Fully Automated Scalable CICD Pipeline for Serverless Django on AWS

Table of Contents

1. Introduction:

This document outlines the steps to configure the serverless automation architecture.

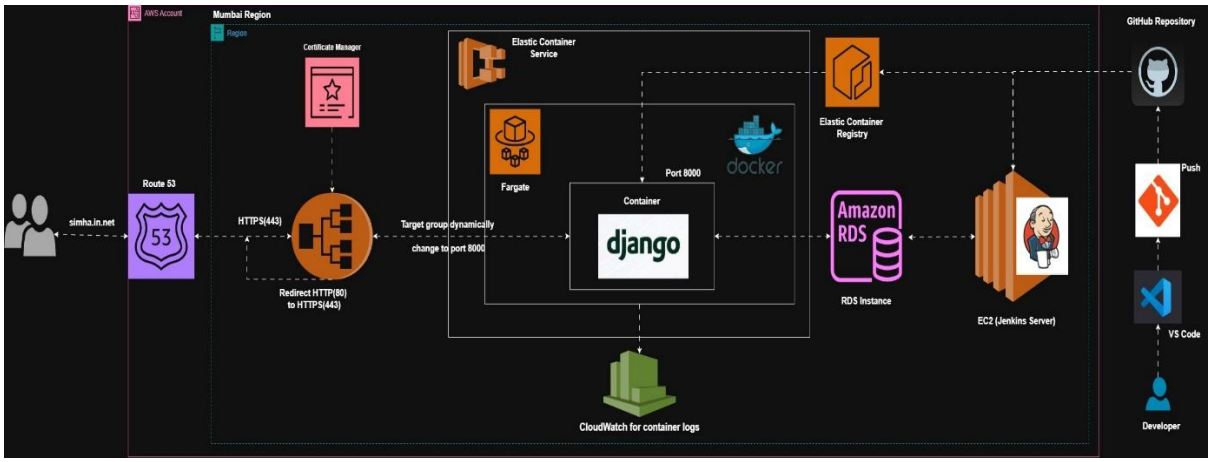
2. Pre-requisites:

Appropriate Access/Credentials to login to the AWS Console.

- AmazonEC2FullAccess
- AmazonVPCFullAccess
- AmazonRDSFullAccess
- AmazonECSFullAccess
- AmazonElasticContainerRegistryPublicFullAccess
- AWSCertificateManagerFullAccess
- CloudWatchEventsFullAccess
- AmazonRoute53FullAccess

3. Summary:

This architecture represents a fully automated and scalable CI/CD pipeline for a Django application on AWS using rolling updates for seamless deployments. The process begins with a developer pushing code from VS Code to GitHub, which triggers Jenkins on an EC2 instance to automate the build and deployment workflow. Jenkins pulls the latest code, builds a Docker image, and pushes it to Amazon Elastic Container Registry (ECR). AWS ECS (Fargate) then pulls the updated image and deploys it within containers running Django on port 8000, dynamically managed by an Application Load Balancer (ALB), which routes HTTP (80) traffic securely to HTTPS (443). The application interacts with an Amazon RDS database, with logs centrally monitored via CloudWatch. Route 53 manages domain resolution, ensuring seamless traffic distribution. Rolling updates are used with a minimum of 100% running tasks and a maximum of 200% to guarantee zero downtime. This setup enables automated deployments, high availability, and scalability, ensuring a secure, efficient, and fully optimized DevOps pipeline with minimal manual intervention. 🚀

