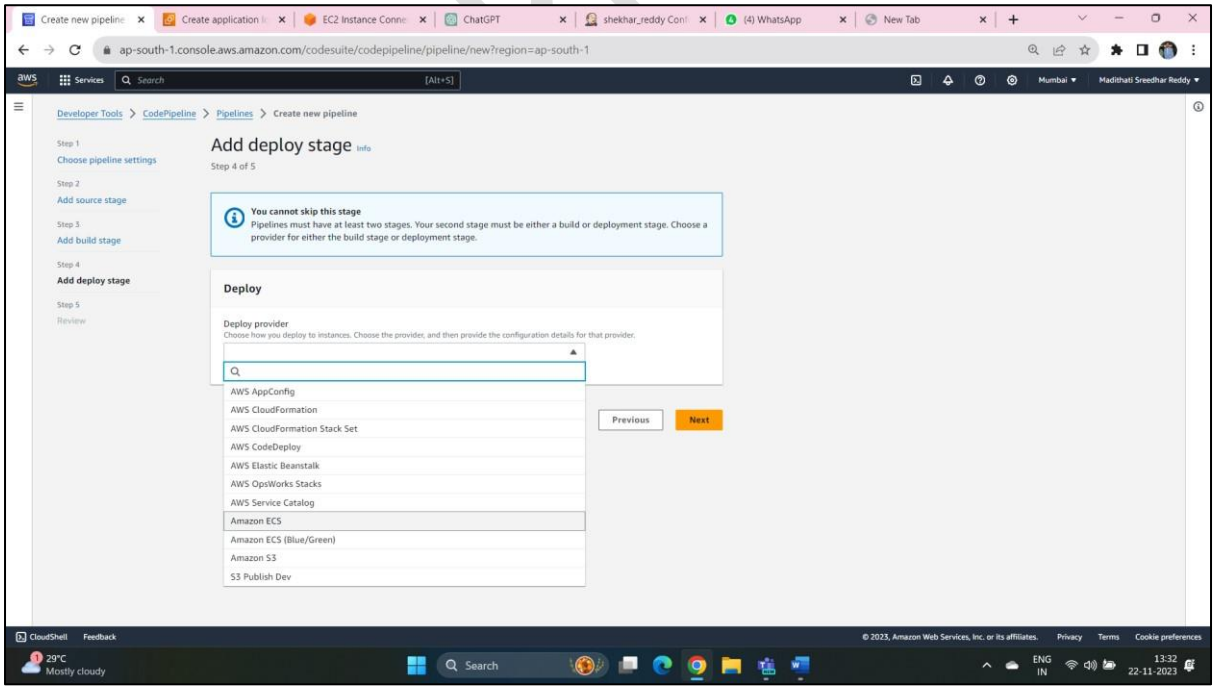
- Project name: Enter the project name you created in the Jenkins plugin



- Click next

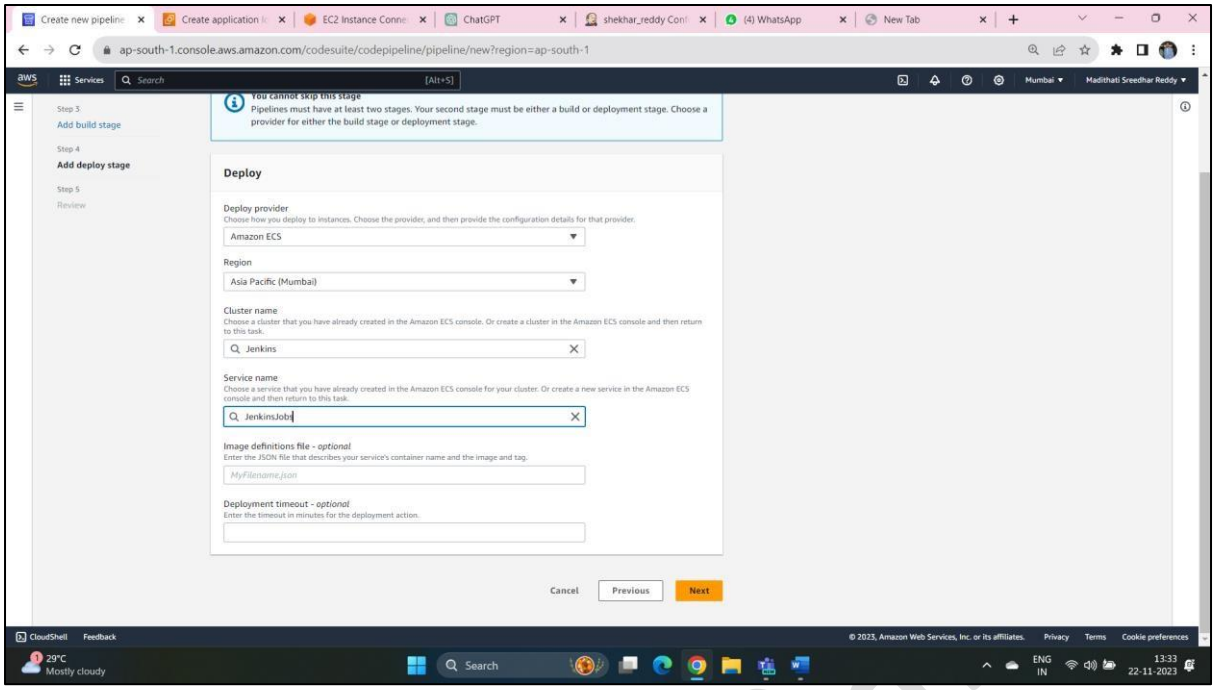
Configure Deploy Stage:

- Add a deployment stage based on your deploy provider (ECS).



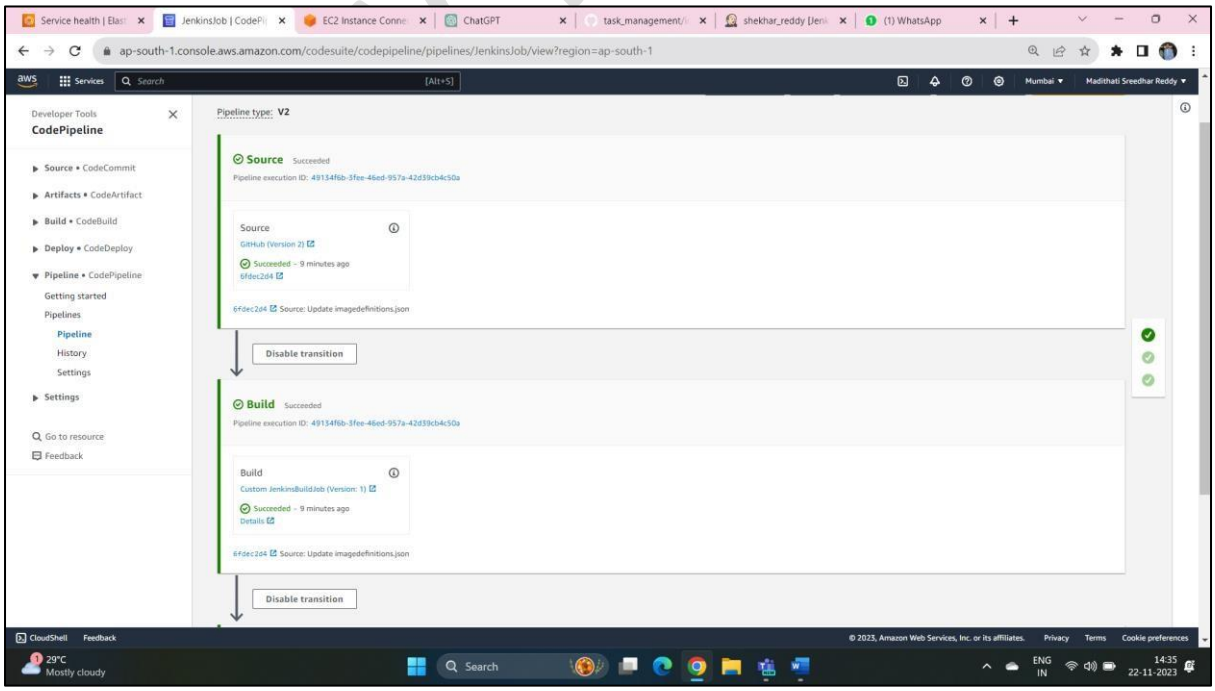
- Configure deployment settings, including the Cluster name, service name and any required parameters

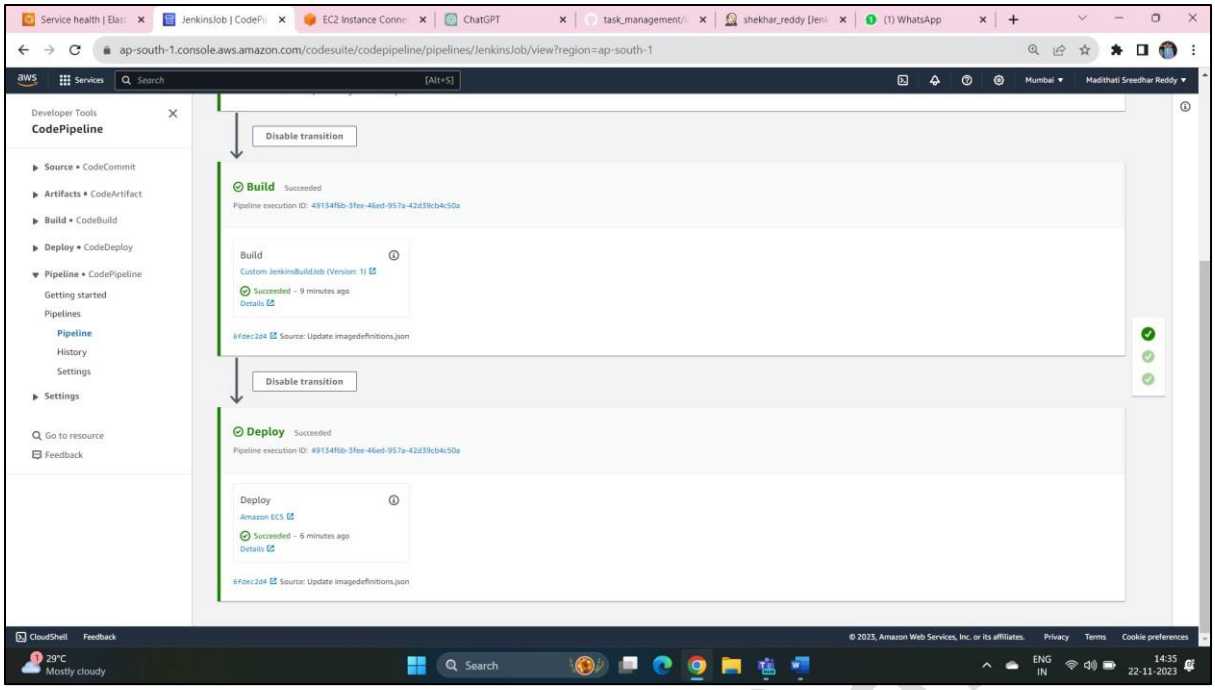
(Defined as per your ECS cluster).



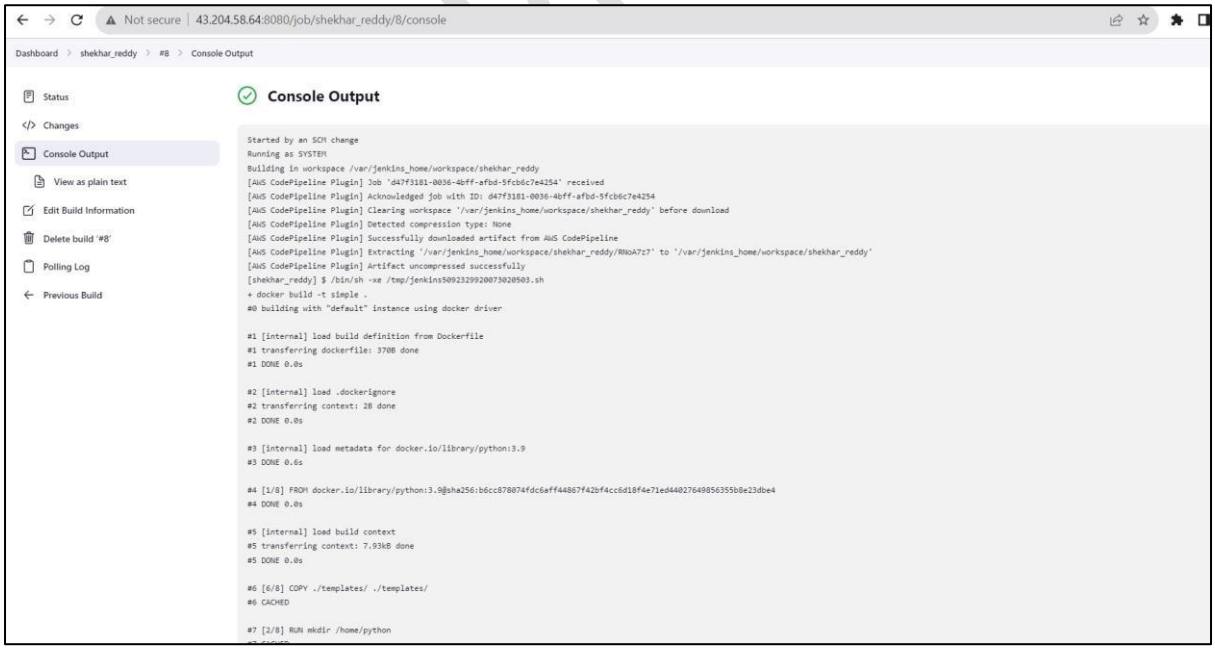
- Click on Next.
- Add additional stages if you have more steps in your pipeline (e.g., testing, approval).
- Review the pipeline configuration.
- Click the "Next" button to create the pipeline.

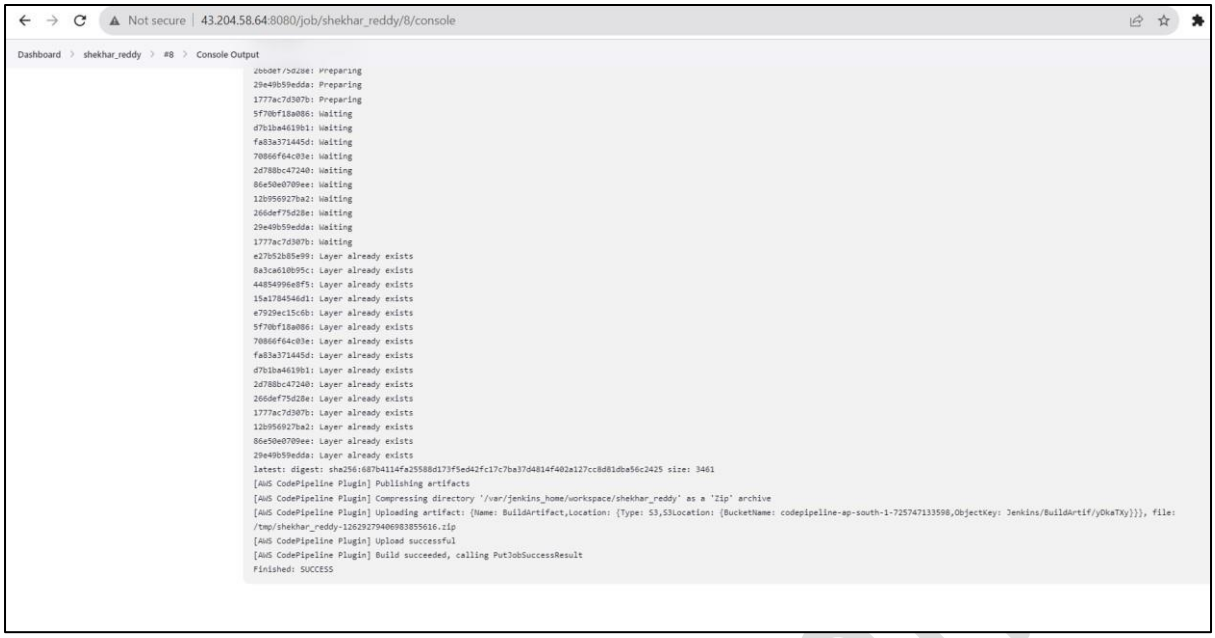
The completion of a pipeline depends on the successful execution of all stages and actions within the pipeline. If all stages and actions succeed, the pipeline execution is considered complete. If any stage or action fails, the pipeline execution may be marked as failed. The successfully executed code pipeline looks like this.



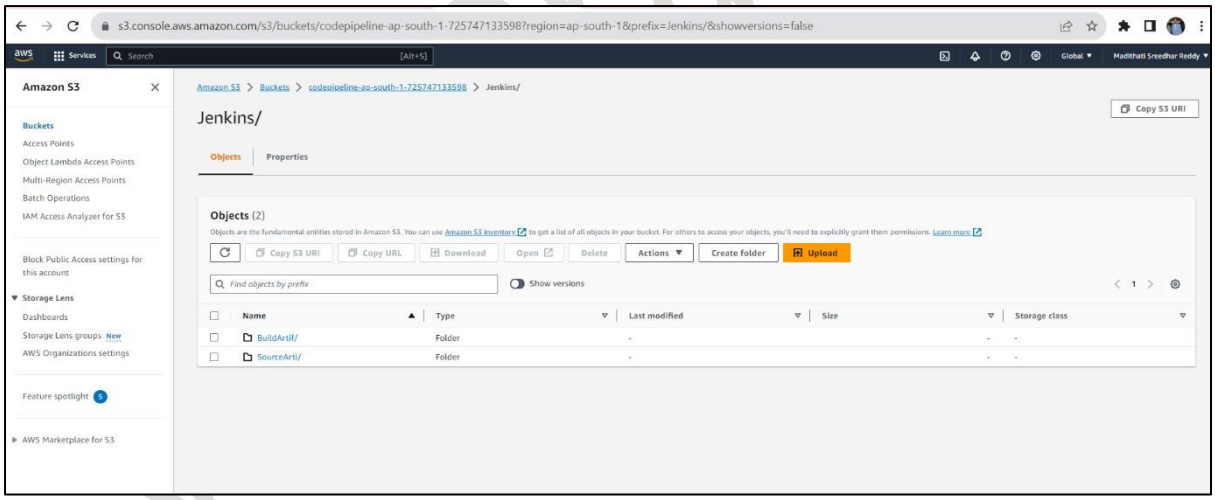


- In the Jenkins Server that is Configured in the AWS CodePipeline see the Console Output
- The main purpose is that This will take the SourceArtifact as Input and run the JenkinsJob that will in return produce the BuildArtifact that is stored in the S3 Bucket

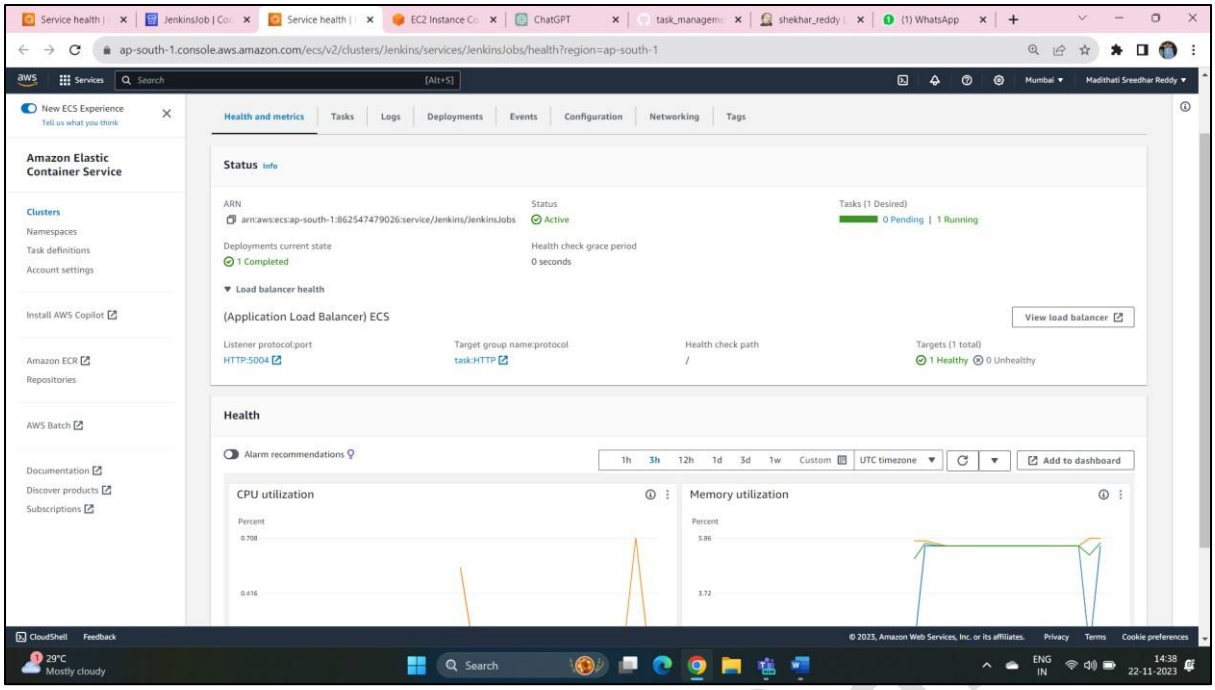




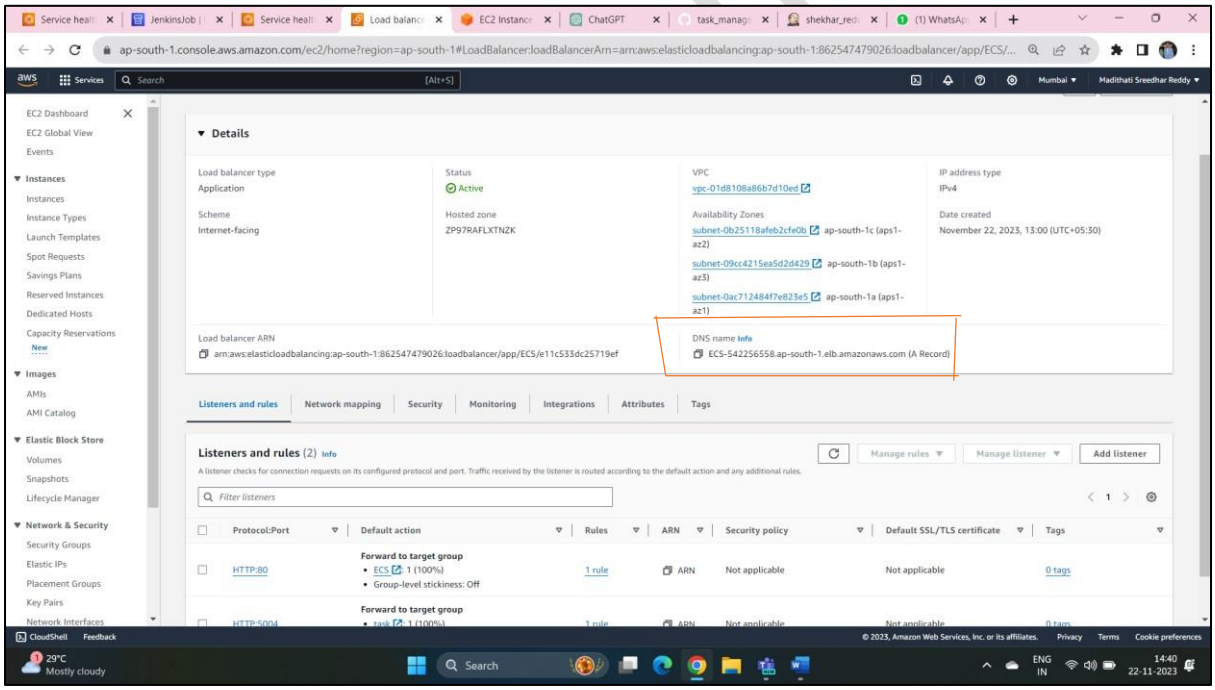
- If your application produces output files or data within the container, you might want to configure a mechanism to retrieve them. This could involve using shared volumes, S3, or other storage solutions.