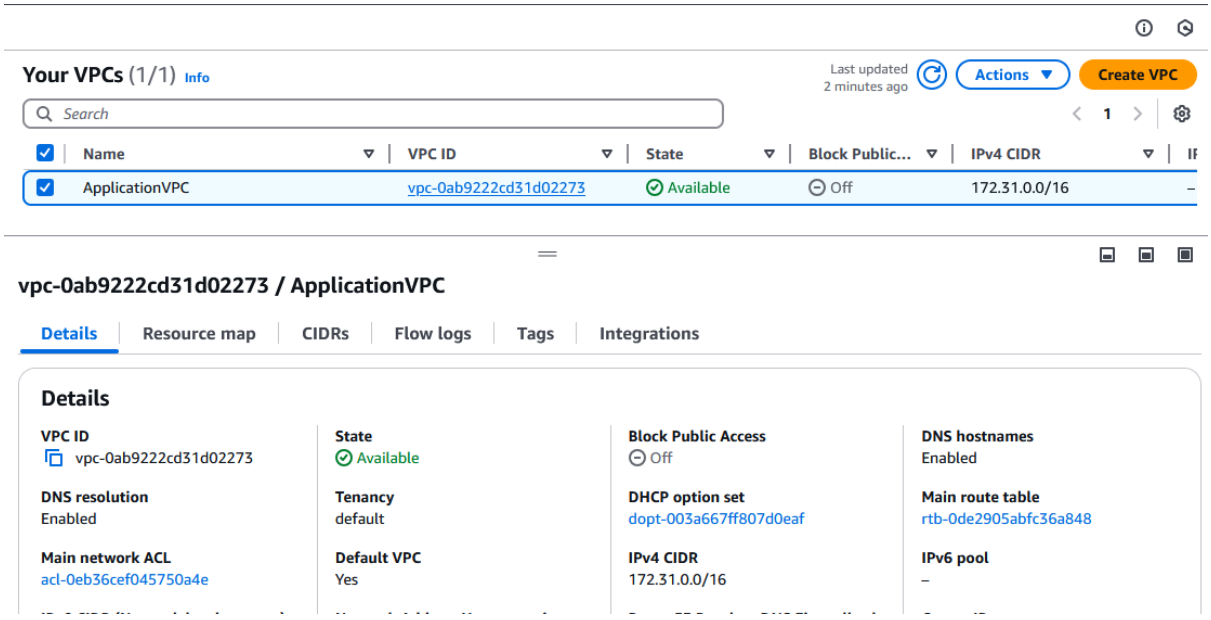


4. Procedure Module wise:

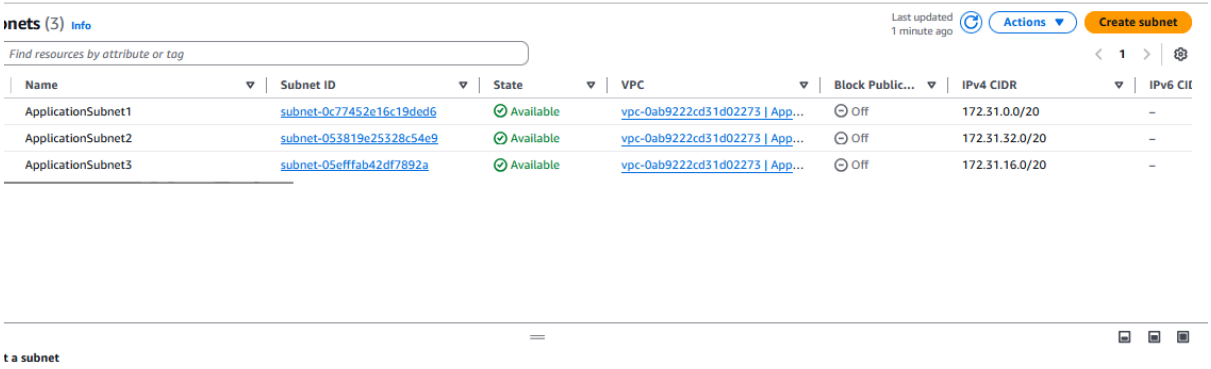
Module 1

Setting up a VPC and an EC2 instance for a Jenkins server with the necessary configuration

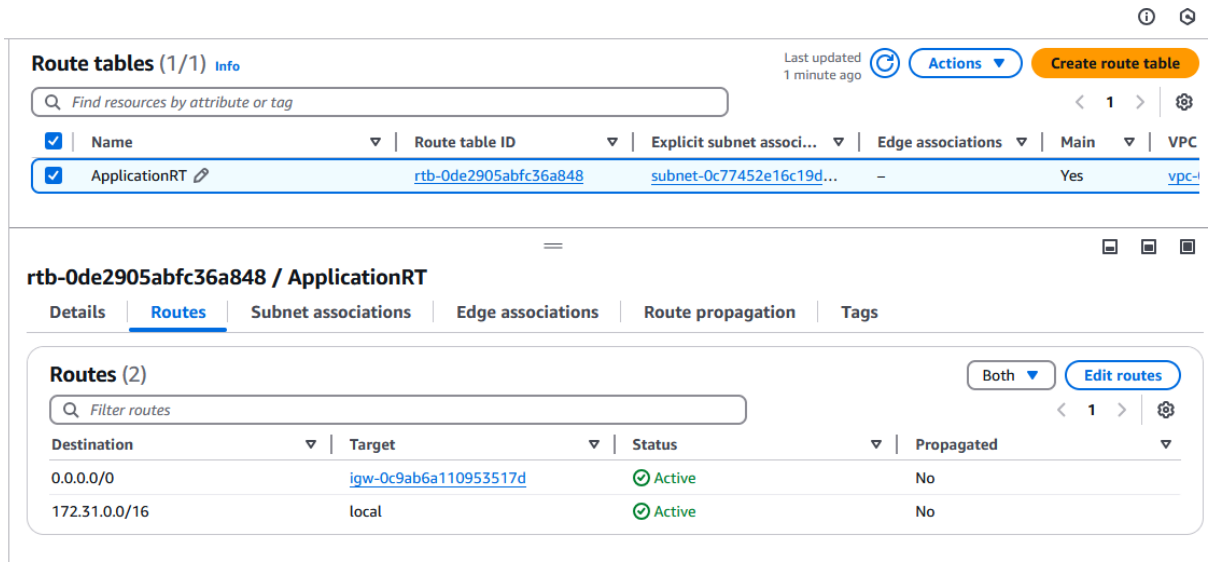
1. Create a VPC with a CIDR block of “172.31.0.0/16” in the Mumbai region.



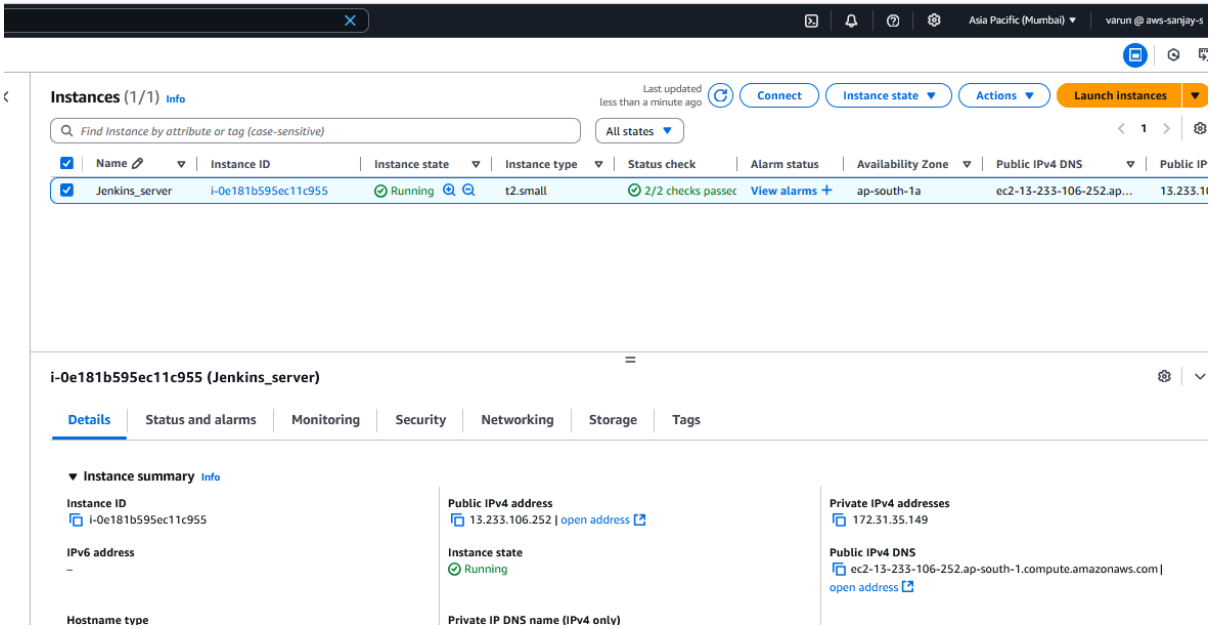
2. Create three subnets within the VPC, each corresponding to a different availability zone, and modify their settings to allow the assignment of public IP addresses.



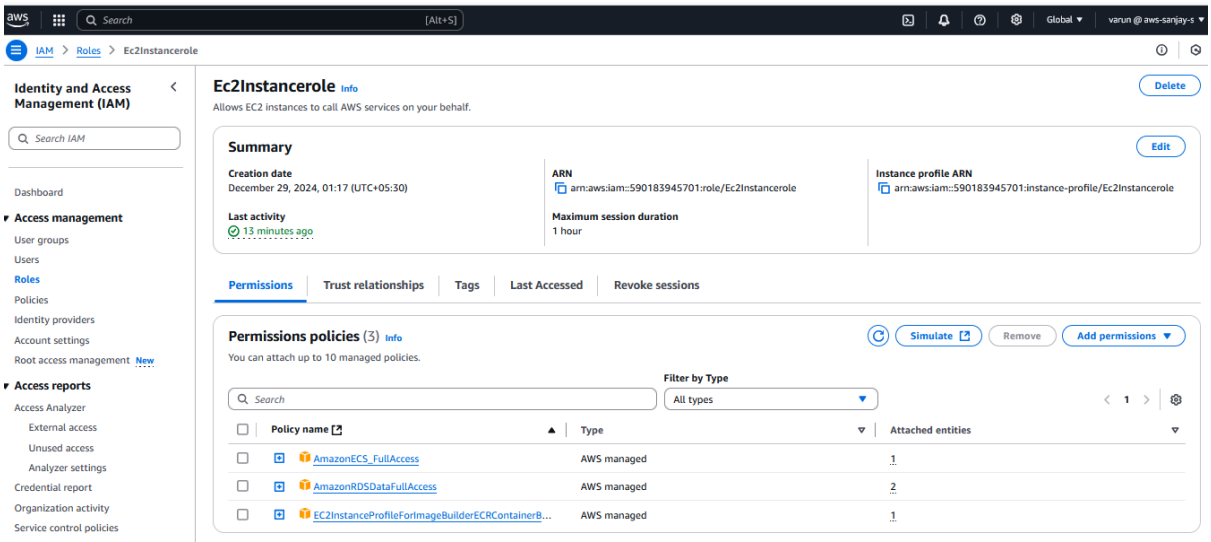
3. Create a route table and associate it with all three subnets, allowing internet-bound traffic by adding a route to the internet gateway.



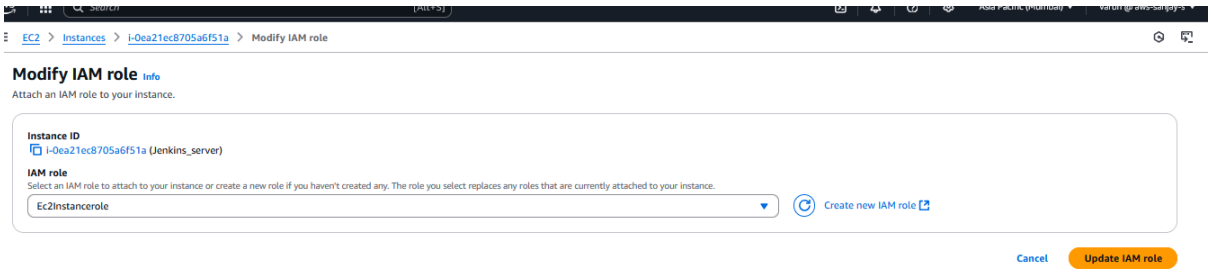
4. Go to EC2 and launch an instance named **Jenkins_server** within the configured VPC, ensuring it is set up for automation tasks without impacting the production environment.



5. Create an IAM role with full access to **RDS, ECS, EC2InstanceProfileImageBuilder for ECR containers**, ensuring it has the necessary permissions for seamless integration and automation.