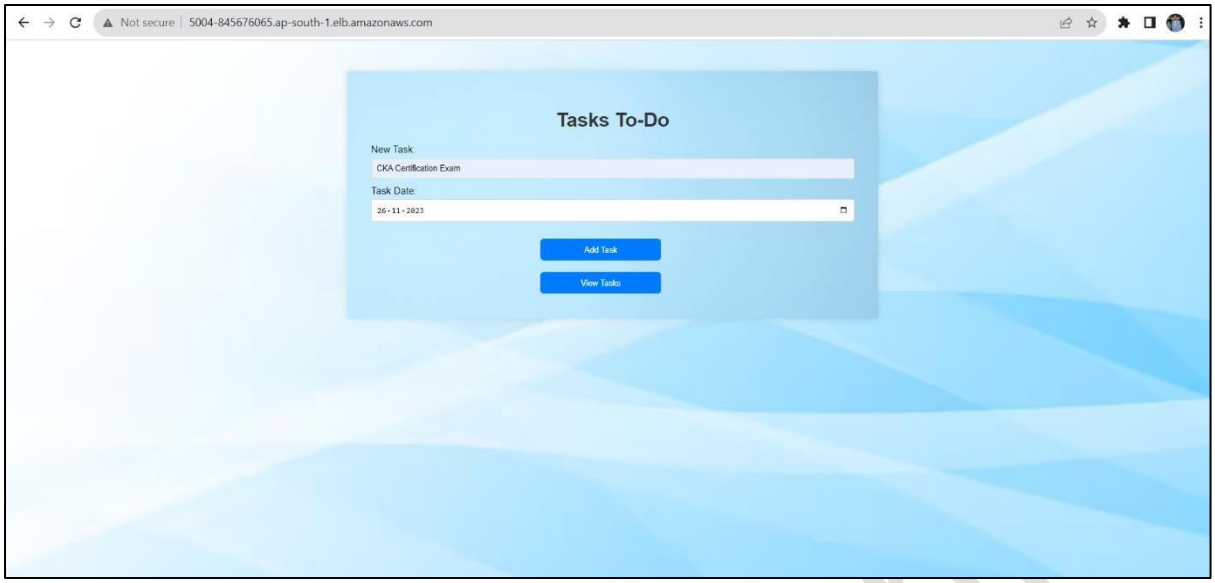
- These BuildArtifact is used by the Deploy Stage i.e, ECS for deploying the application
- If all went right the cluster UI will look Like this



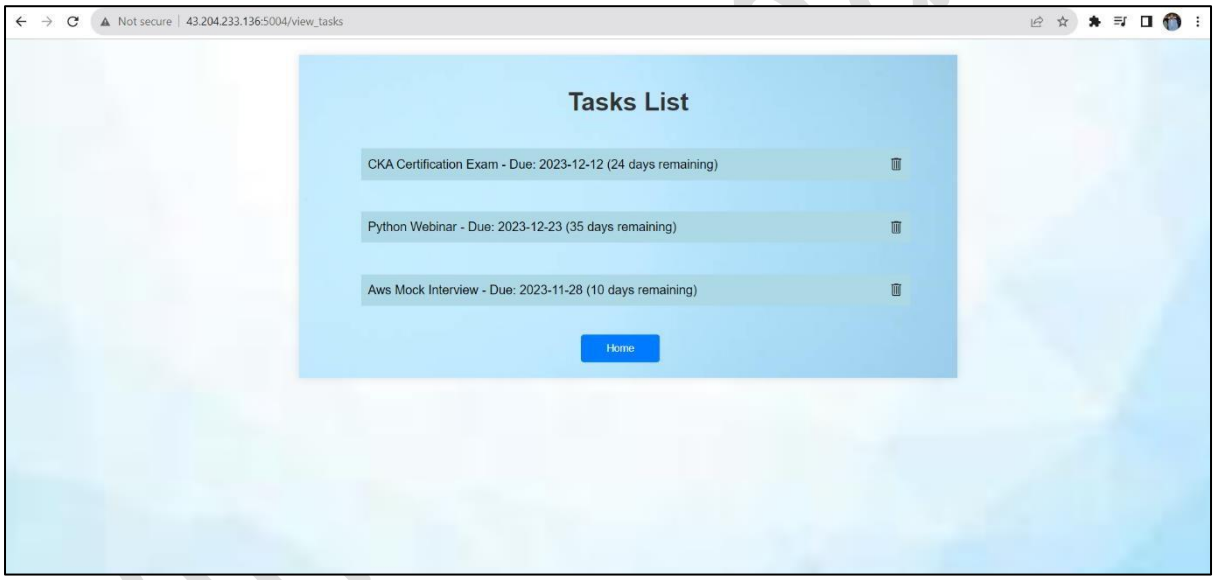
- Click on View Load Balancer.
- Then Copy the DNS Name of it and paste it in your browser



- We can access our Web application using DNS name of the Load balancer.
- The web UI will Look Like below.
- Now add Some Tasks as per your Requirement.



- Click on View Tasks that will look as below.



Define deployment actions, including application parameterization.

In Amazon Elastic Container Service (ECS), deployment actions refer to the steps and processes involved in releasing and updating containerized applications. ECS is a fully managed container orchestration service that simplifies the deployment, management, and scaling of Docker containers on Amazon EC2 instances or AWS Fargate.

Here are the key components and concepts related to deployment actions and application parameterization in ECS:

1. Task Definition:

- A task definition is a blueprint for your application. It defines various parameters, such as which Docker image to use, how much CPU and memory to allocate, networking information, and more.

- When updating an application, you can revise the task definition to include new container images or modify other settings.

2. Service:

- A service in ECS allows you to run and maintain a specified number of instances of a task definition simultaneously.

- During deployment, ECS manages the desired count of tasks, ensuring that the specified number of tasks are running.

3. Deployment Types:

- ECS supports two deployment types: rolling updates and blue/green deployments.

- **Rolling Updates**: ECS gradually replaces instances of the previous version of your task with the new version, minimizing downtime.

- **Blue/Green Deployments**: This involves running two separate environments (blue and green) and switching traffic from one to the other when deploying updates.

4. Application Load Balancer (ALB) or Network Load Balancer (NLB):

- Load balancers help distribute incoming traffic across multiple tasks in your ECS service. This is crucial for achieving high availability and seamless updates.

- ALBs and NLBs can be used to route traffic to different versions of your application during a blue/green deployment.

5. Application Parameterization:

- Parameterization involves configuring your application with external parameters, allowing you to customize its behavior without modifying the application code.

- In ECS, you can use environment variables and task definition parameters for parameterization.

- For example, you might use environment variables to specify configuration settings or connection strings dynamically.

6. ECS CLI and AWS Management Console:

- The ECS CLI and AWS Management Console provide tools for managing ECS tasks, services, and deployments.

- You can use these interfaces to update task definitions, change desired task counts, and monitor the status of deployments.

7. Continuous Integration/Continuous Deployment (CI/CD):

- Many organizations integrate ECS deployments into their CI/CD pipelines. Tools like AWS CodePipeline and AWS CodeBuild can be used to automate the building and deployment of containerized applications on ECS.

Application parameterization in AWS ECS refers to the practice of configuring and managing application-specific parameters or settings dynamically, allowing for flexibility and customization without modifying the application code. This can include environment variables, configuration files, or other mechanisms to adjust the behaviour of the application.

- Define environment variables in your ECS task definition.
- Store Sensitive Data:
  For sensitive information like API keys or database passwords, consider using AWS Secrets Manager.
  Store secrets in Secrets Manager and reference them in your ECS task definition.
- Store Configurations in an S3 Bucket:
  Store configuration files in an S3 bucket, and download them at runtime.
- Service Discovery:
  Use service discovery for dynamic port mapping.
  Automatically register services with a service discovery namespace.

Sub Task 5:

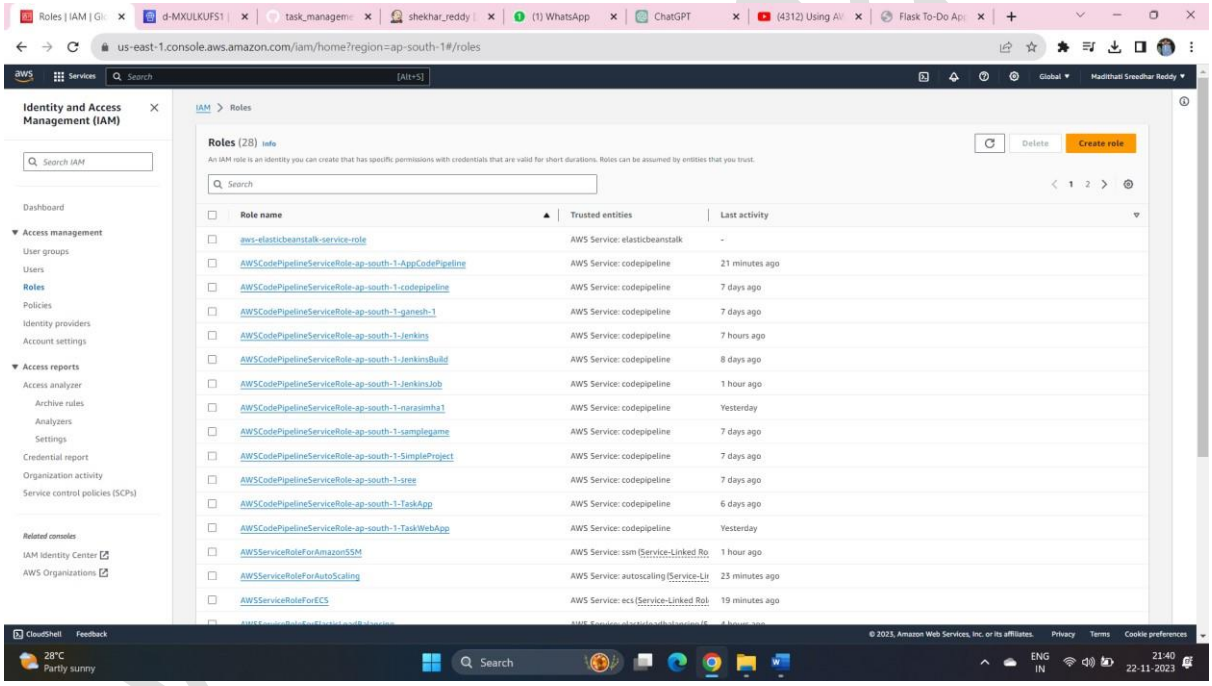
Deploy the application to AWS ECS or EKS using AWS CodeDeploy.

AWS CodeDeploy is a fully managed deployment service that automates software deployments to a variety of compute services, including Amazon EC2 instances, AWS Lambda functions, and instances running on-premises. It allows you to consistently deploy your applications across your development, test, and production environments.

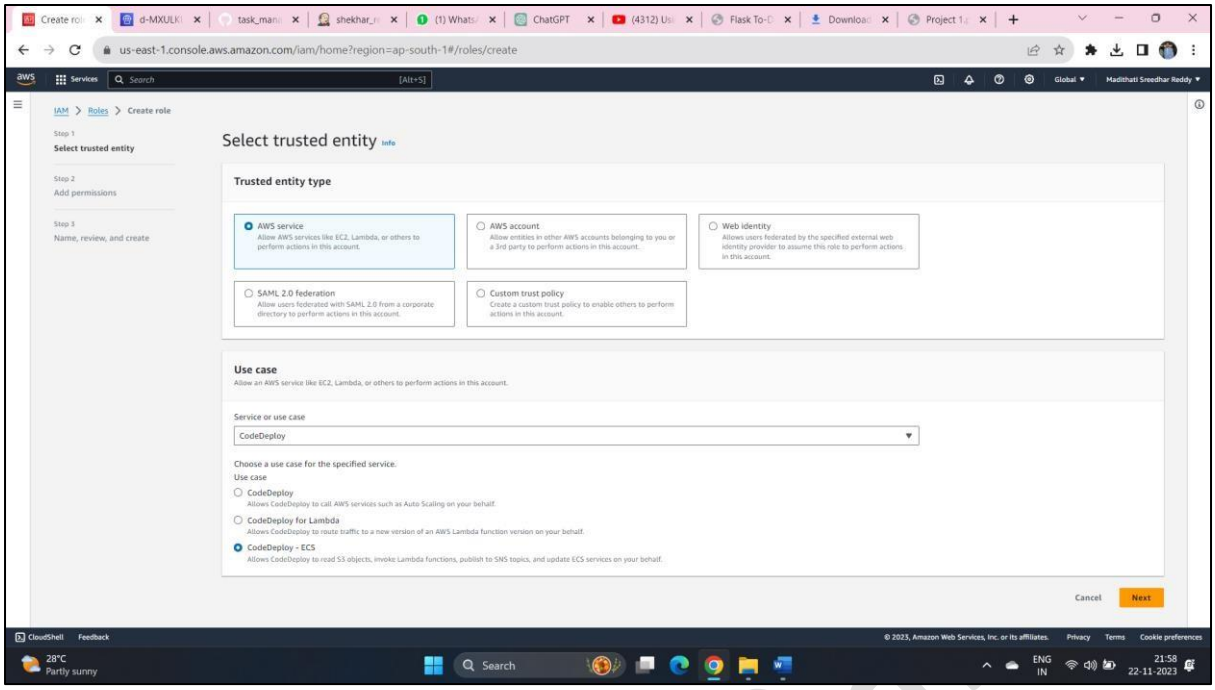
Creation of the IAM Role

Creating a role in AWS involves using AWS Identity and Access Management (IAM), which allows you to manage access to AWS services and resources securely. Here are the general steps to create a role in AWS:

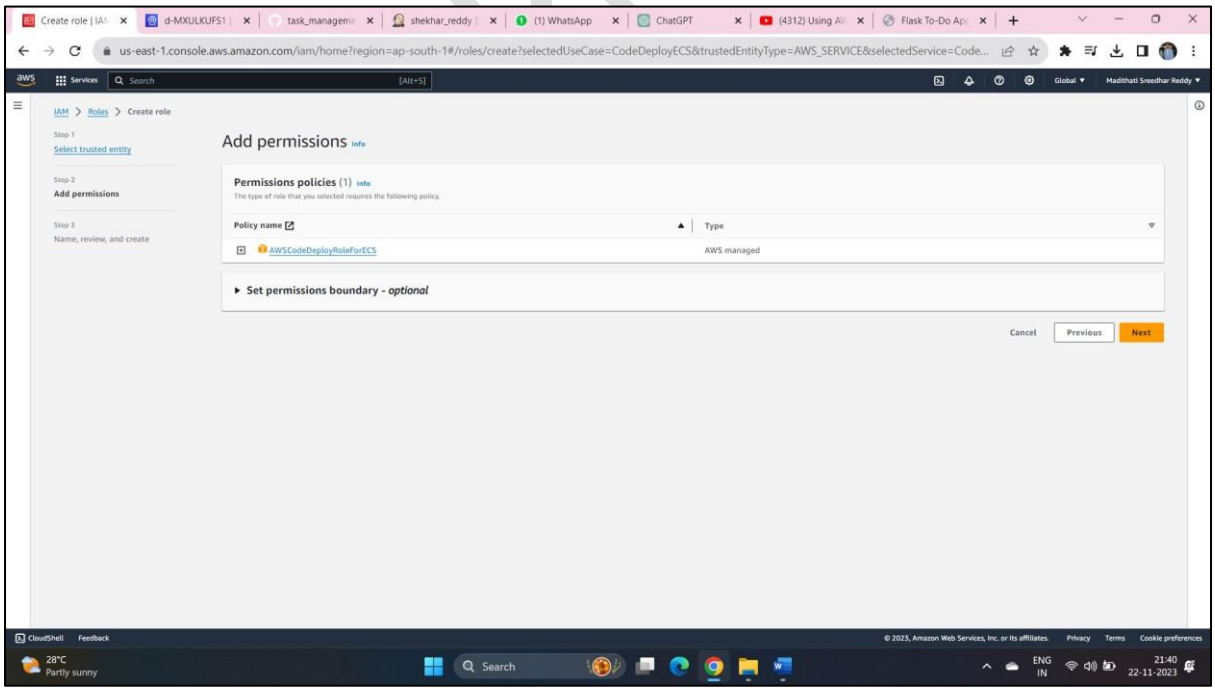
- Navigate to IAM:
  In the AWS Management Console, go to the IAM service. You can find it under the "Services" dropdown or by searching for "IAM." □ Select "Roles" in the IAM Dashboard:
  In the IAM dashboard, select "Roles" from the left navigation pane.



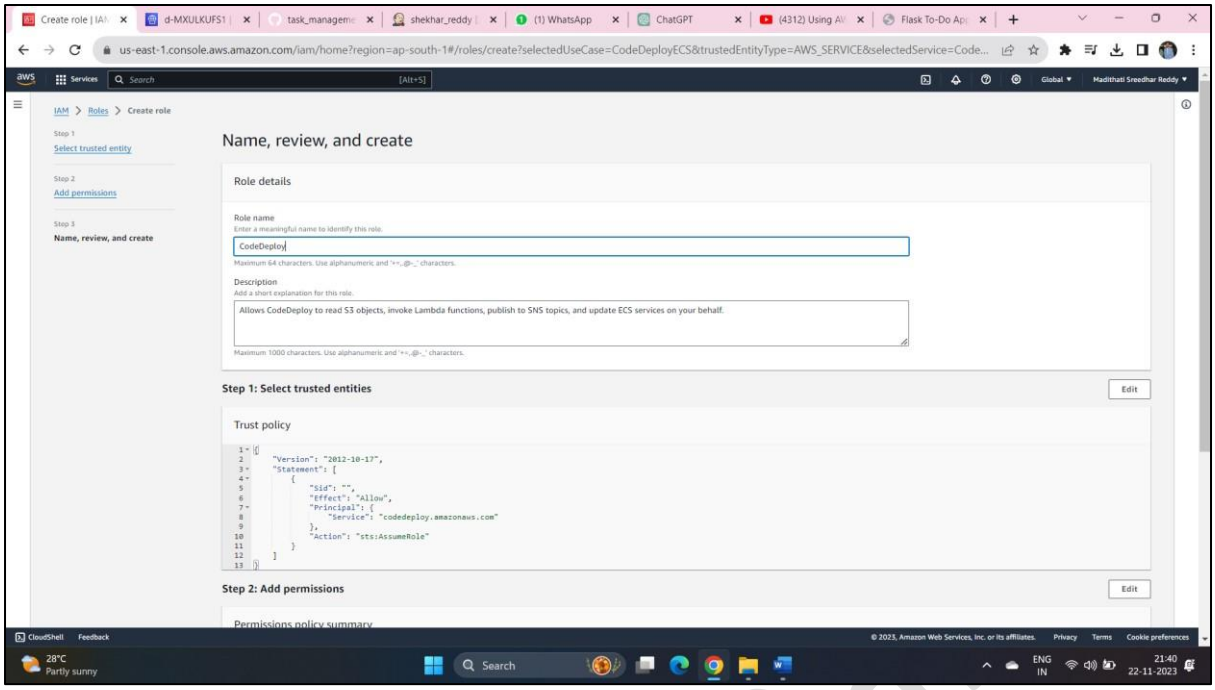
- Click "Create role":
- Click the "Create role" button to start the role creation process.



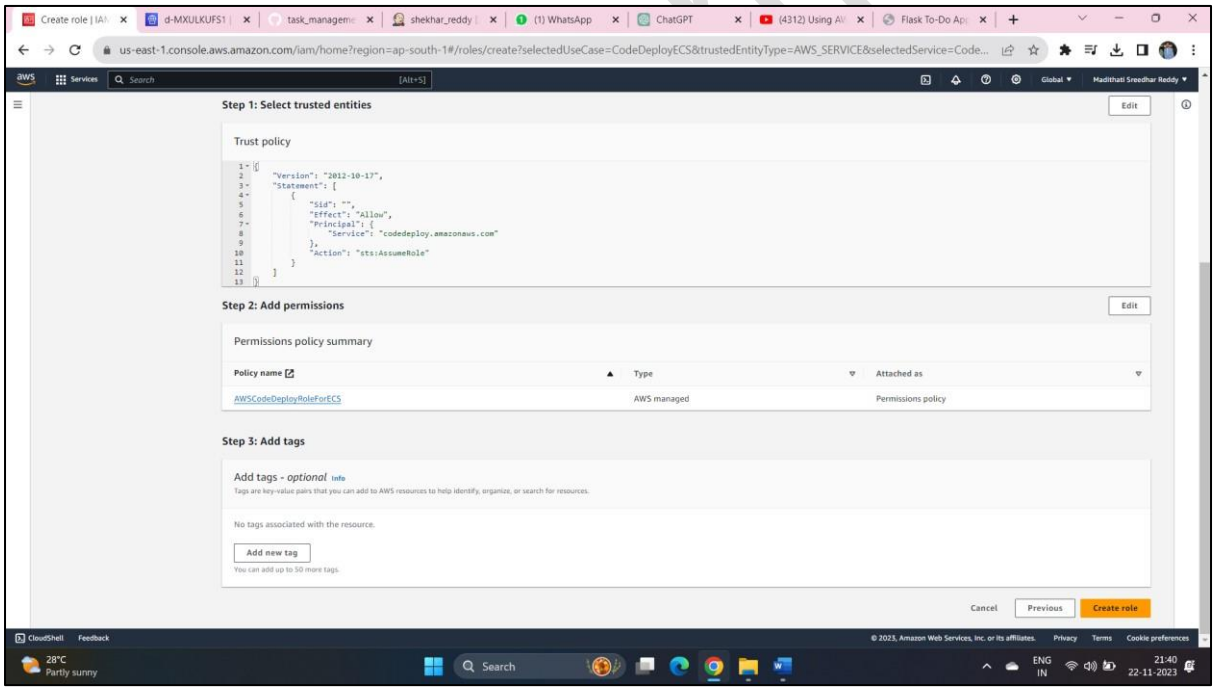
- Choose the use case as CodeDeploy
- It will reveal 3 options on which you need to select the CodeDeploy-ECS
- Click on Next



- Add Permissions as above and click on Next



- Give the Name for the role .