6. Assign the created IAM role to the **Jenkins EC2 instance**, allowing it to seamlessly access **RDS, ECS, EC2 InstanceProfileImageBuilder ECR containers** for smooth operation.



7. Install the required packages mentioned below,

a. Install Docker and docker-compose:

1. sudo apt update

2. sudo apt upgrade

3. sudo apt install -y apt-transport-https ca-certificates curl software-properties-common

4. curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o /usr/share/keyrings/docker-archive-keyring.gpg

5. echo "deb [arch=$(dpkg --print-architecture) signed-by=/usr/share/keyrings/docker-archive-keyring.gpg] https://download.docker.com/linux/ubuntu $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null

6. sudo apt update

7. sudo apt install -y docker-ce docker-ce-cli containerd.io

8. sudo systemctl status docker

9. sudo apt install docker-compose

b. Install the AWS CLI:

1. sudo apt install aws-cli

OR

1. curl "https://awscli.amazonaws.com/awscli-exe-linux-x86_64.zip" -o "awscliv2.zip"

2. sudo apt update

3. sudo apt install unzip

4. unzip awscliv2.zip

5. sudo ./aws/install

c. Install the Jenkins:

1. sudo apt update

2. sudo apt install -y openjdk-17-jdk

3. curl -fsSL https://pkg.jenkins.io/debian-stable/jenkins.io-2023.key | sudo tee /usr/share/keyrings/jenkins-keyring.asc > /dev/null

4. echo "deb [signed-by=/usr/share/keyrings/jenkins-keyring.asc] https://pkg.jenkins.io/debian-stable binary/" | sudo tee /etc/apt/sources.list.d/jenkins.list > /dev/null

5. sudo apt update

6. sudo apt install -y jenkins

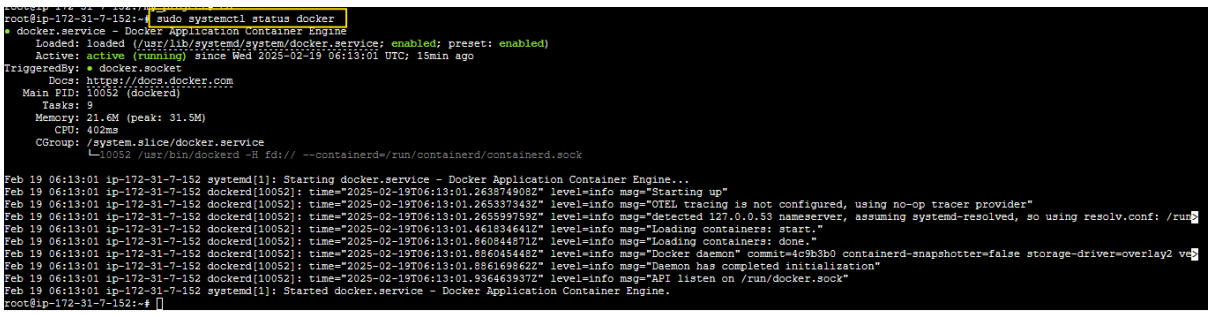
7. sudo systemctl enable --now jenkins

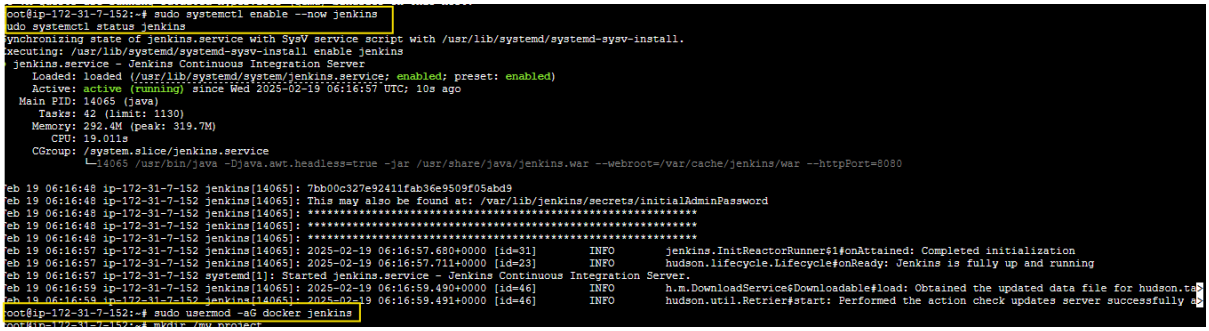
sudo systemctl status Jenkins

d. Give the full access to Jenkins user for using Docker:

1. sudo usermod -aG docker Jenkins

2. systemctl restart Jenkins

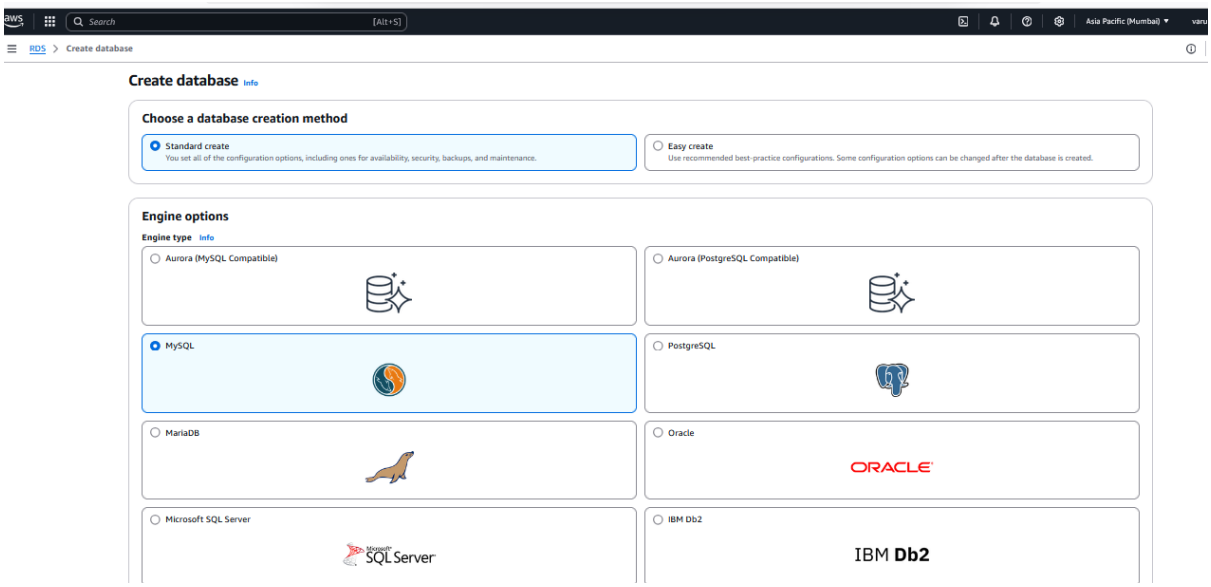




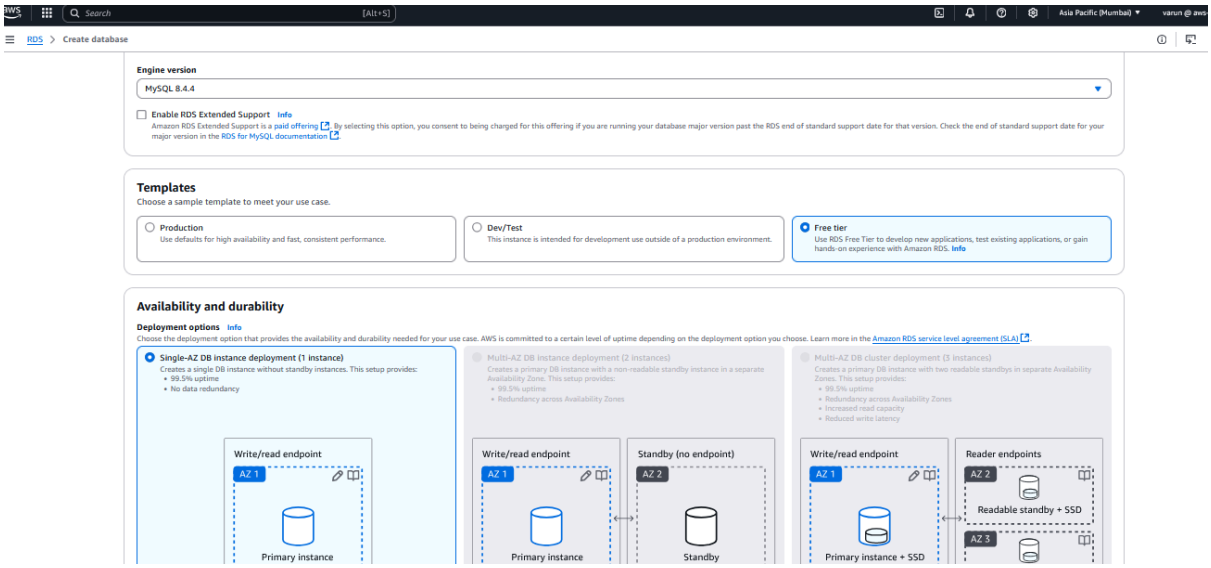
Module 2

Set up an RDS database, configure it appropriately, and create the necessary database and tables

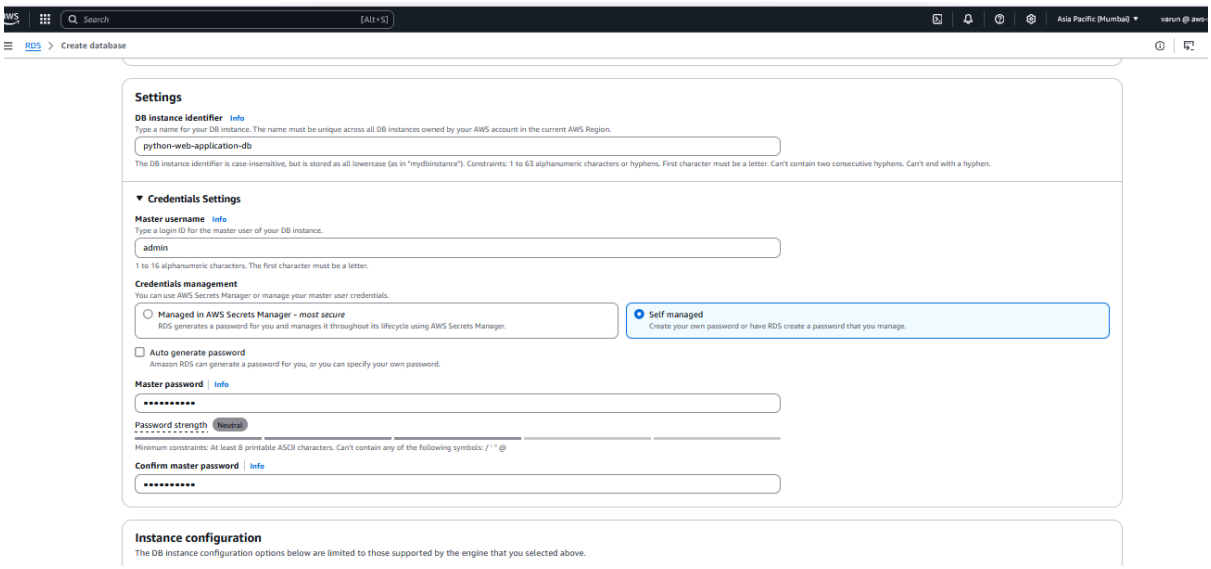
1. Create an **RDS database** to store the application data. Use the **standard method** and select **MySQL** as the database engine based on the requirements.