- Click on Create Role

Creation of Cluster, Task Definition file:

- Follow the Same procedure for creating the Cluster and the Task Definition File mentioned in the Sub Task -4

Creation of the ECS Service:

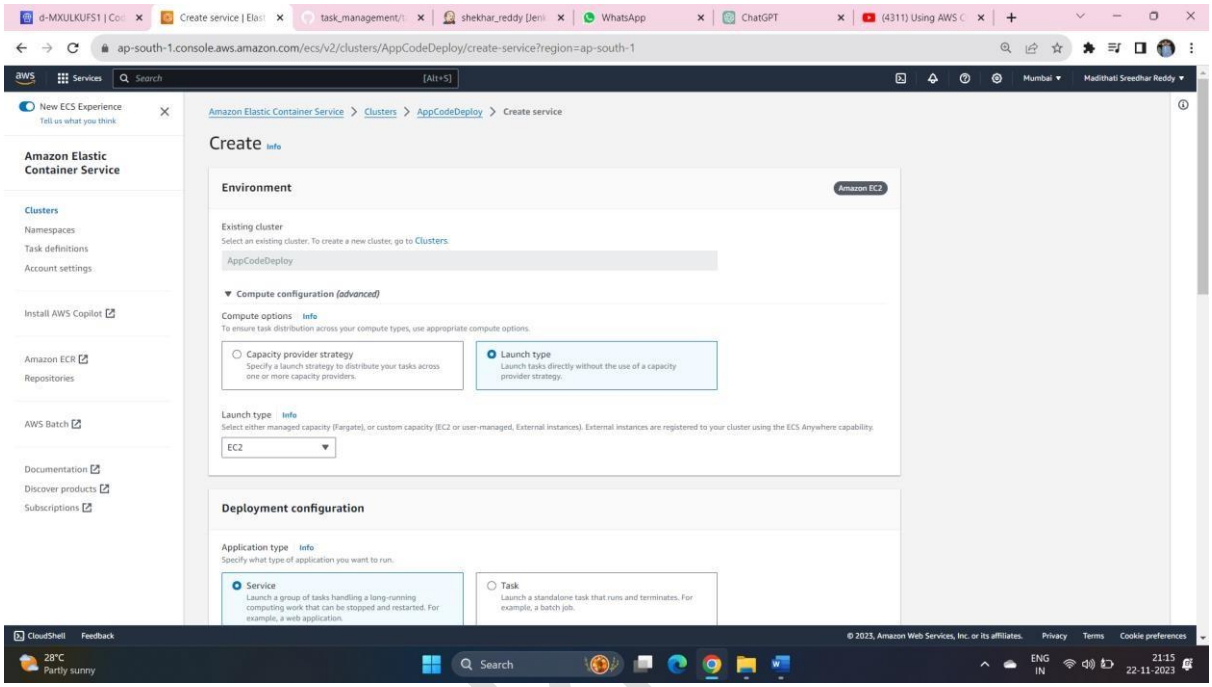
Navigate to ECS:

In the AWS Management Console, go to the ECS service. You can find it under the "Services" dropdown or by searching for "ECS." **Select your Cluster:**

In the ECS dashboard, select the ECS cluster where you want to create the service.

- **Click "Create" in the "Services" tab:**

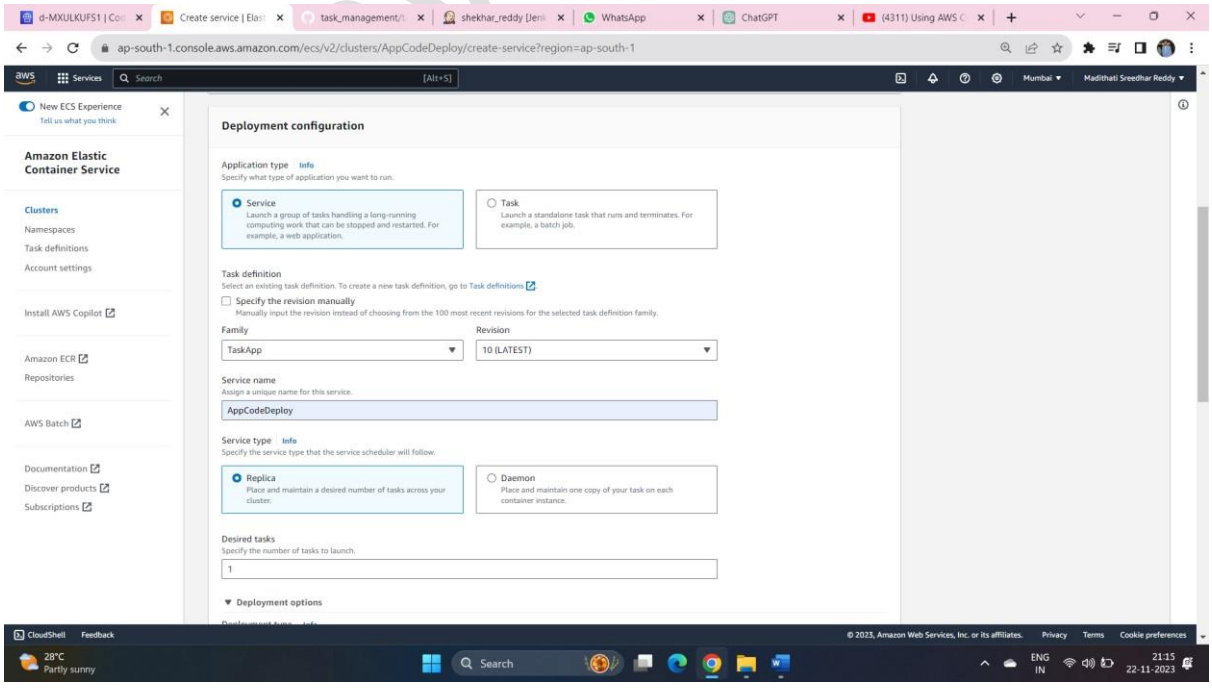
In the "Services" tab of your cluster, click the "Create" button.



- **Configure Service:**

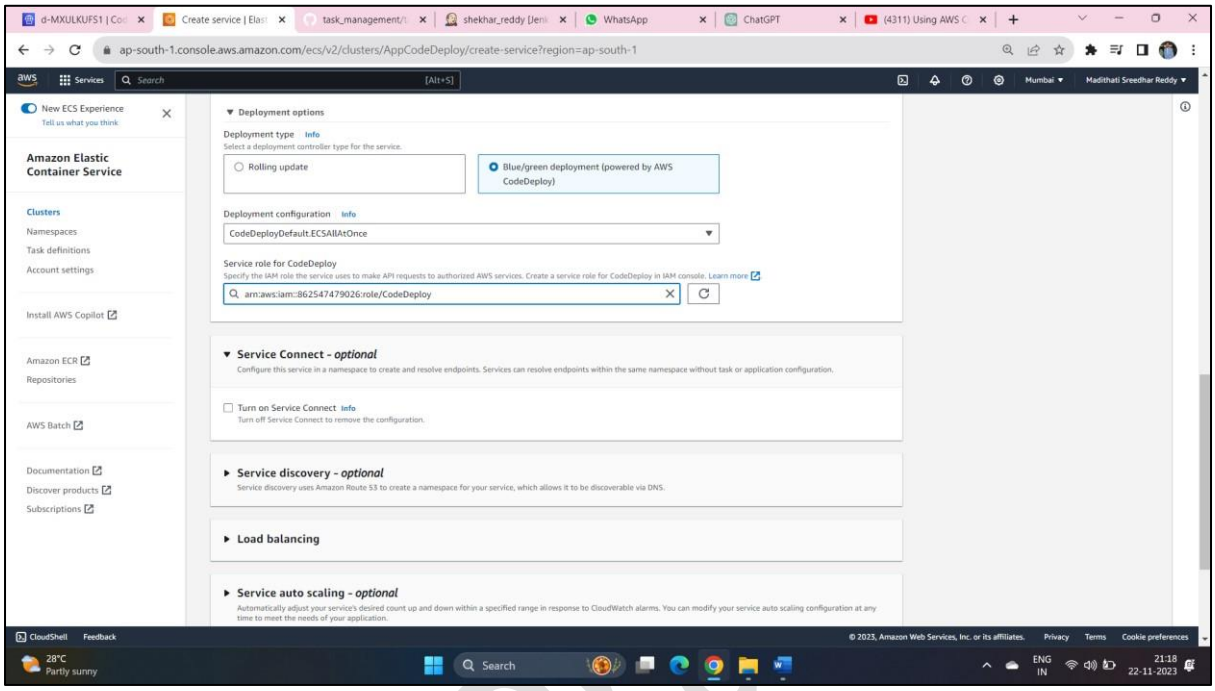
Launch type: Choose whether your tasks will run on EC2 instances or as AWS Fargate tasks.

Task Definition: Select the task definition that you want to run as part of the service.



Service name: Provide a unique name for your service**.**

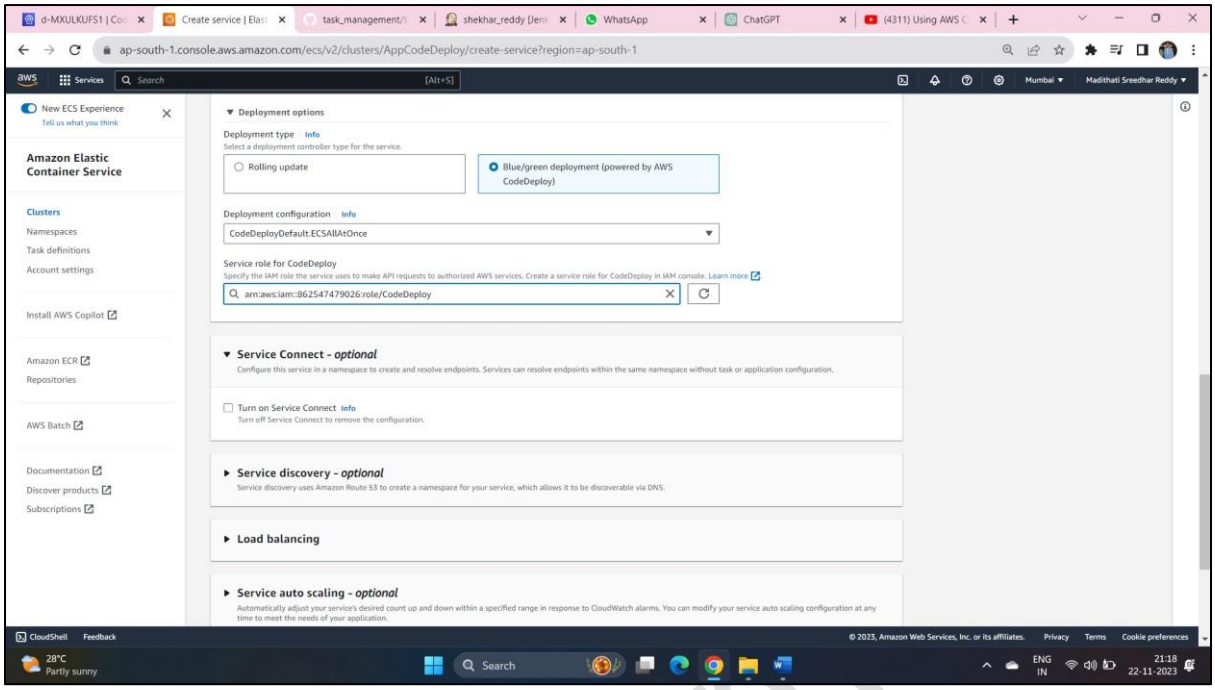
- **Number of tasks:** Specify the number of tasks (containers) to run in the service**.**
- **Cluster VPC:** Choose the VPC in which to launch your tasks.
- **Subnets:** Select the subnets in which to place your tasks**.** **Security Groups:** Choose security groups for your tasks.



- **Set Deployment Configuration:**

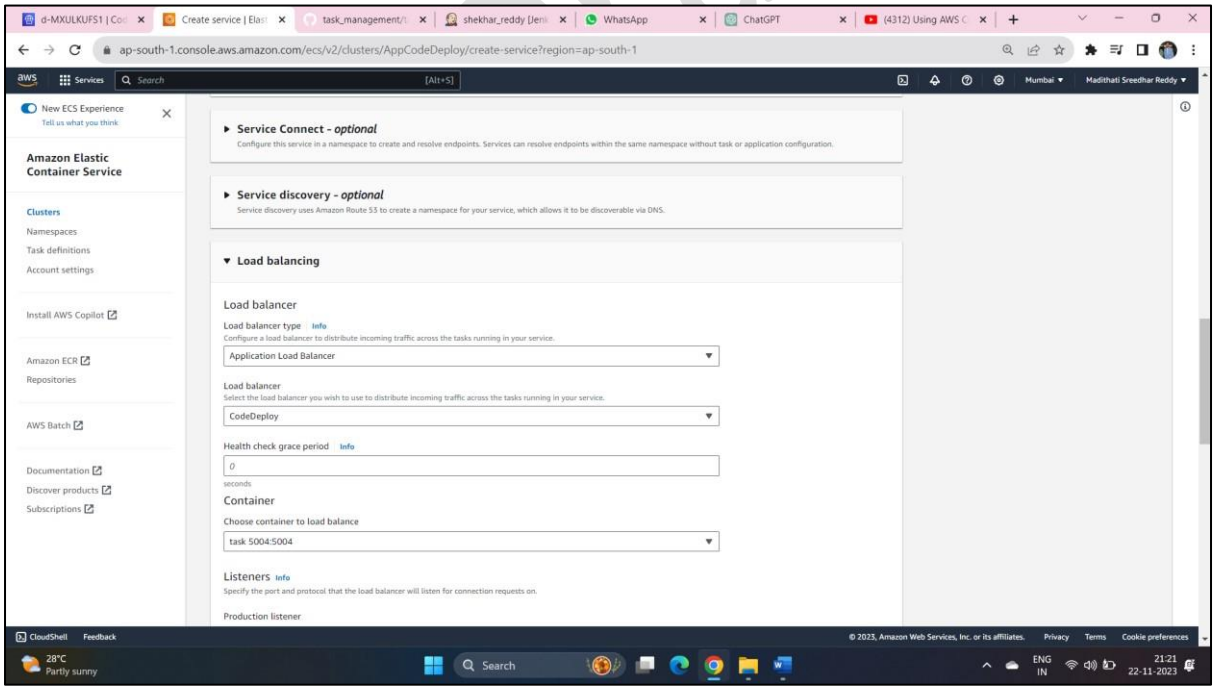
Choose the deployment configuration that determines how your service tasks are deployed. Select the option as blue/green deployments.

-
- Select the Service role created earlier

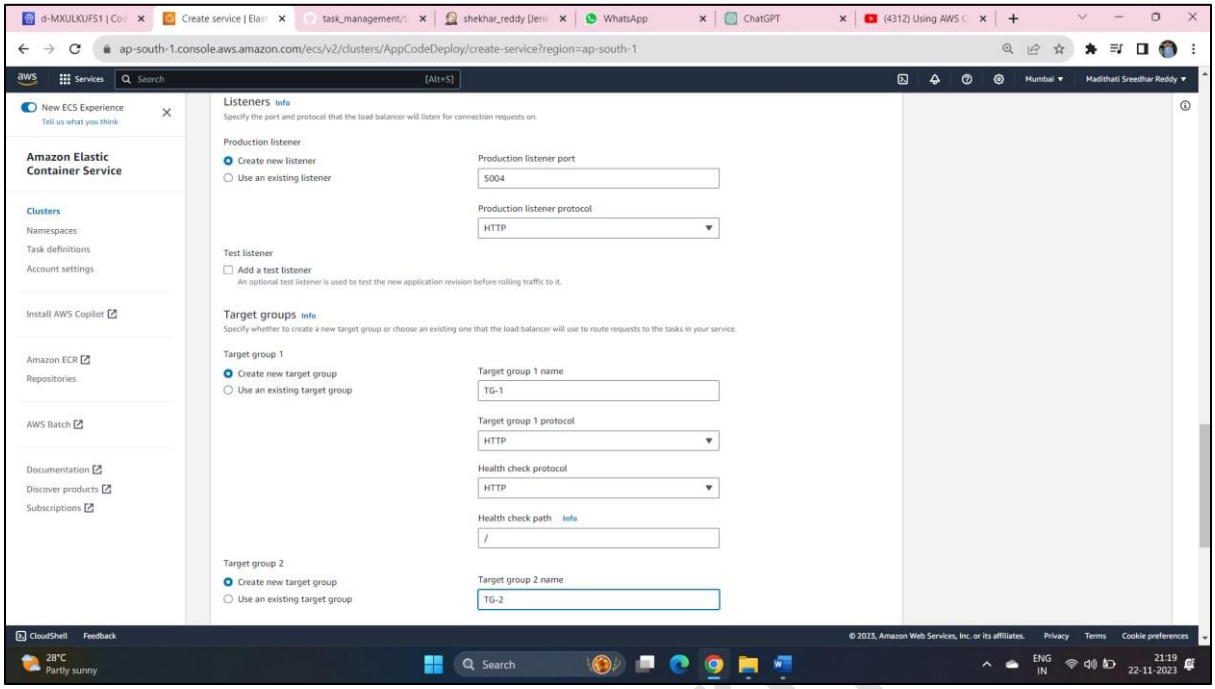


Load Balancer: If you want to distribute traffic to your tasks, configure the load balancer settings.

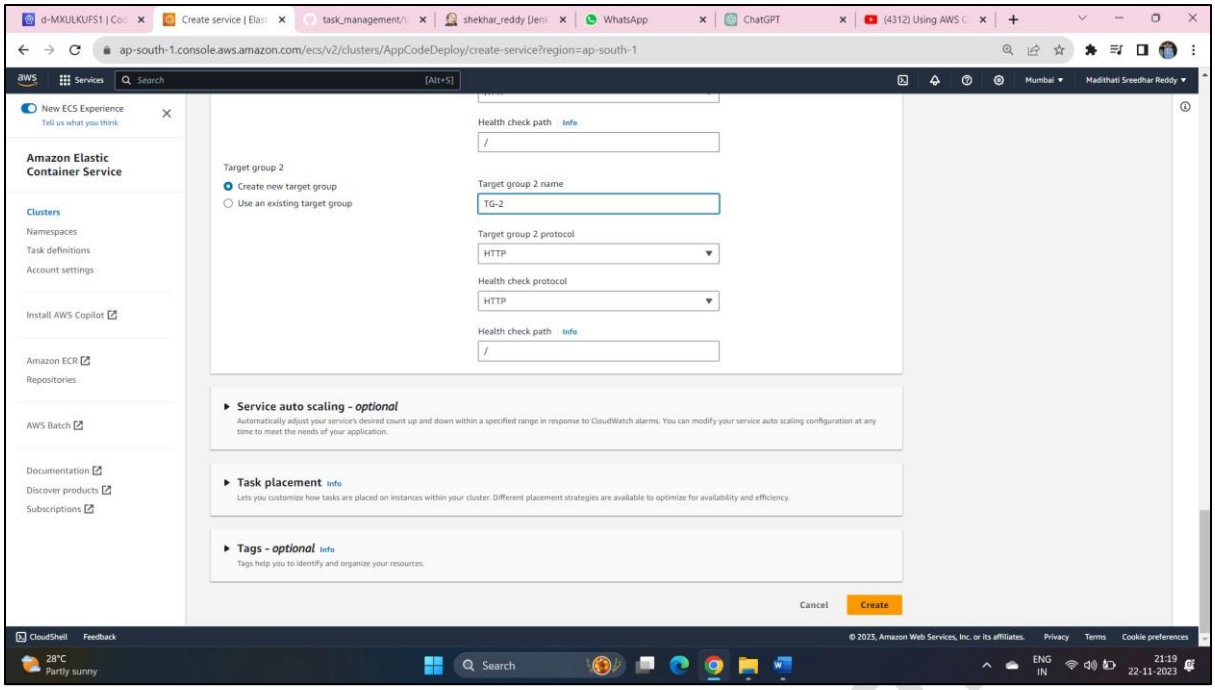
- Give the configuration as below.