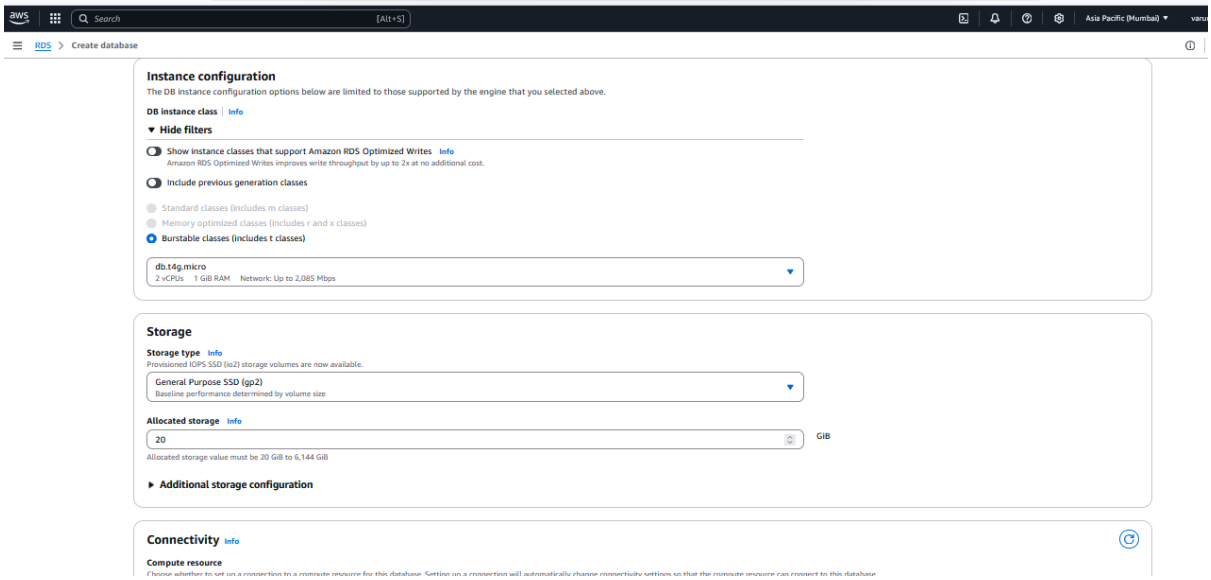


2. Always choose the latest engine version or required for your and select the "Free Tier" template as we are doing for learning purposes.

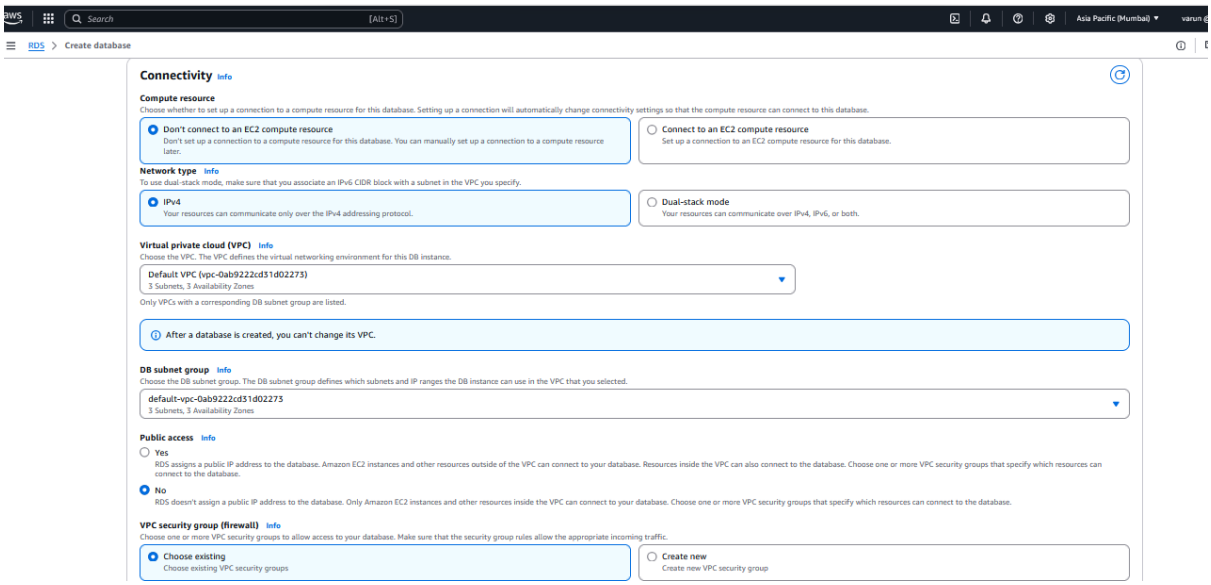


3. Provide the database name and the credentials for logging into the RDS instance. Additionally, configure the CPU, memory, and storage according to the requirements.