-
- Create the Listeners as below



- Create new target groups or Choose the Target Groups from the existing ones configured

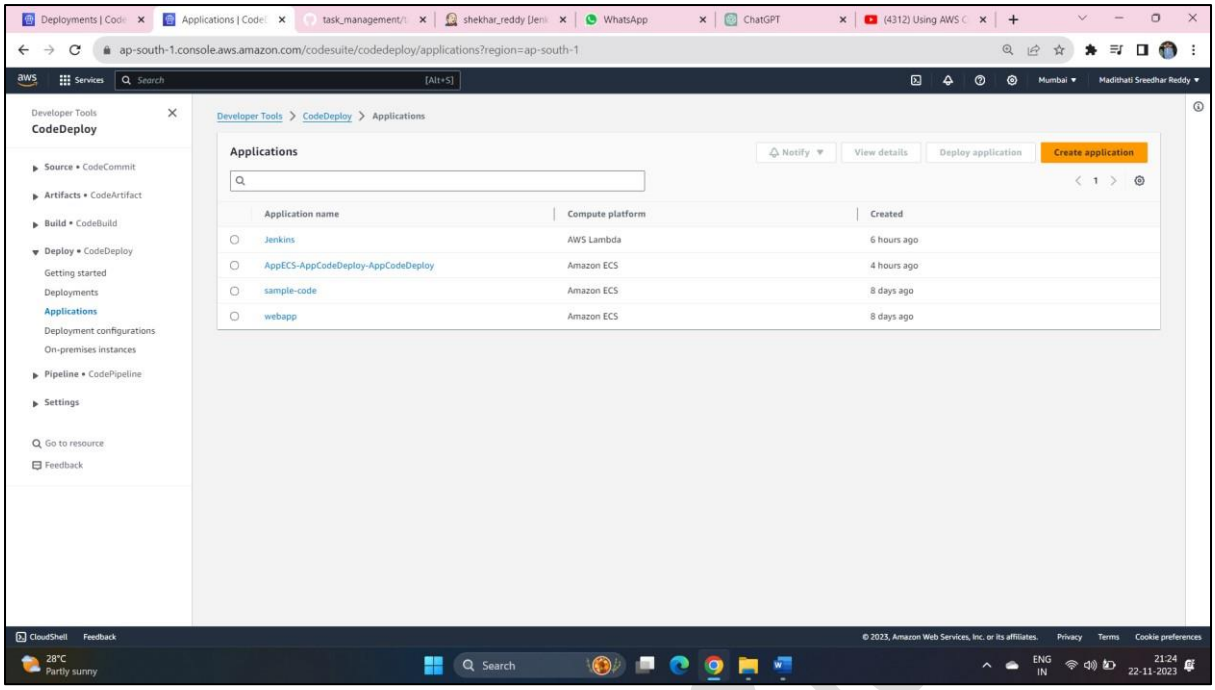


- Then Review the configurations to ensure they are correct.
- Click the "Create Service" button to create the ECS service

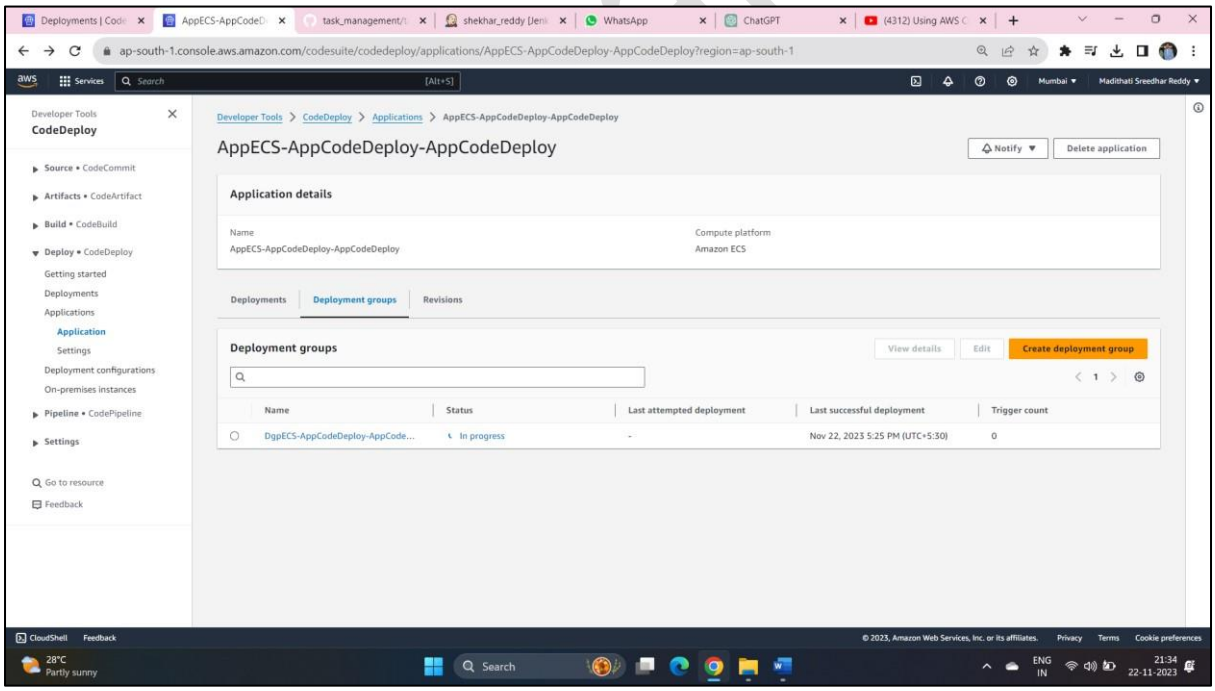
Configuring the Code Deploy:

- Navigate to AWS CodeDeploy:
  In the AWS Management Console, go to the CodeDeploy service. You can find it under the "Services" dropdown or by searching for "CodeDeploy."
- Under Applications You can see the application created with the name of our Cluster Created earlier

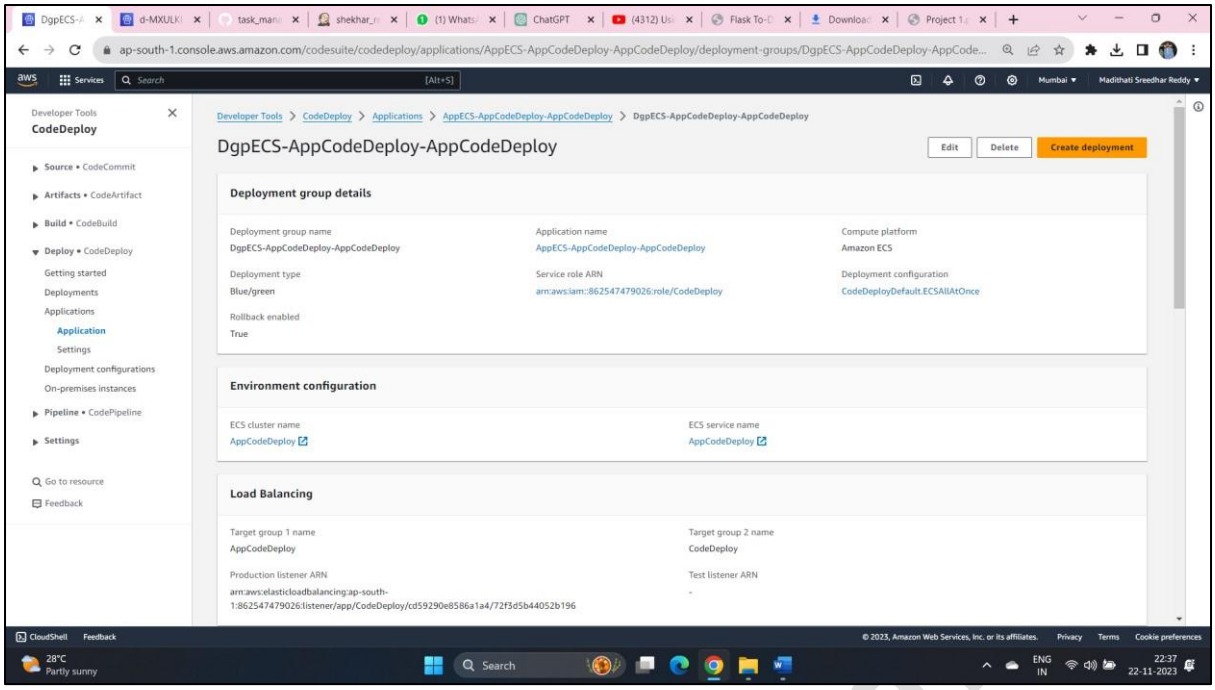
- Click on it.



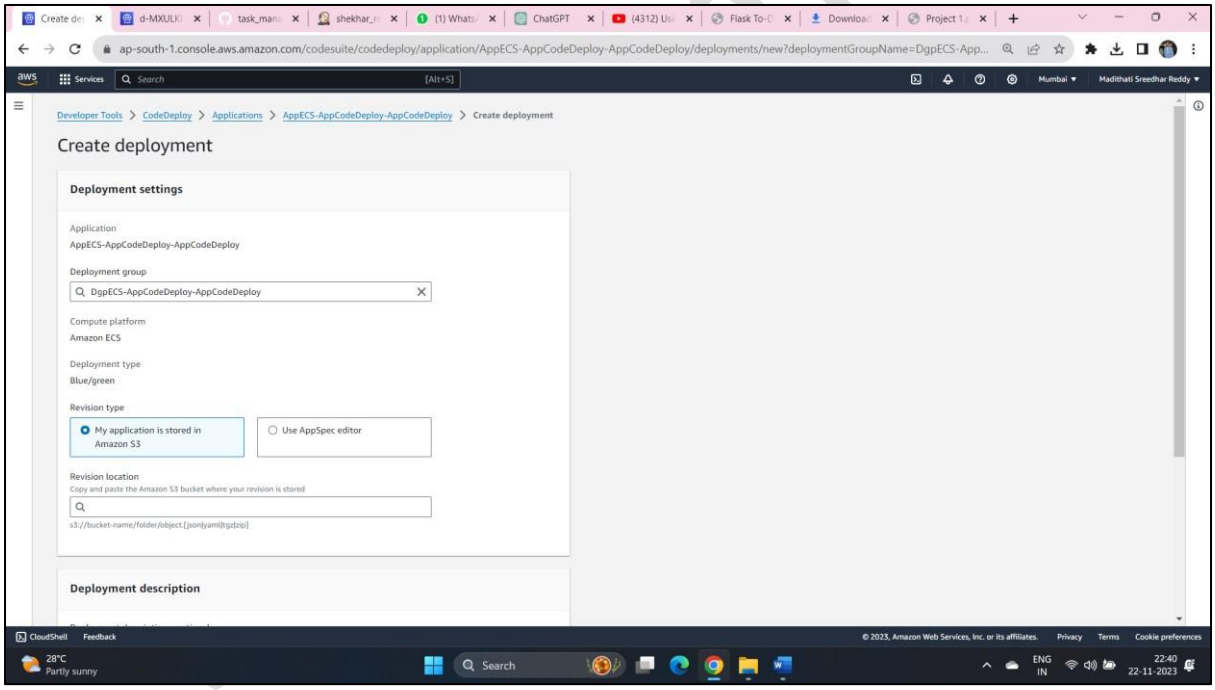
- You can see an Deployment Already Created. Click on that.



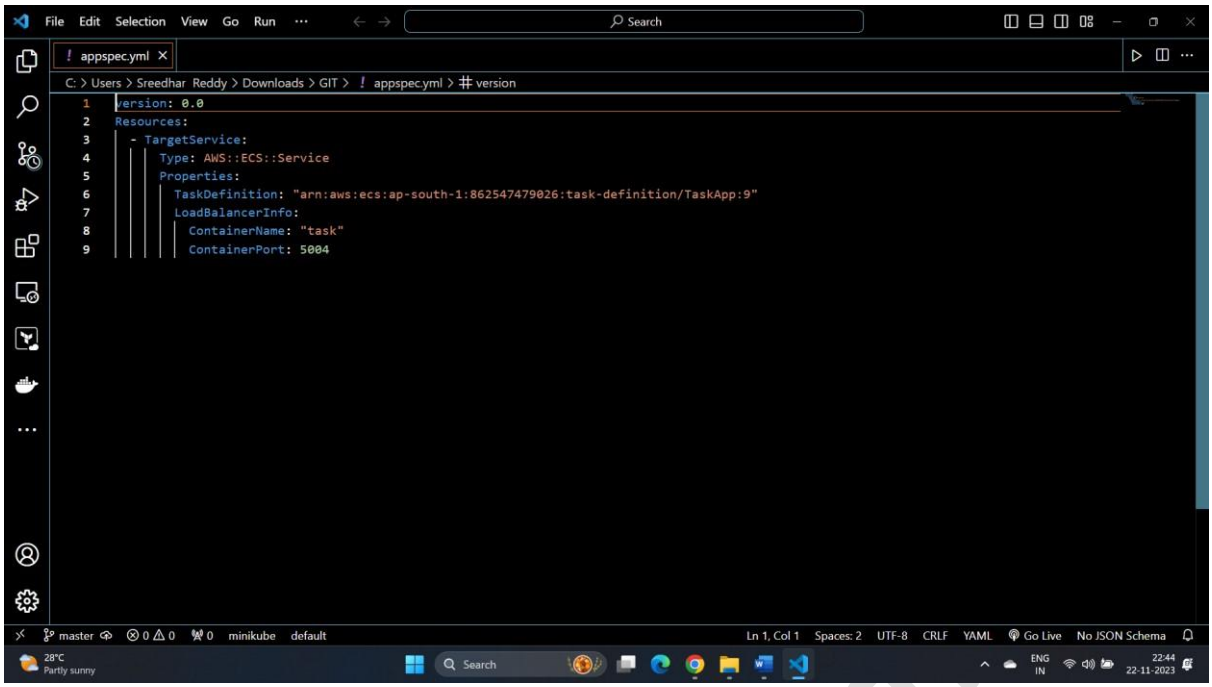
- Then Click on the Create Deployment Option



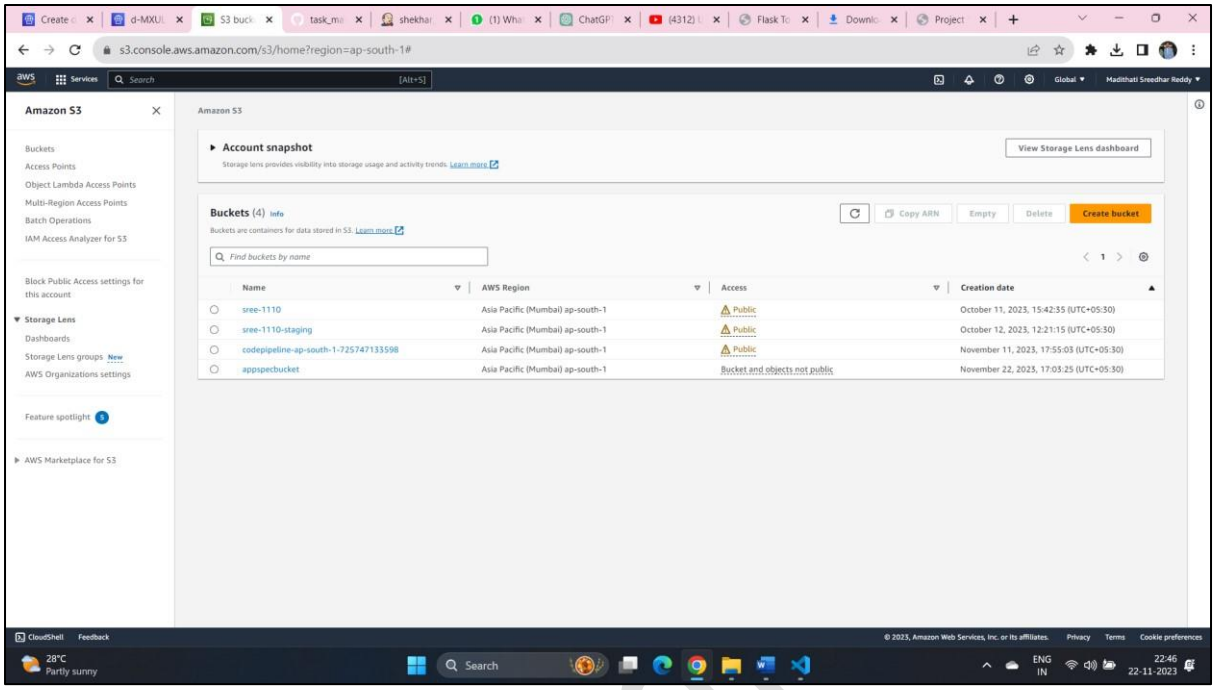
- Under it Choose the Deployment group



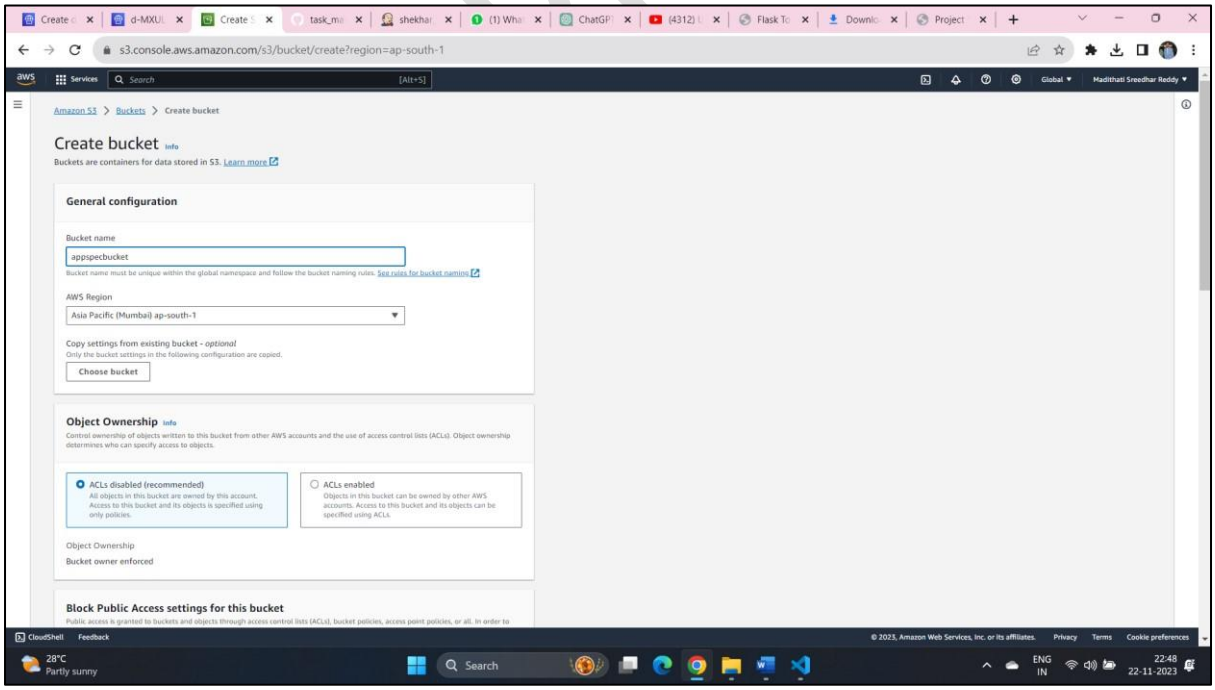
- For the AppSpec file you need to do the Following steps initially Create the file with the content as below.



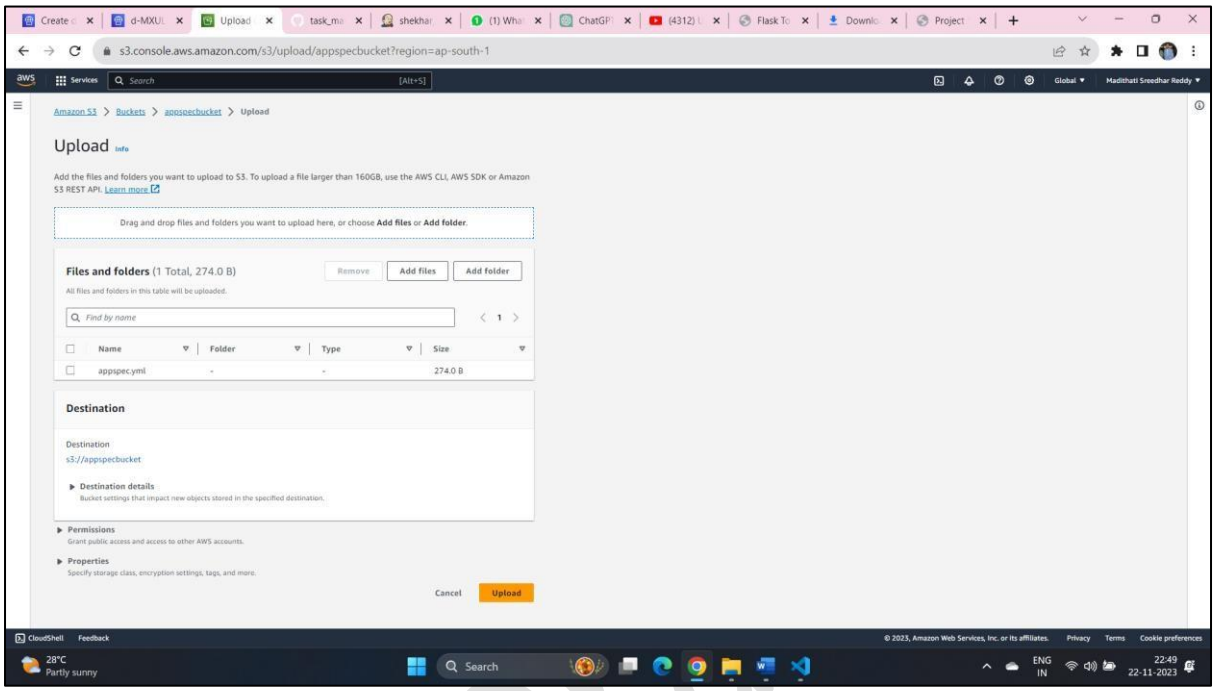
- Create a new Bucket with Unique name in Amazon S3



- Click on Create Bucket
  - Give an Unique name and click create

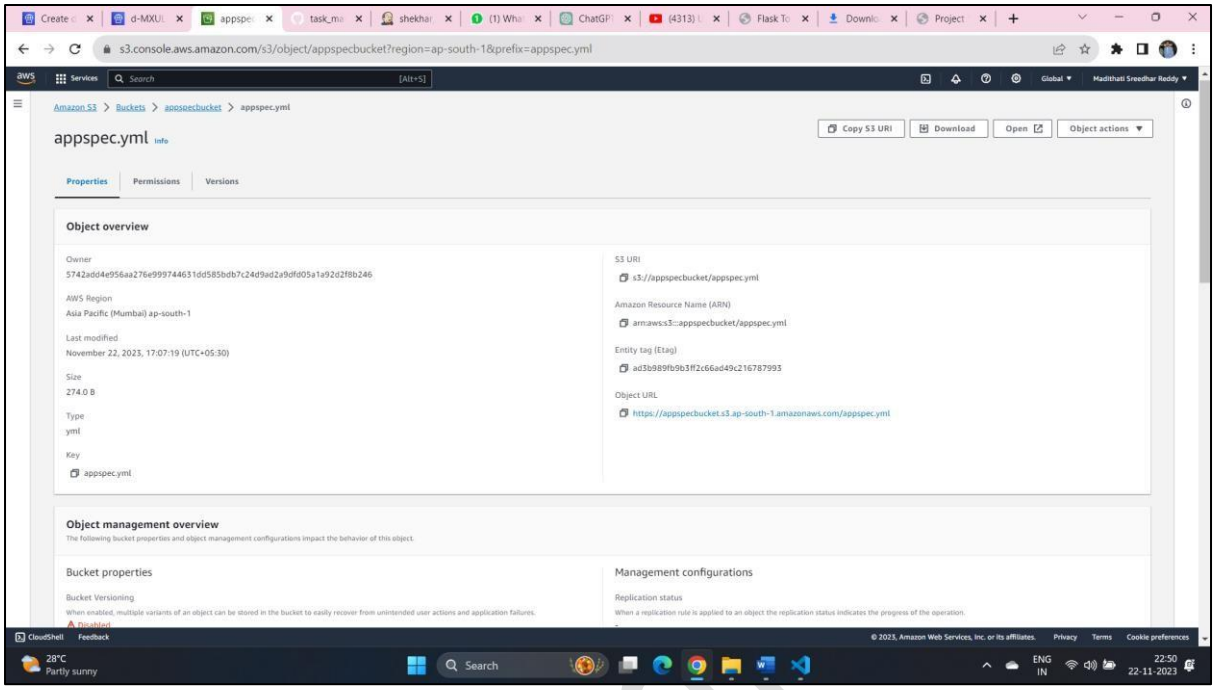


Upload the file created in the local to it.



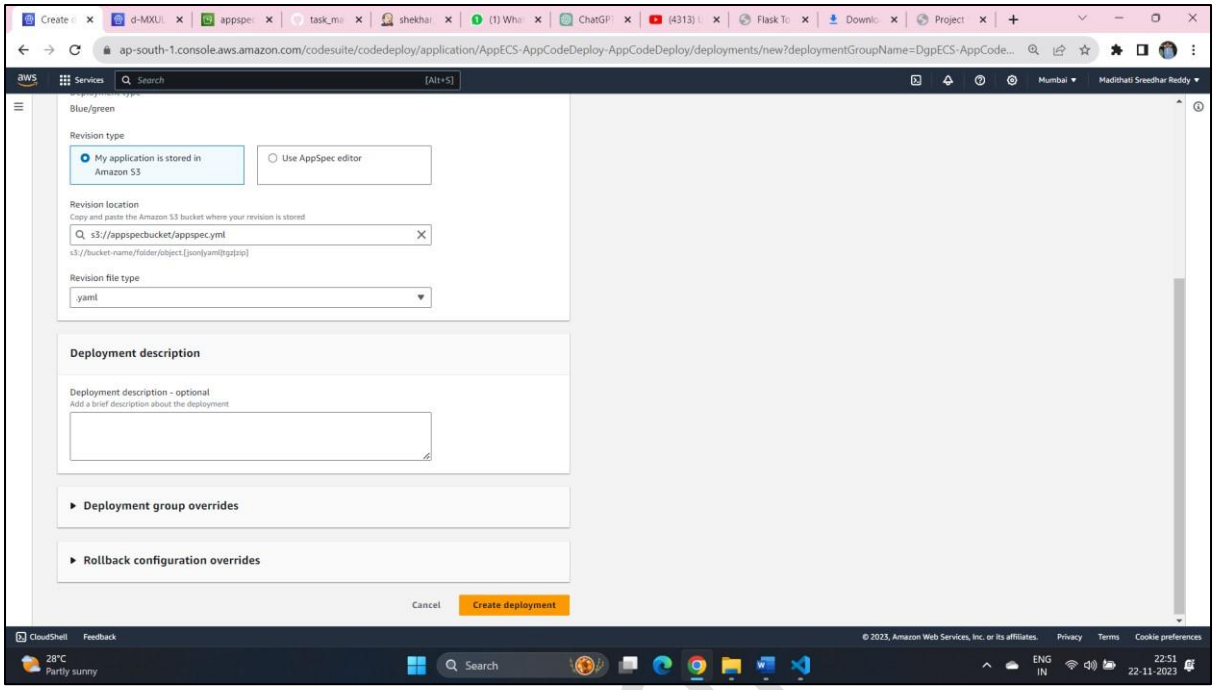
- Once Uploaded S3 URI of it

-

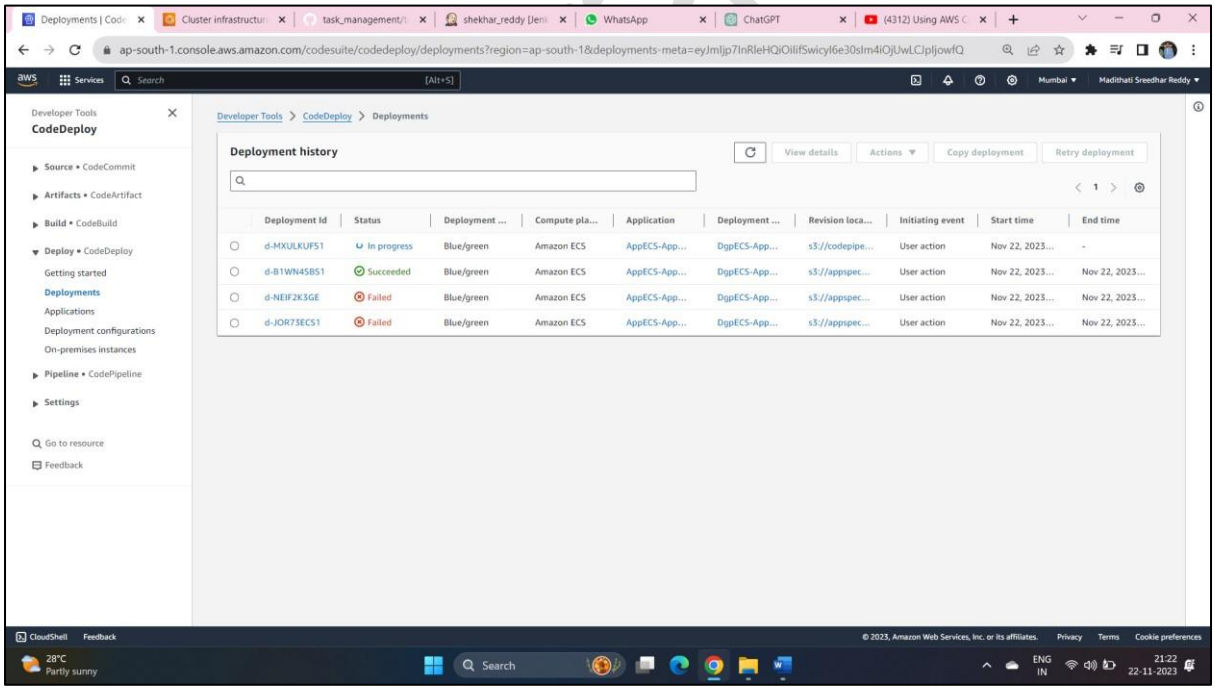


- Paste it under the Revision location

  Choose the Revision File Type as .yaml Click on Create Deployment

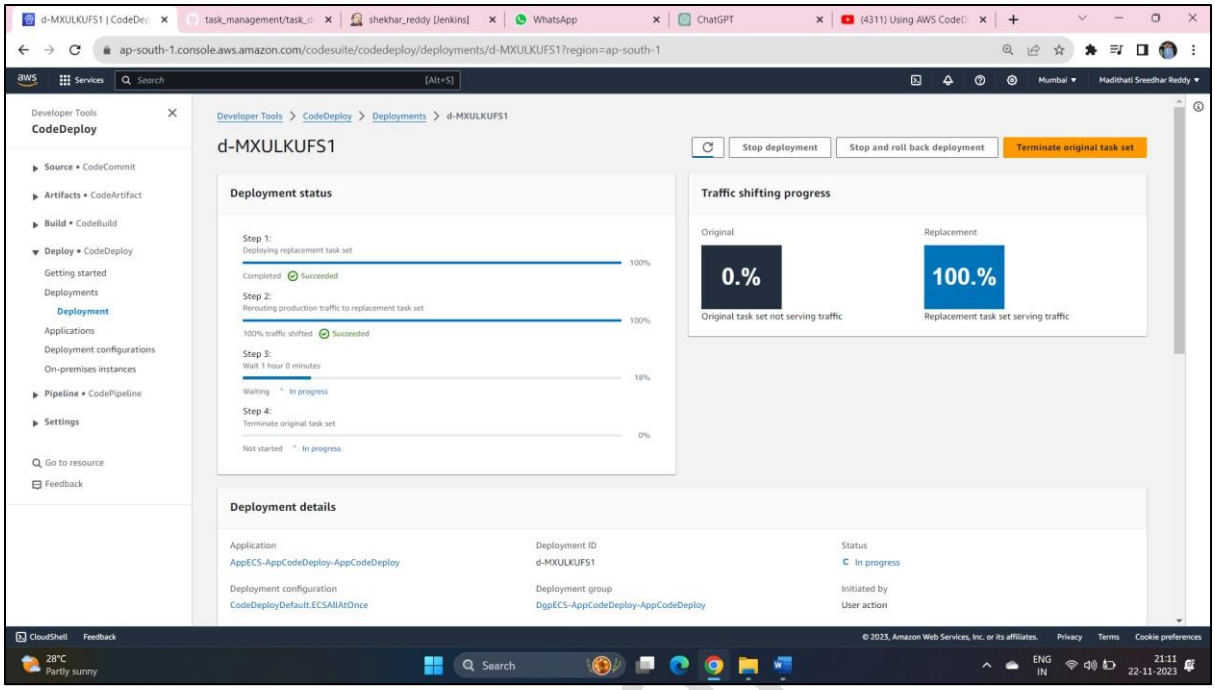


- Under Deployments Section The one created will be in progress state

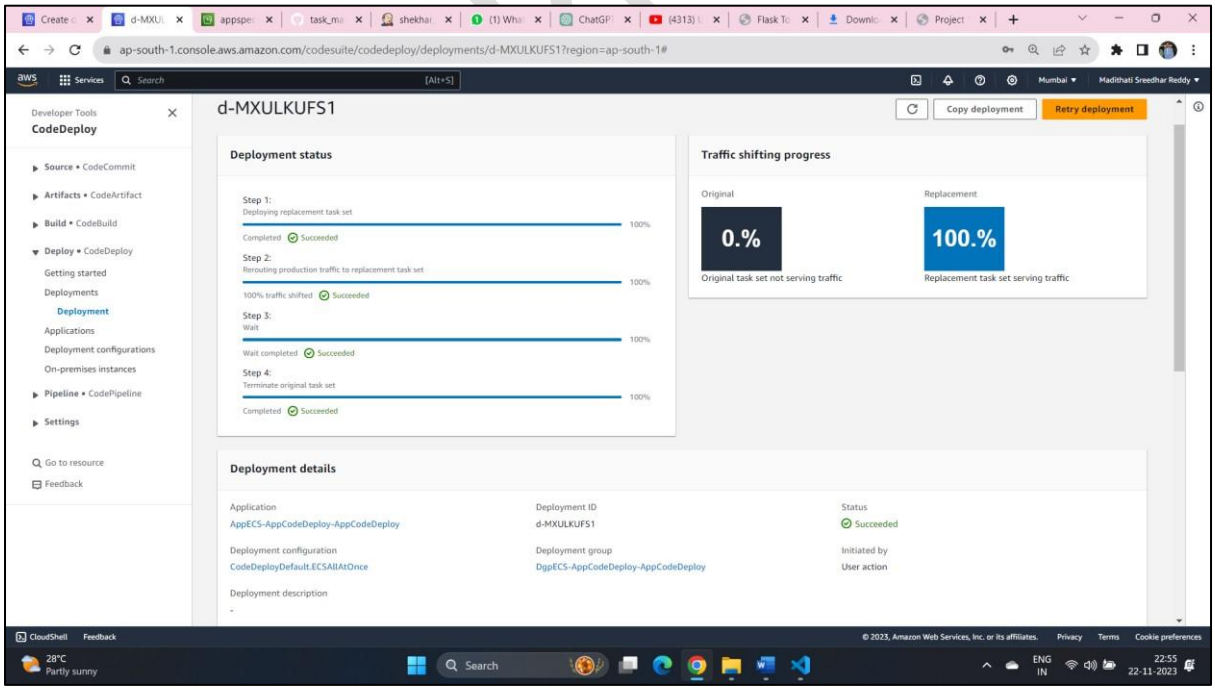


Click on our Deployment.

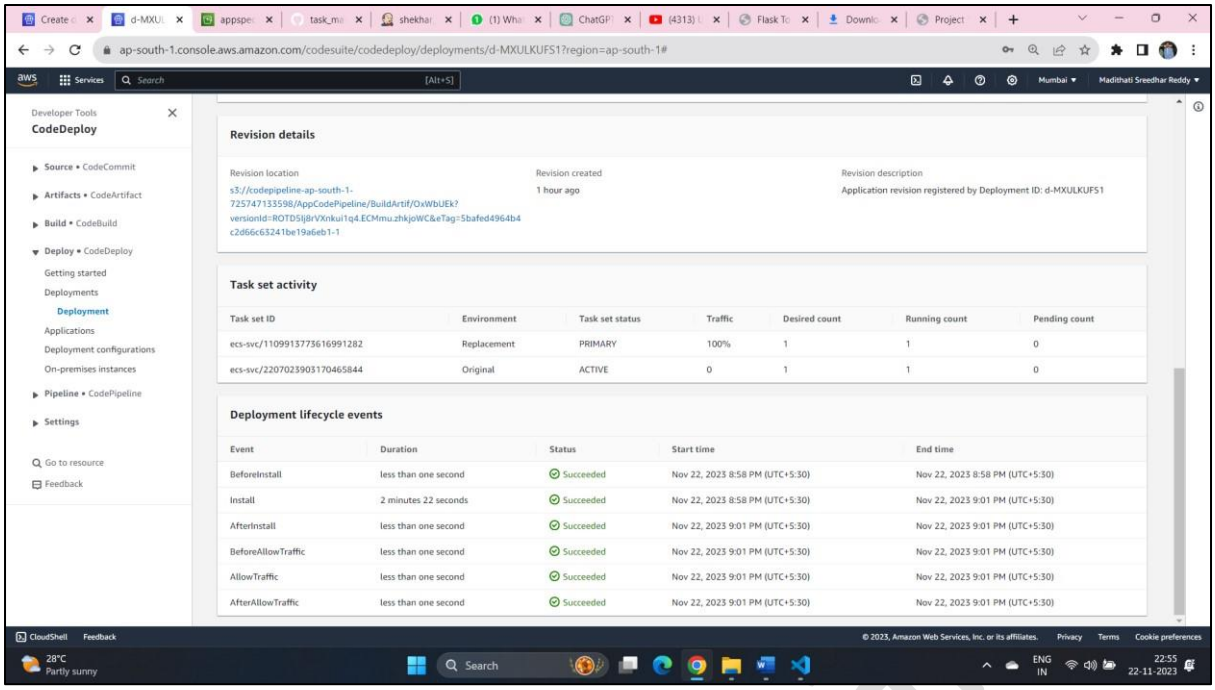
□



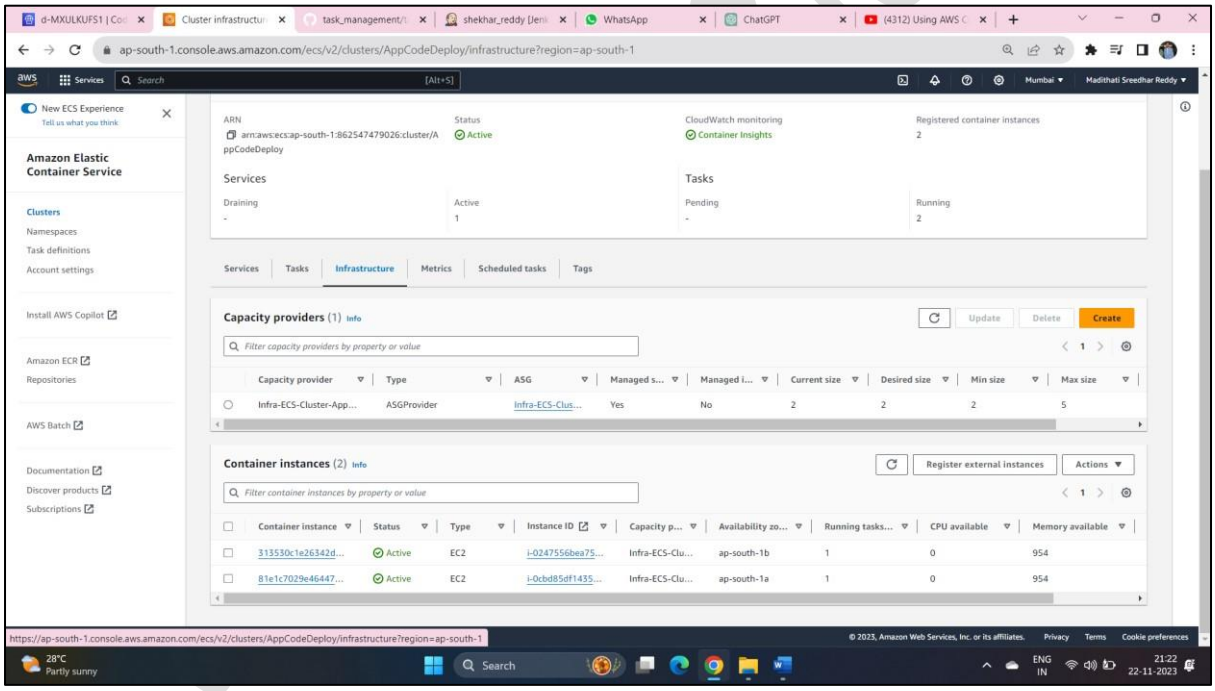
- Once it is fully Completed and all the above steps are performed correctly It will look as below.



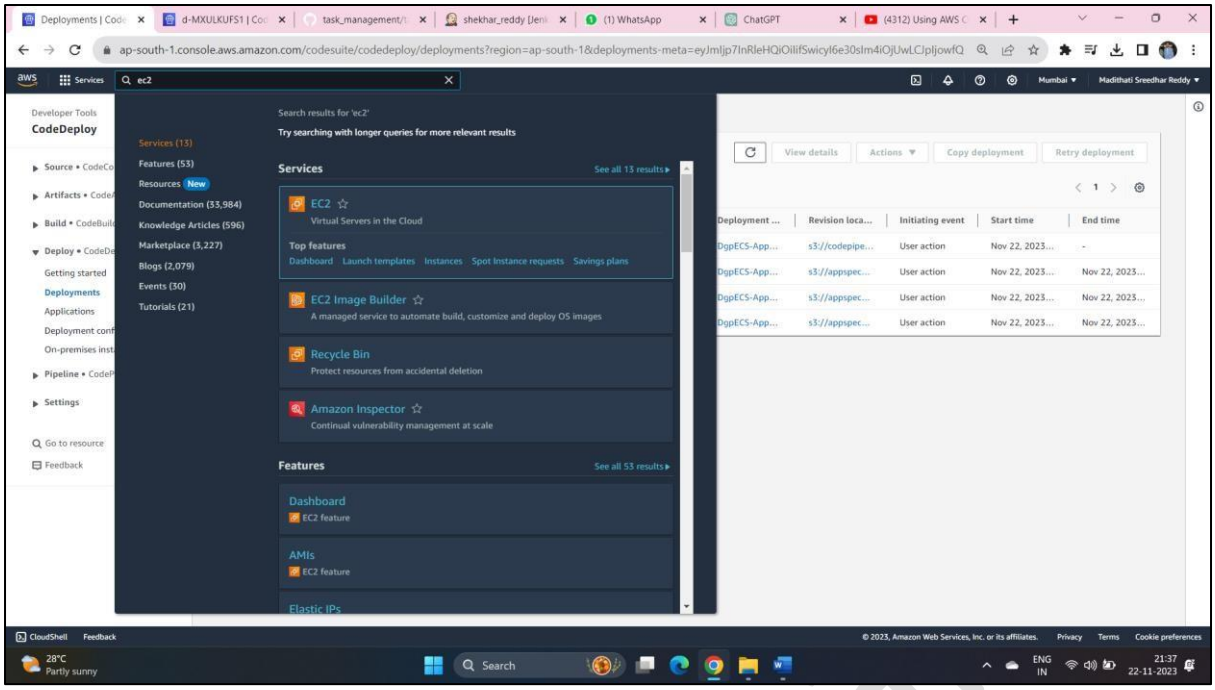
And all the deployment events will be marked as Succeded



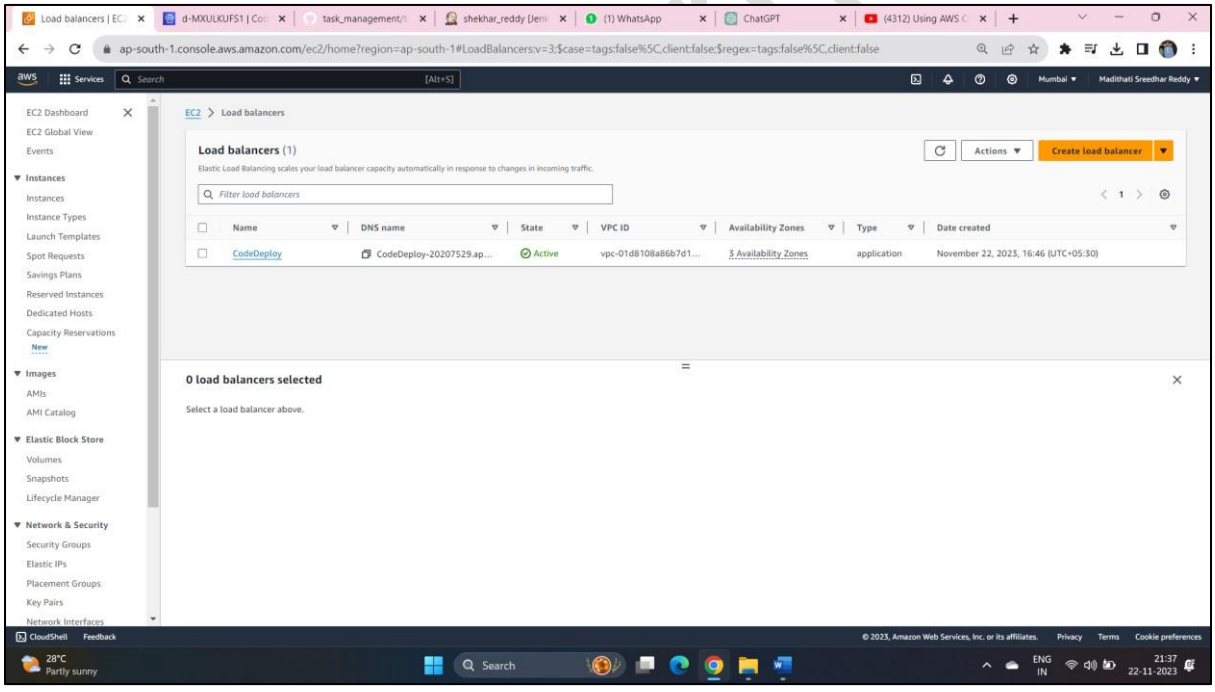
- Navigate to the created Cluster it will resemble as below