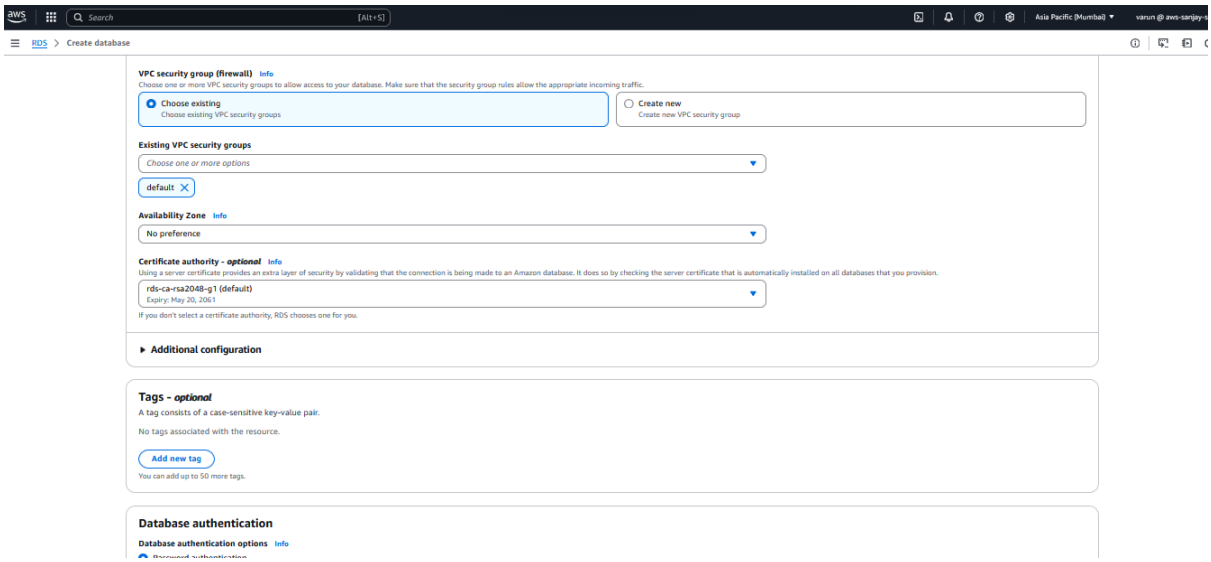


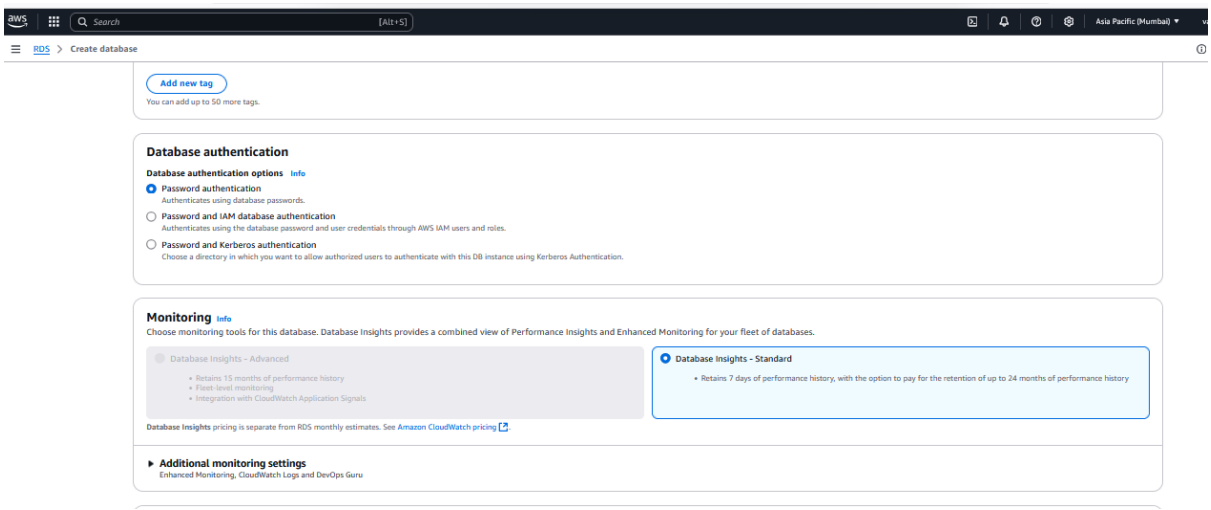


4. For connectivity, select **"Don't connect to an EC2 instance"**, as the goal is to connect the RDS database to **ECS** instead.