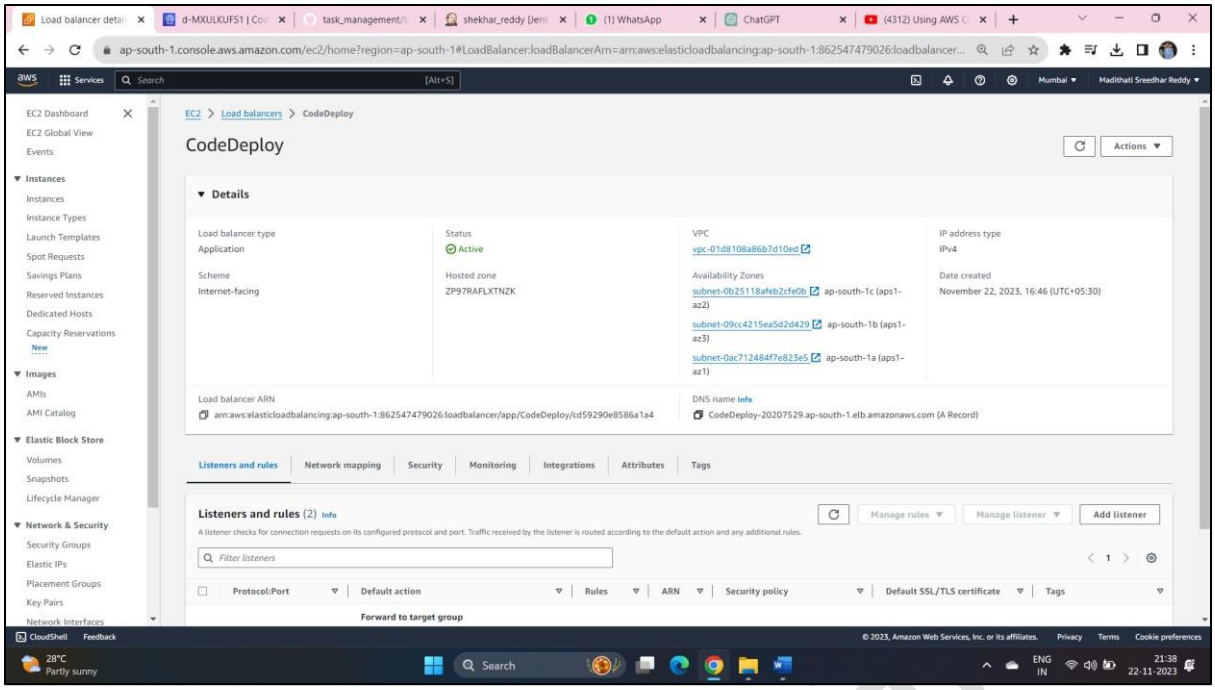
- For Accessing Our Web Application Navigate to the EC2 Console



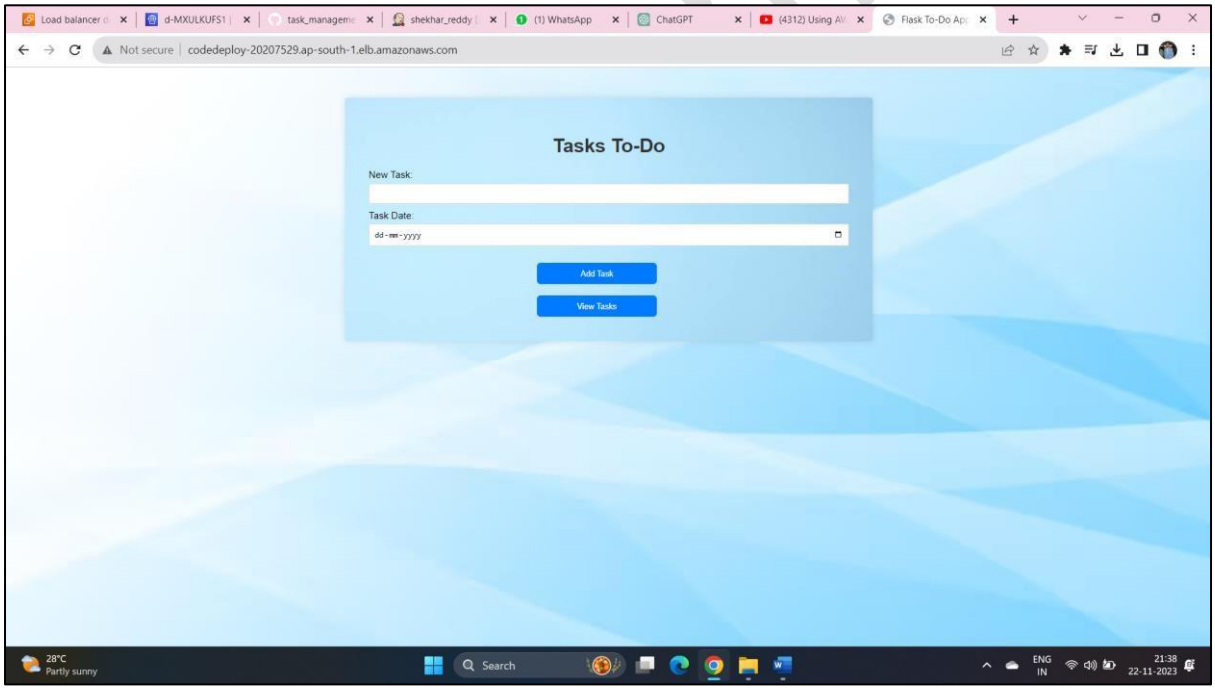
- Choose the Created Load Balancer



- Copy the DNS Name of it

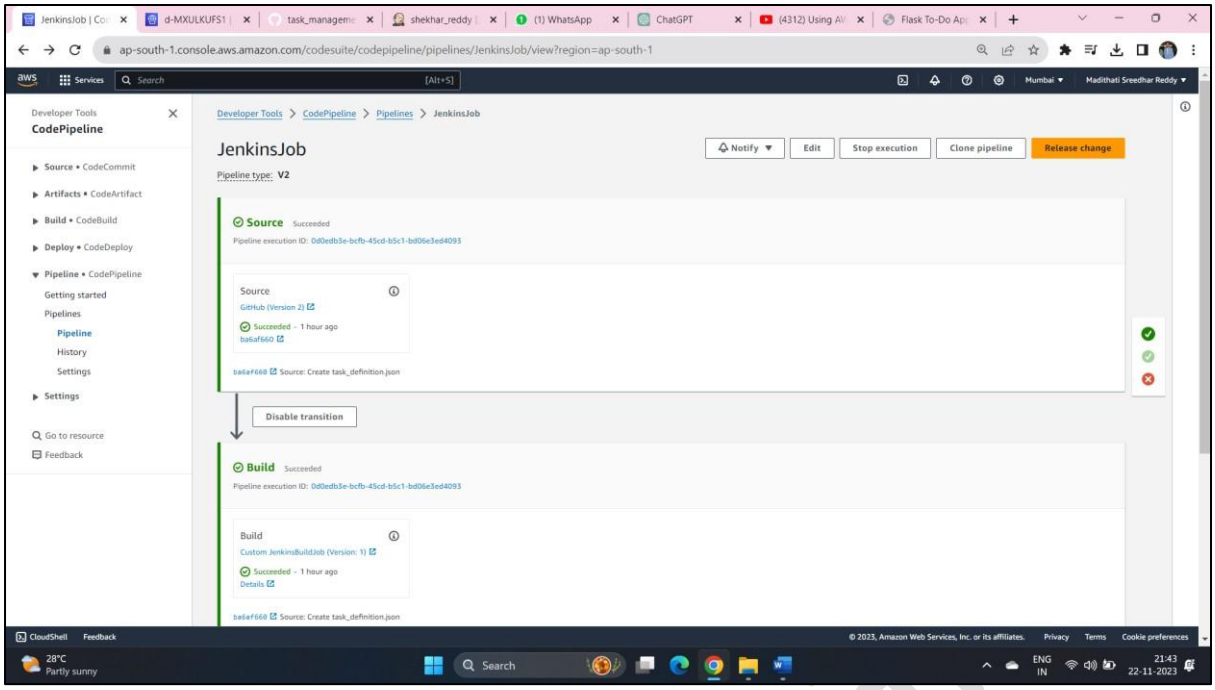


- Paste it on the Web Browser and It will Resembles as below

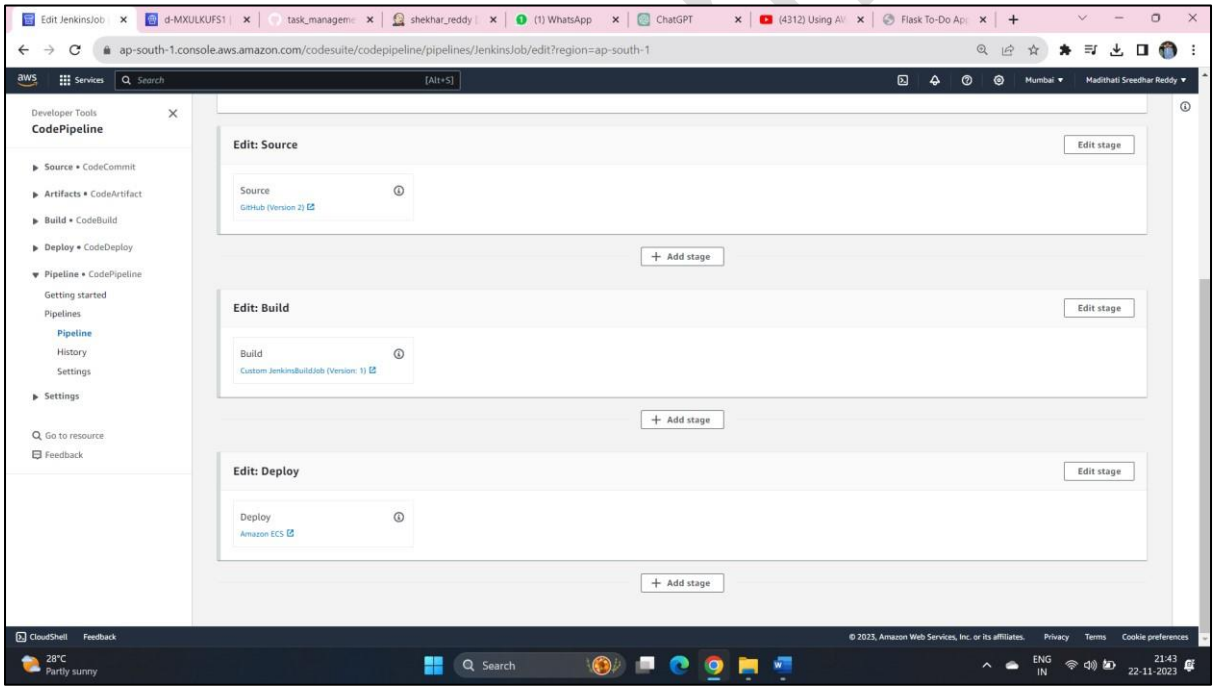


Configuring CodePipeline For Updates:

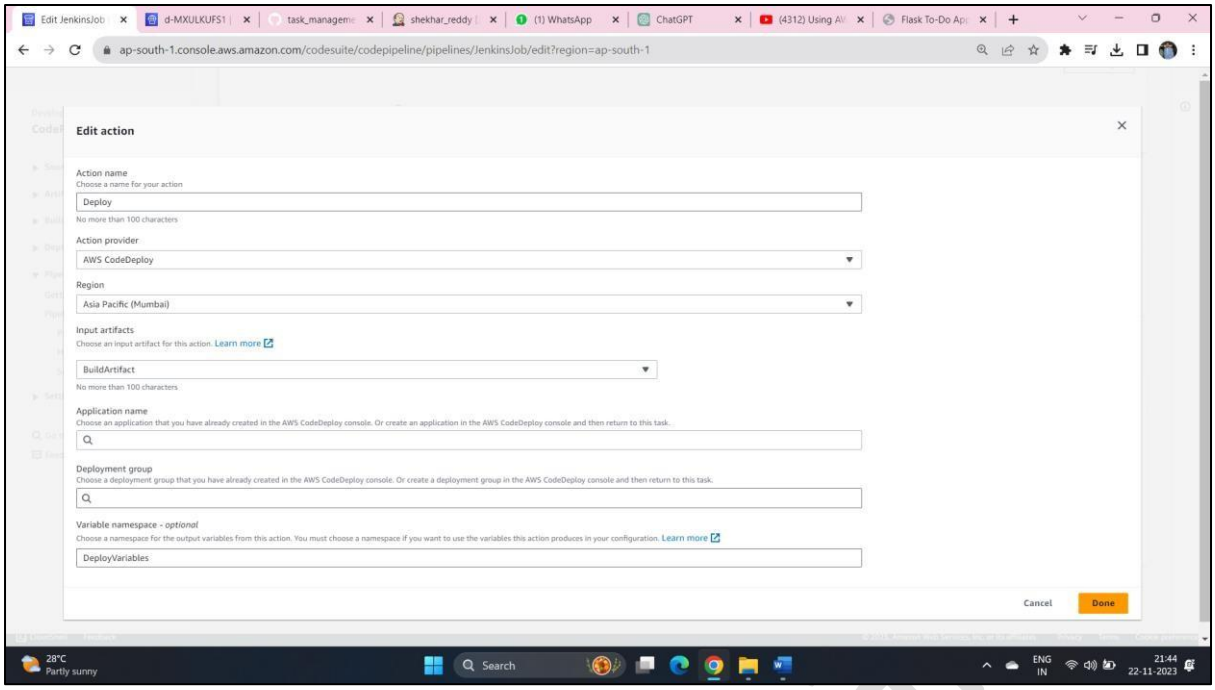
- Navigate to the CodePipeline
- Choose the created pipeline earlier
- Click on Edit



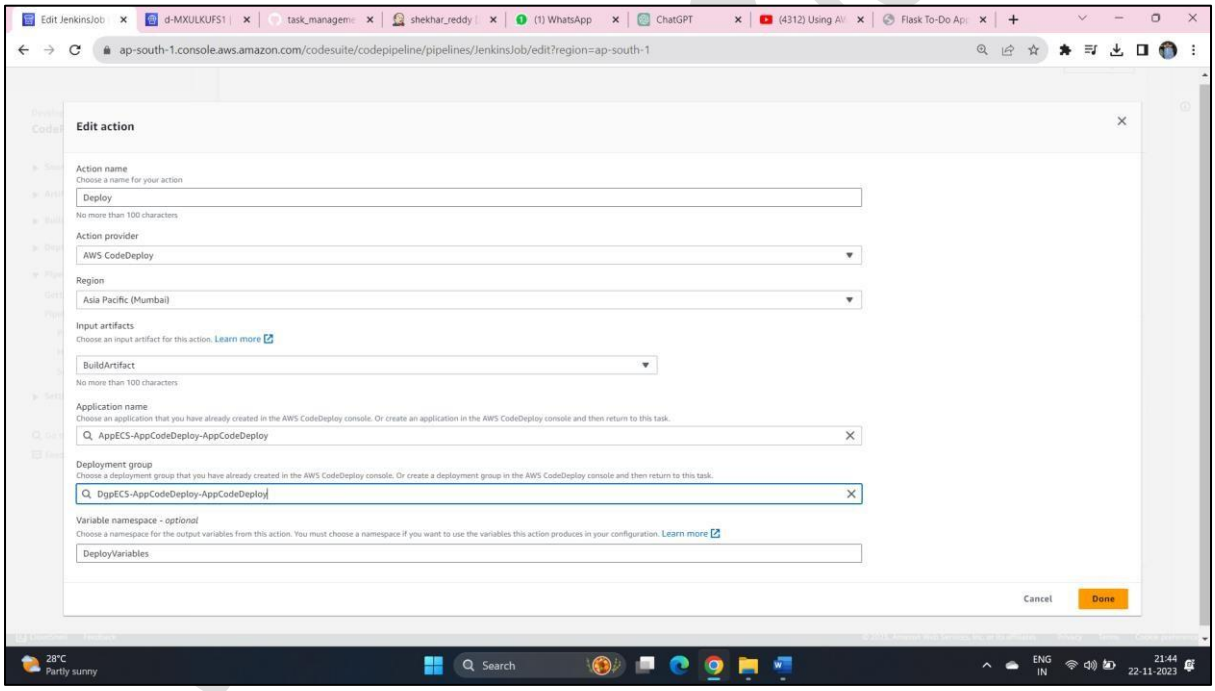
- Navigate to the Deploy Stage
- Choose Edit Stage



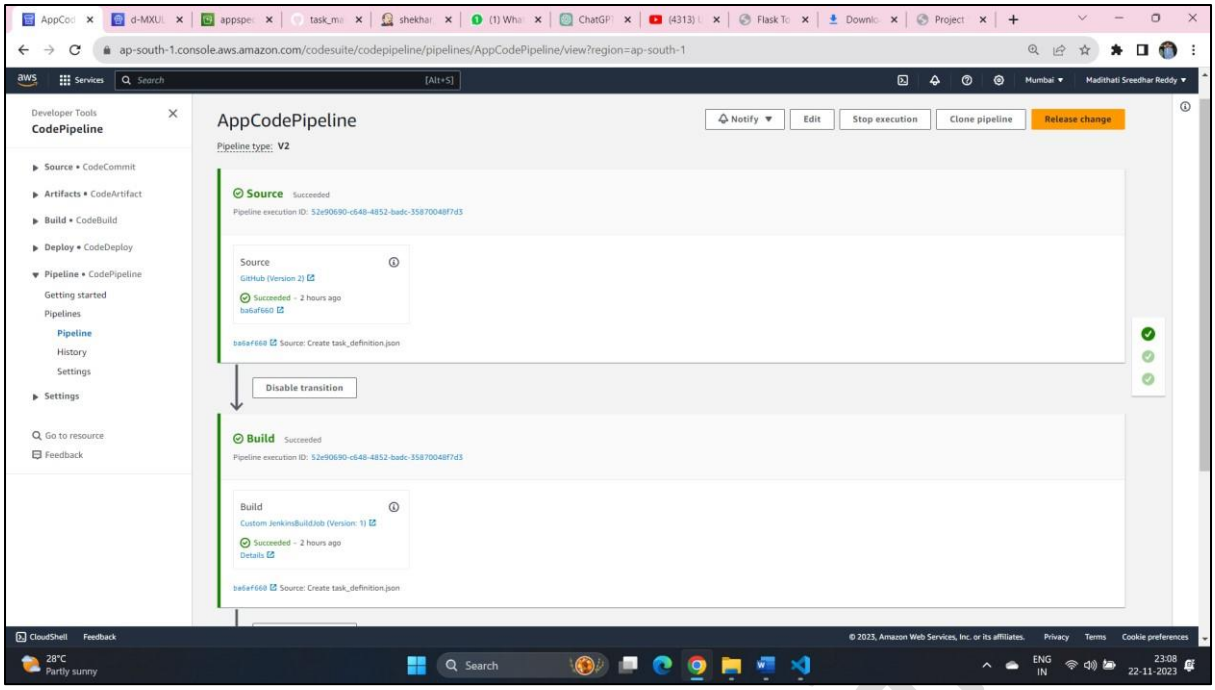
- Choose the Action Provider as AWS CodeDeploy

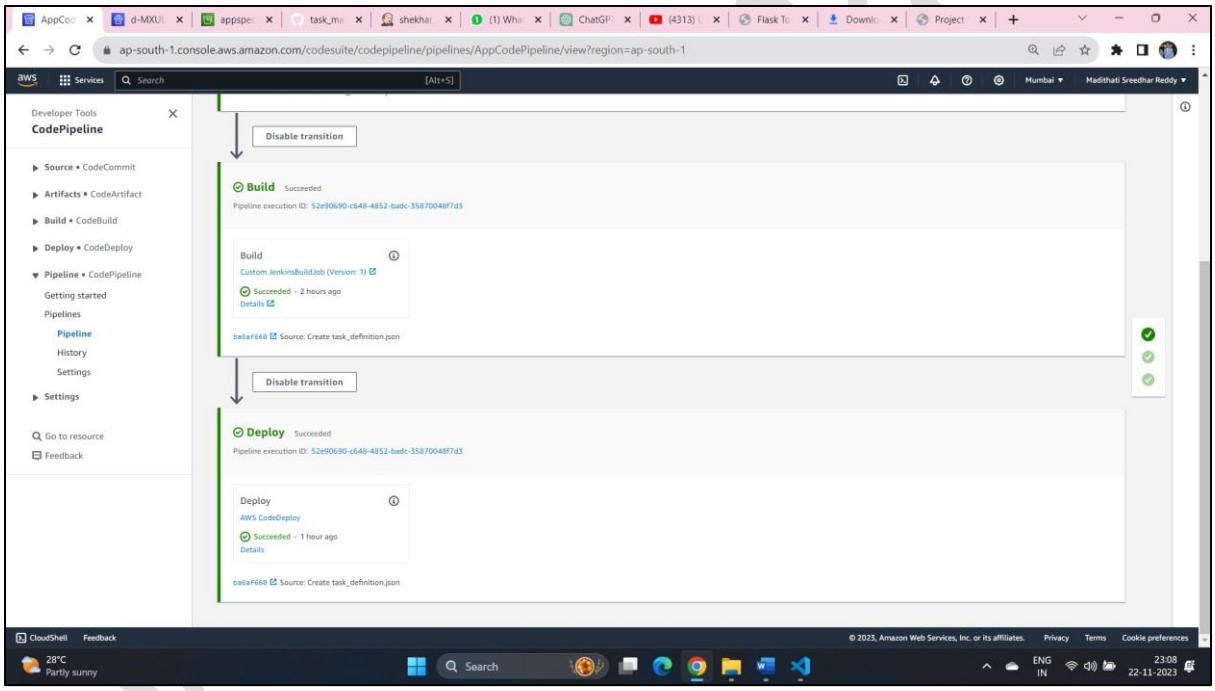


- Then choose the other options correctly from the existing ones



- Now Rerun the Pipeline Again with the New Image that contain the latest updates Once it is Successful it will look as below.



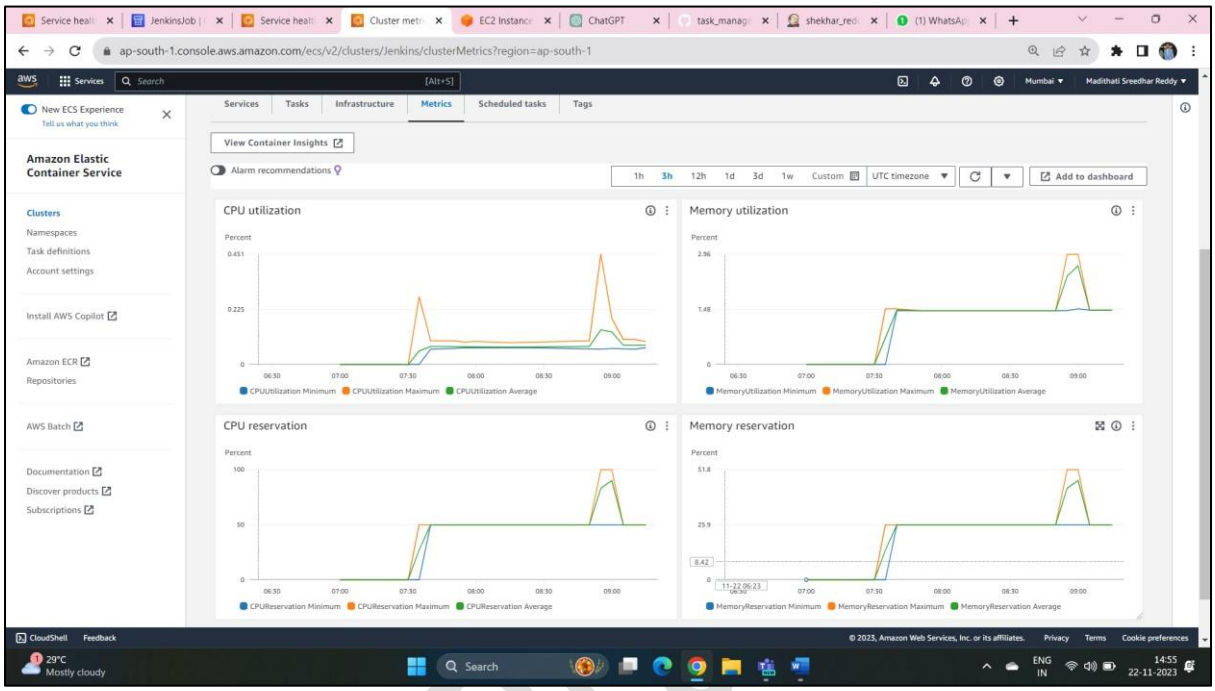


- Everytime we Release the changes It will be running in one instance that is named as Blue with target group TG-1
- Another will be the old Version of our Application running in another instance that is named as Green with Target group TG-2
- If anything goes wrong the loadbalancer will reroute the traffic to the healthy instance.

Monitoring:

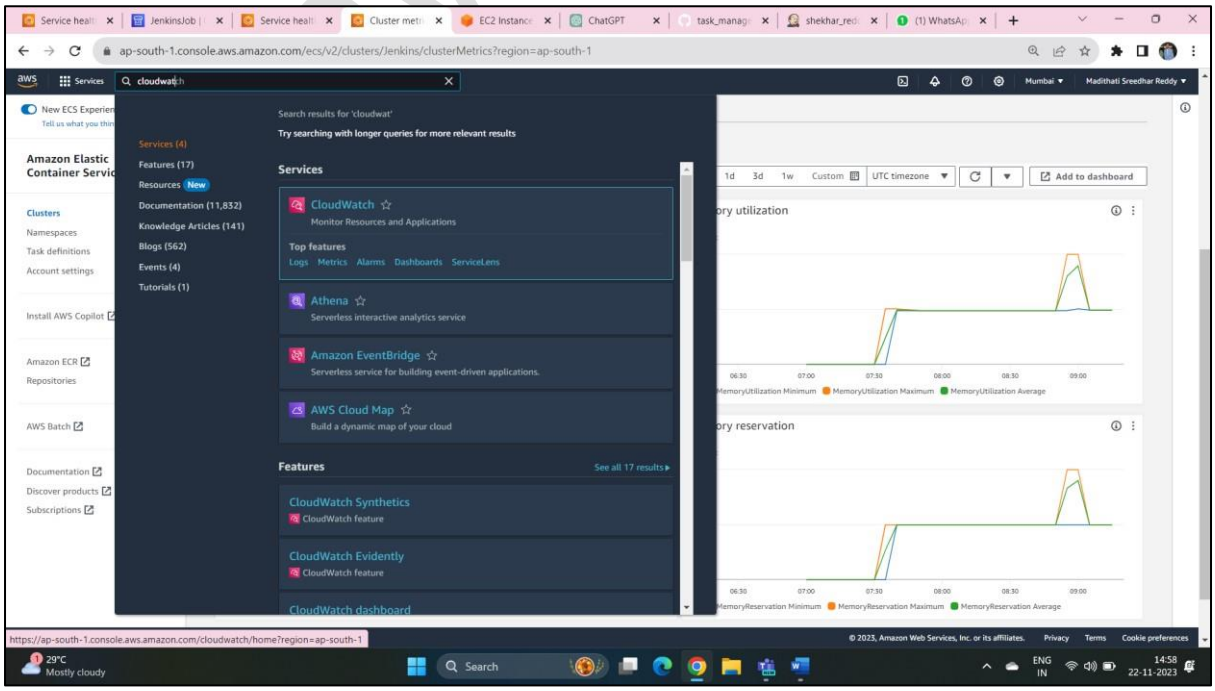
1. View the metrics

- Go to the ECS console
- Navigate to our cluster
- Then click on metrics that will show the ECS metrics like CPU Utilization.



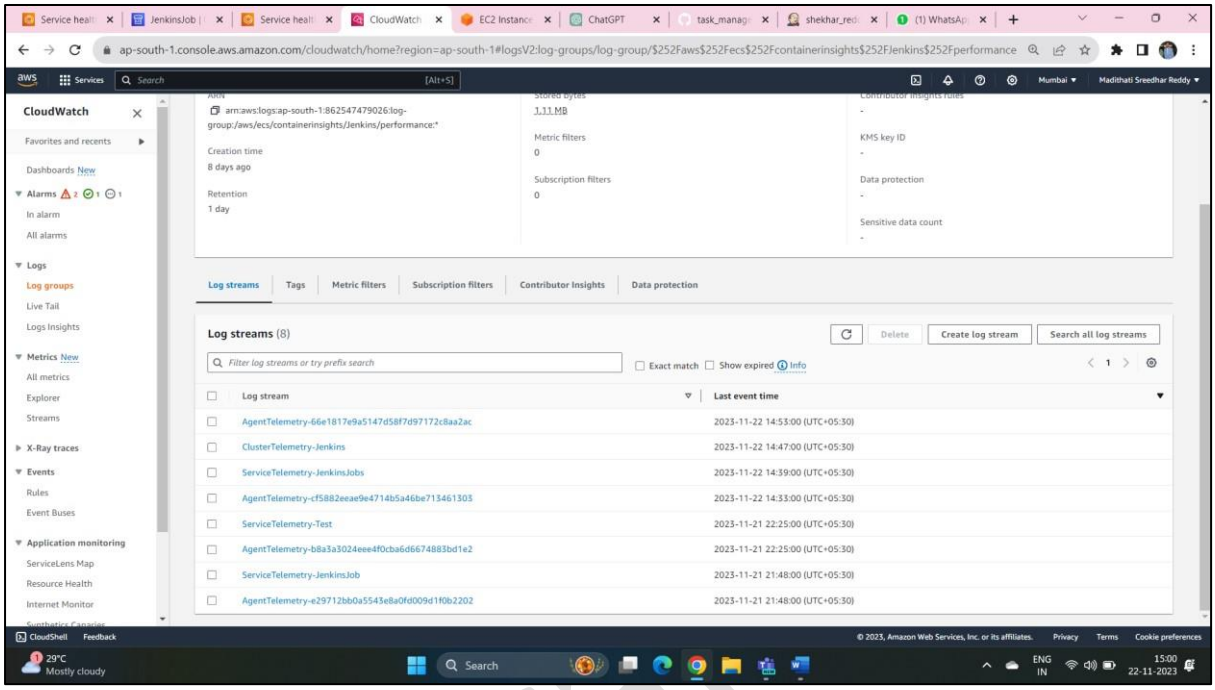
2.View Logs in CloudWatch Console:

- Search for the CloudWatch Console and Navigate to it.

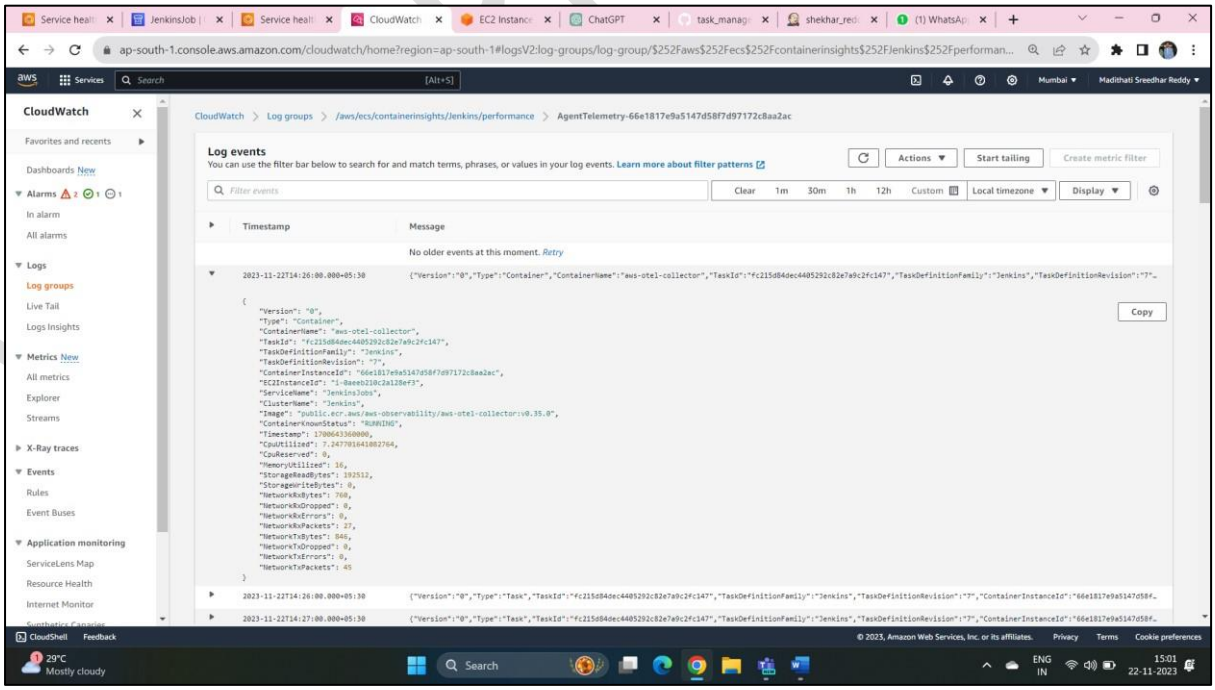


- Navigate to "Logs" in the left navigation pane.

- In the "Log groups" section, find and select the log group associated with your ECS task.
- View and search logs within the log group. Logs are organized by log streams, with each stream representing an instance of your ECS task.

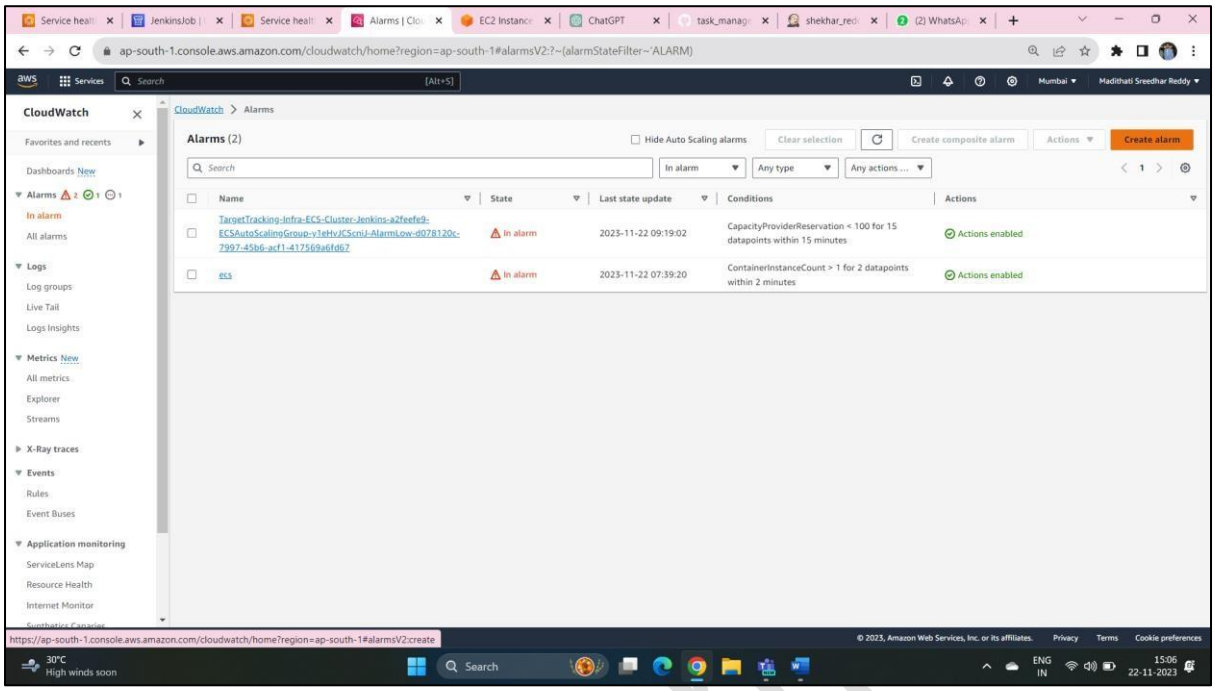


- Navigate to the path mentioned below
- This helps in knowing the live utilization metrics

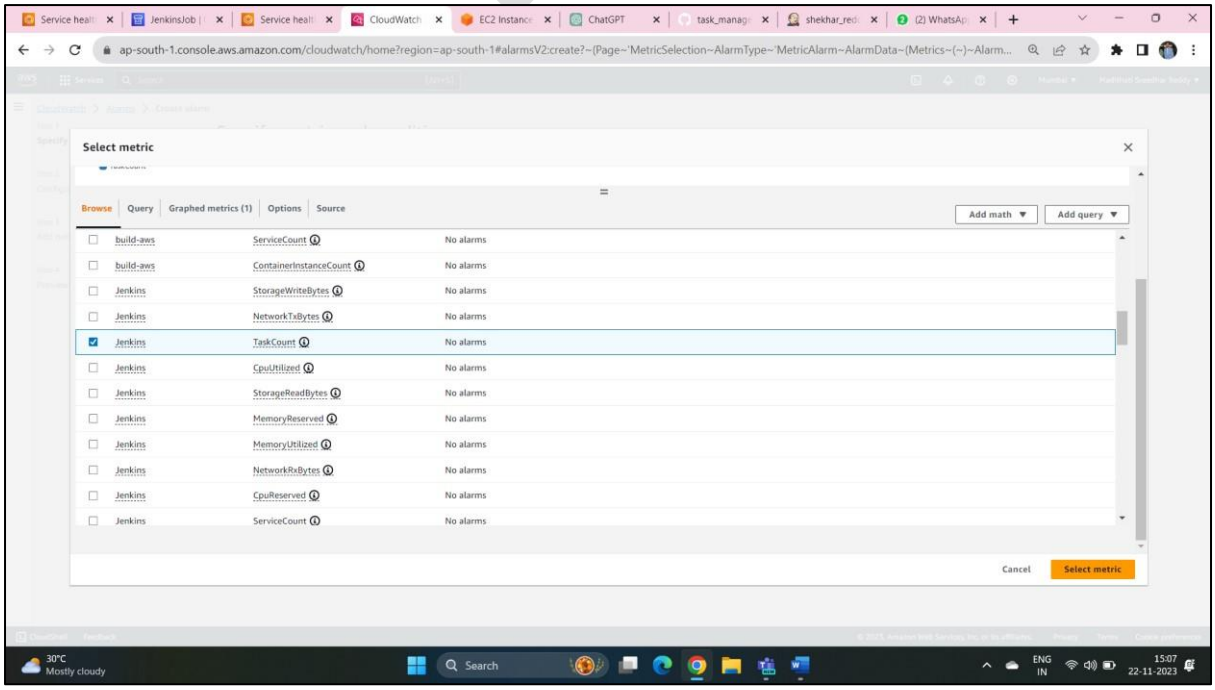


- We can use these logs for troubleshooting issues on any failures.
- We can create Alarm for the events that monitors the cluster and if something goes wrong then It will trigger the tigger the alarm and send an email

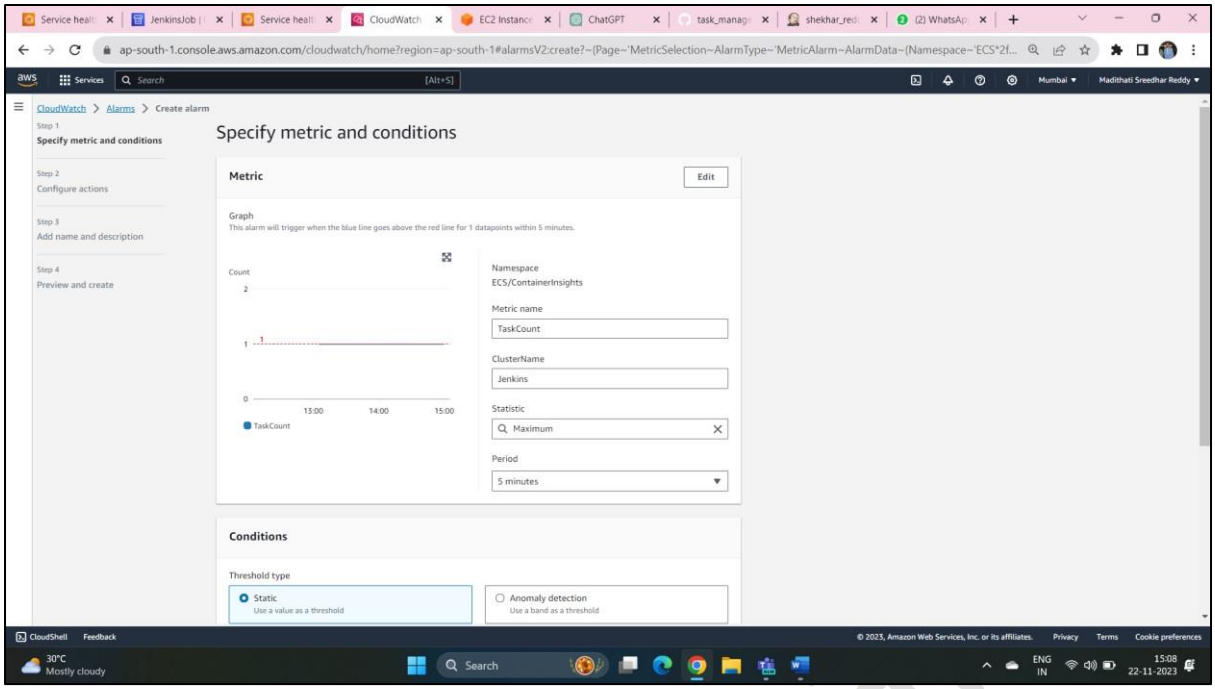
- The Steps for doing this is:
  - In the Same Console Navigate to Alarm



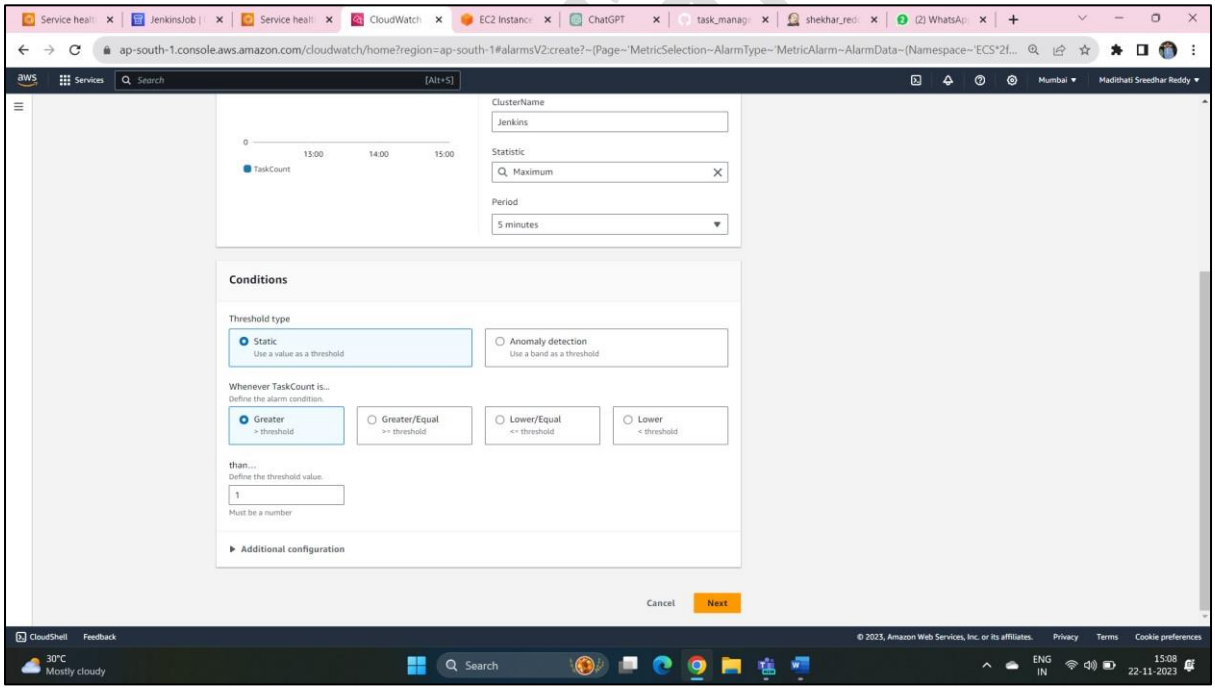
- Click on Create Alarm
- Then select the metric you want to create alarm



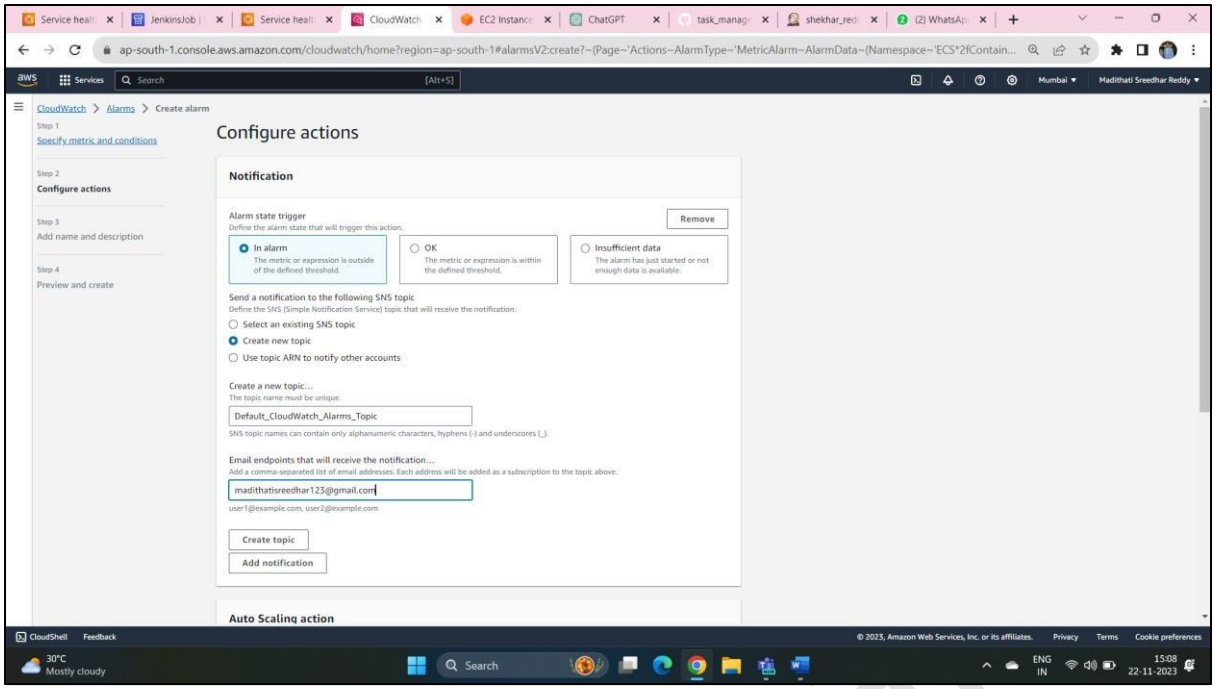
- Then Configure the metrics and conditions



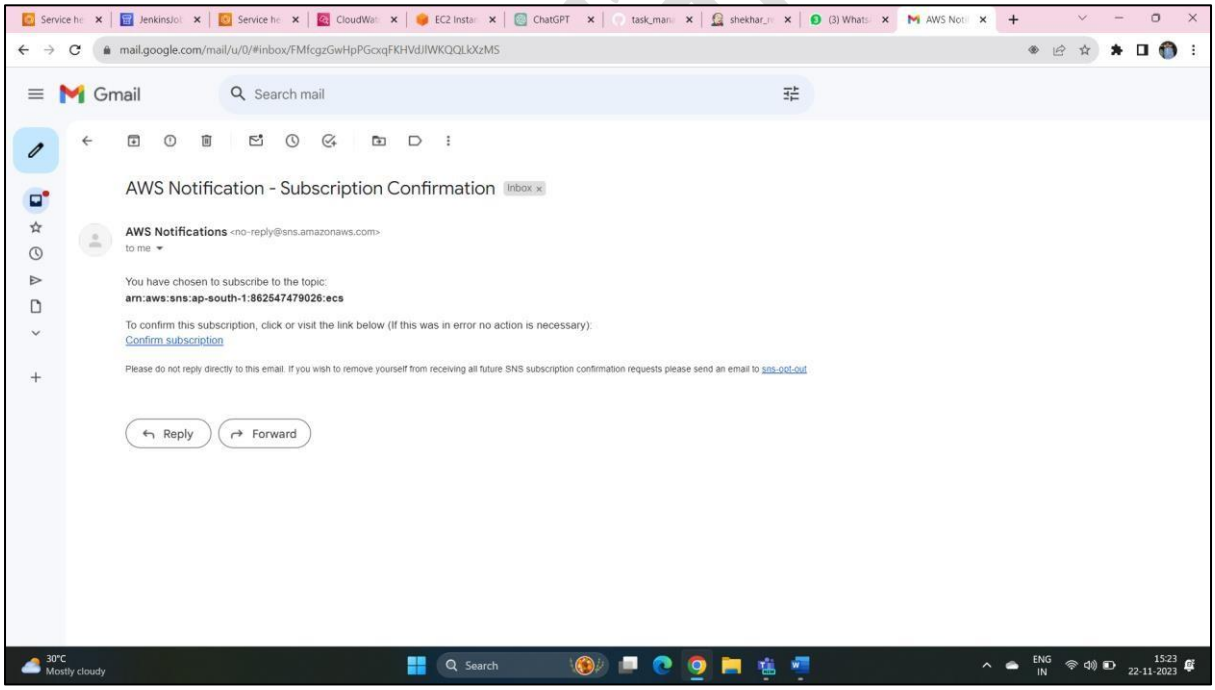
- Specify the Threshold value for it



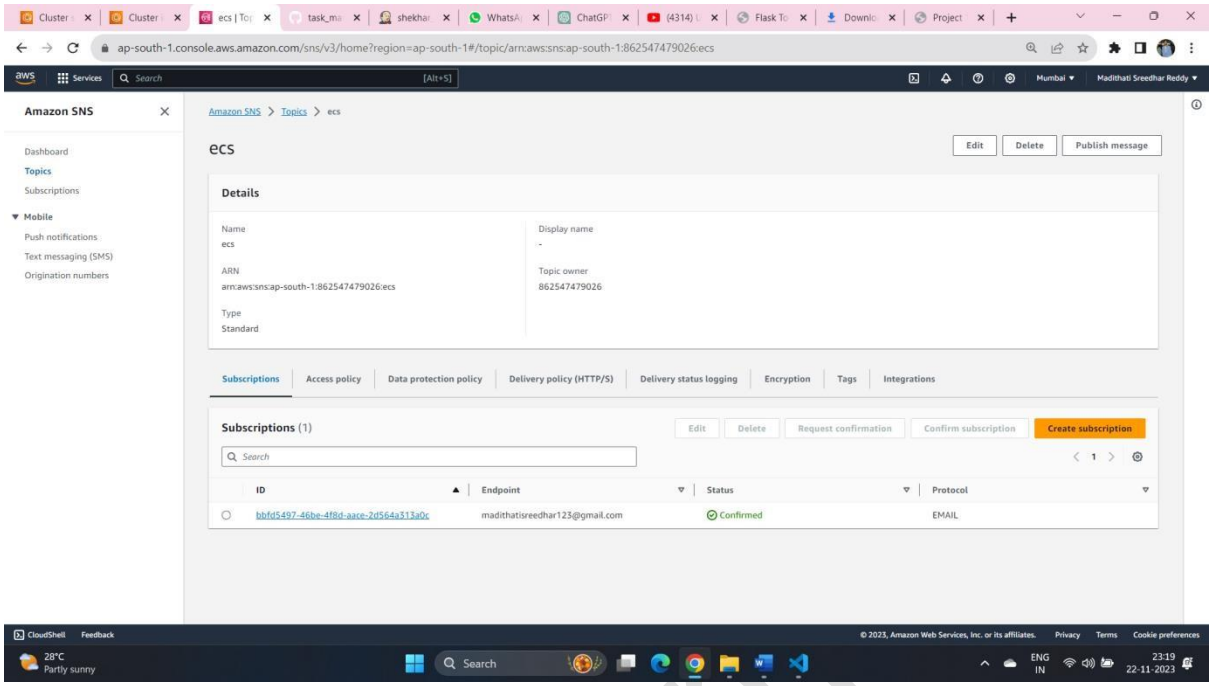
- Click on Next

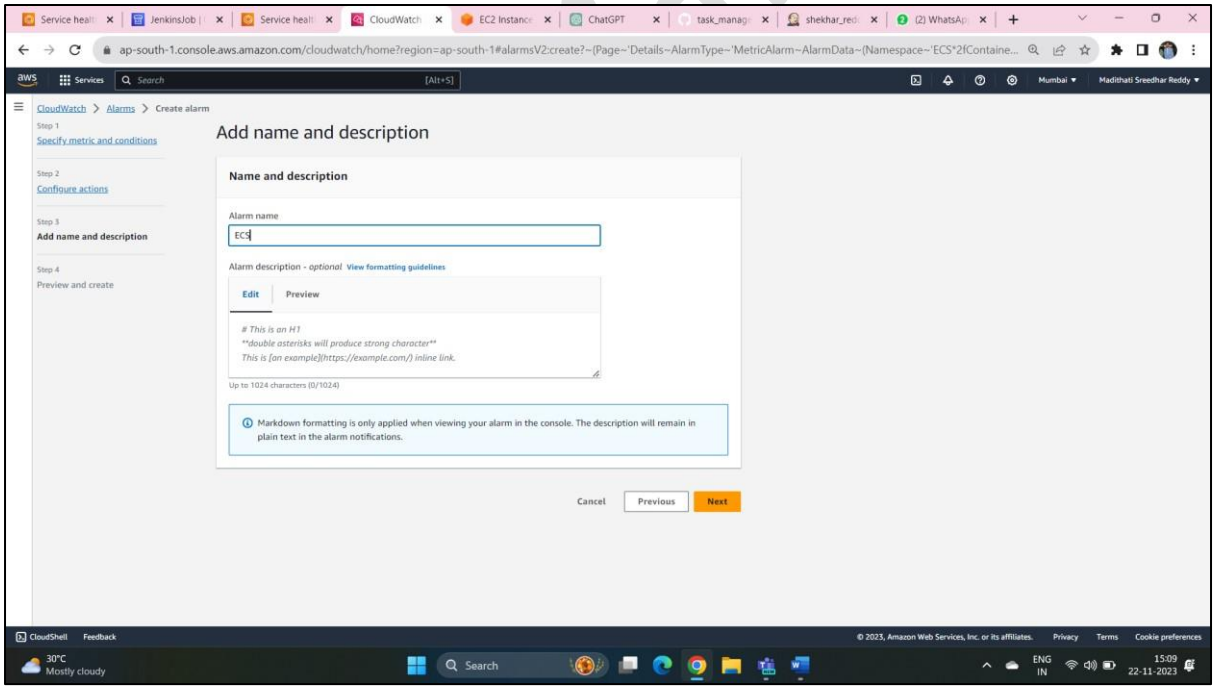


- It will send the Subscription mail Accept it

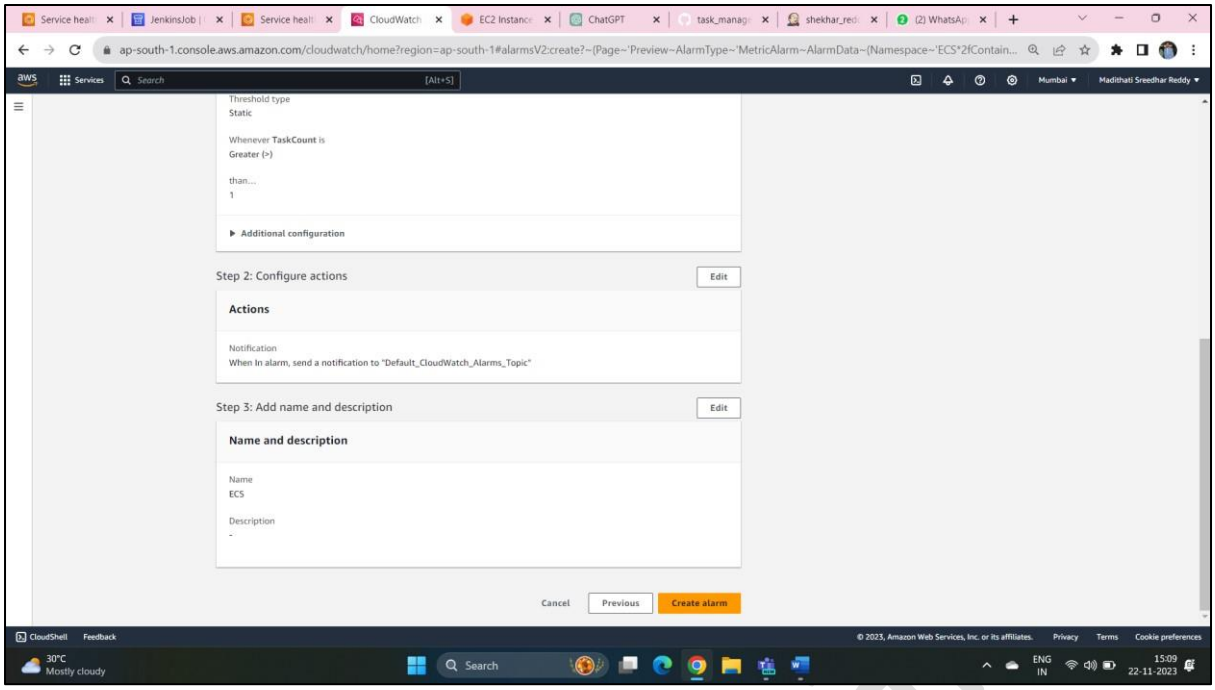


- Click on Create topic



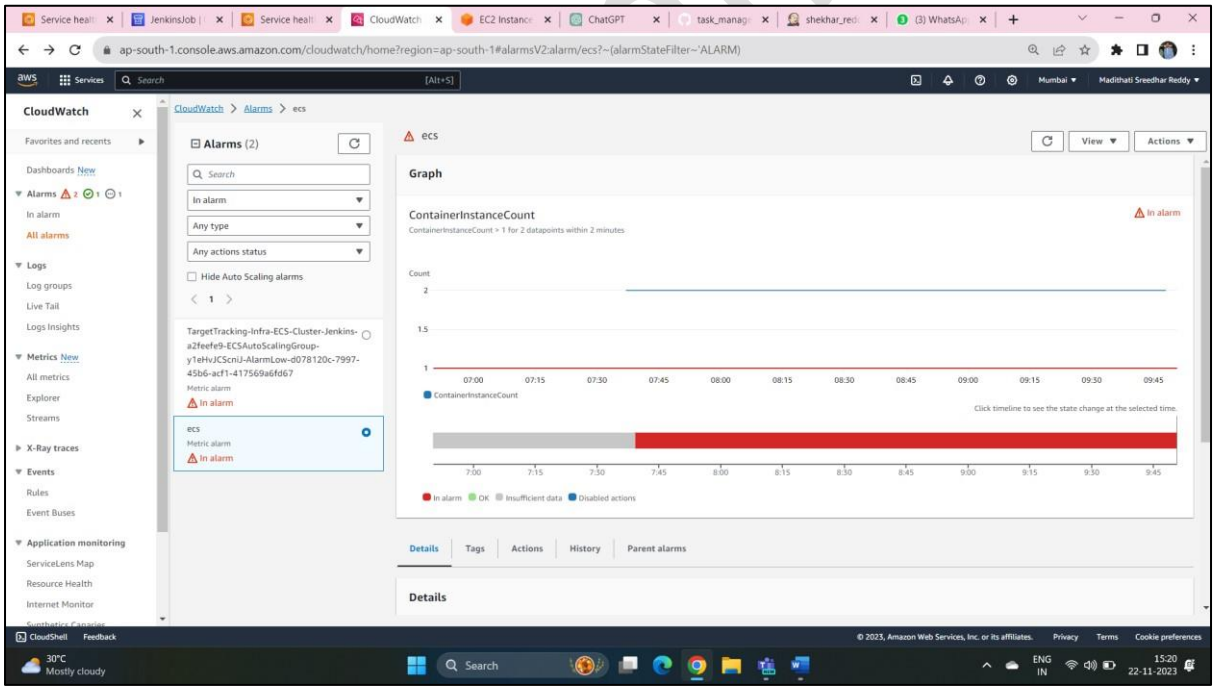


- Click on Next for Preview

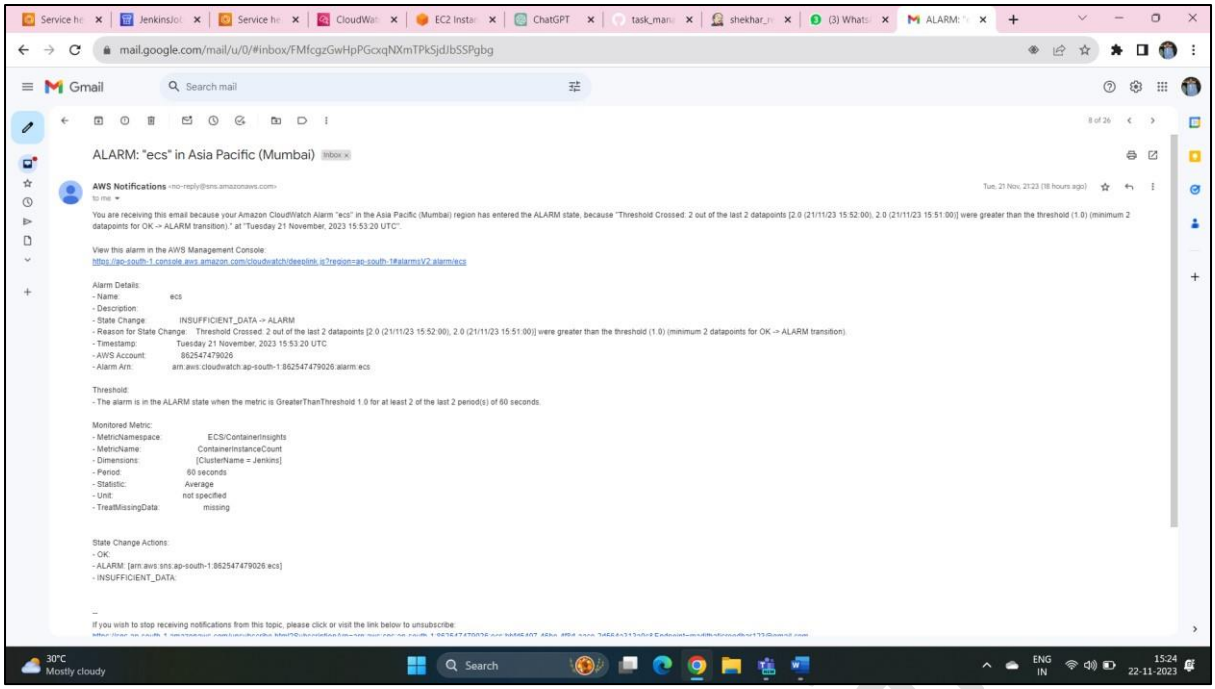


- Click on Create Alarm

- Intially It would in Insufficient data
- After the mentioned time is met, it will turn to the Alarm state



- If Anything goes Wrong It will change from Alarm to the OK state Then it will send an email defining the breach of the mentioned metric.


mail.google.com/mail/u/0/#inbox/FMfcgzGwHpPGcxqNXmTPkSjdJbSSPgbg
Gmail
Search mail
ALARM: "ecs" in Asia Pacific (Mumbai)
Inbox
AWS Notifications <no-reply@sns.amazonaws.com>
Tue, 21 Nov, 21:23 (18 hours ago)
to me
You are receiving this email because your Amazon CloudWatch Alarm "ecs" in the Asia Pacific (Mumbai) region has entered the ALARM state, because "Threshold Crossed: 2 out of the last 2 datapoints [2.0 (21/11/23 15:52:00), 2.0 (21/11/23 15:51:00)] were greater than the threshold (1.0) (minimum 2 datapoints for OK -> ALARM transition)." at "Tuesday 21 November, 2023 15:53:20 UTC".
View this alarm in the AWS Management Console:
https://ap-south-1.console.aws.amazon.com/cloudwatch/deeplink.js?region=ap-south-1#alarmsV2:alarm/ecs
Alarm Details:
- Name: ecs
- Description:
- State Change: INSUFFICIENT_DATA -> ALARM
- Reason for State Change: Threshold Crossed: 2 out of the last 2 datapoints [2.0 (21/11/23 15:52:00), 2.0 (21/11/23 15:51:00)] were greater than the threshold (1.0) (minimum 2 datapoints for OK -> ALARM transition).
- Timestamp: Tuesday 21 November, 2023 15:53:20 UTC
- AWS Account: 862547479026
- Alarm Arn: arn:aws:cloudwatch:ap-south-1:862547479026:alarm:ecs
Threshold:
- The alarm is in the ALARM state when the metric is GreaterThanThreshold 1.0 for at least 2 of the last 2 period(s) of 60 seconds.
Monitored Metric:
- MetricNamespace: ECS/ContainerInsights
- MetricName: ContainerInstanceCount
- Dimensions: [ClusterName = Jenkins]
- Period: 60 seconds
- Statistic: Average
- Unit: not specified
- TreatMissingData: missing
State Change Actions:
- OK:
- ALARM: [arn:aws:sns:ap-south-1:862547479026:ecs]
- INSUFFICIENT_DATA:
--
If you wish to stop receiving notifications from this topic, please click or visit the link below to unsubscribe:
30°C
Mostly cloudy
Search
ENG
IN
15:24
22-11-2023