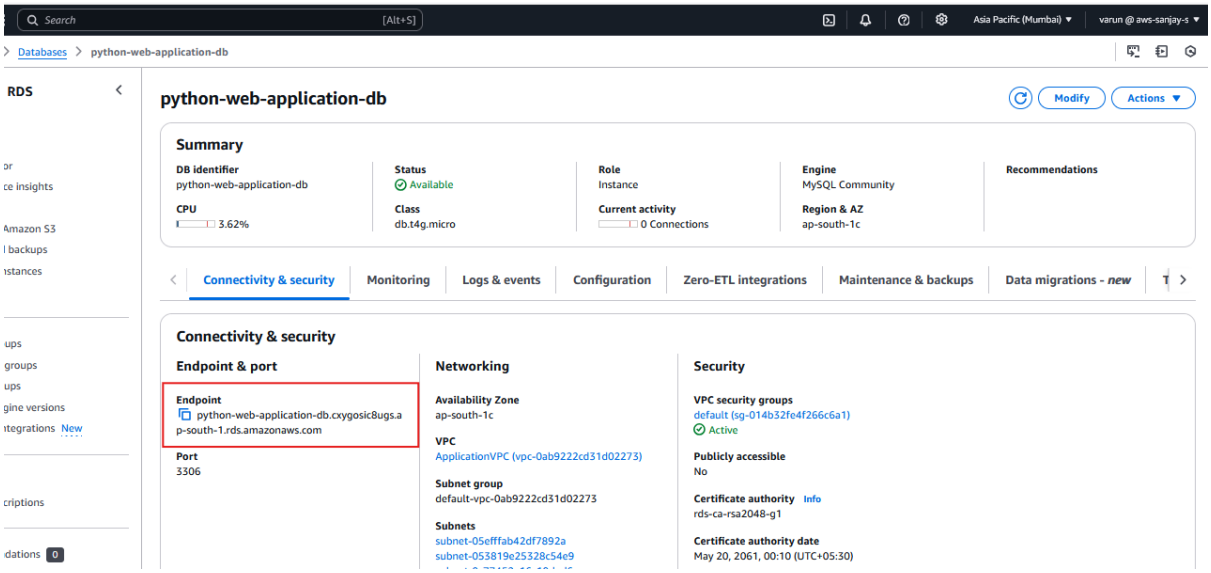


5. Select the **automatic options**, keep the settings at their **default values**, and then click **"Create"** to set up the RDS database.





6. Wait until the **RDS status** changes to **"Available."** Then, **note down the RDS endpoint**, as it will be required for configuring both the **application** and the **container**.



7. Now, log in to the RDS from the server and set up the database along with its tables for the application.
   - mysql -h <RDS_ENDPOINT> -u admin -p
   - Give the password which given while creating the RDS

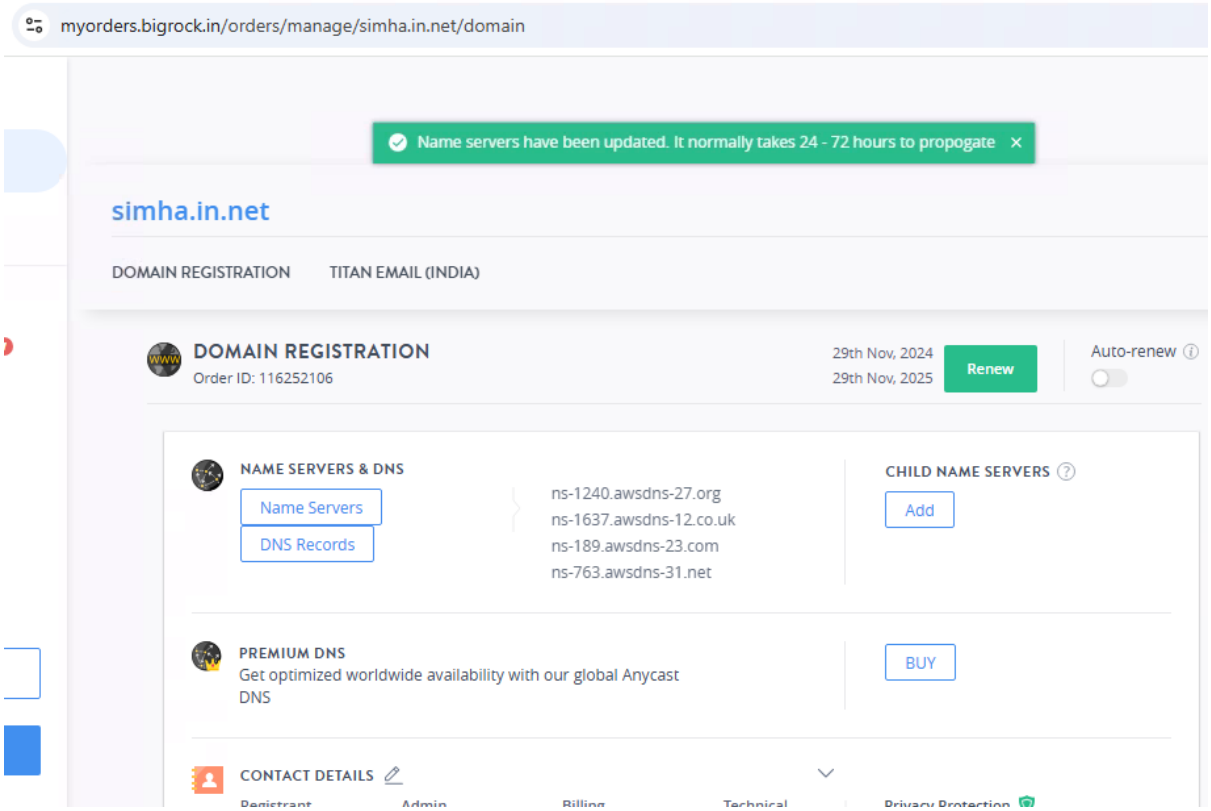
Give the following commands mentioned to below:

```
CREATE DATABASE IF NOT EXISTS django_crud;

USE django_crud;


CREATE TABLE IF NOT EXISTS app_register(
  id INT AUTO_INCREMENT PRIMARY KEY,
  firstname VARCHAR(50) NOT NULL,
  lastname VARCHAR(50) NOT NULL,
  email VARCHAR(50) NOT NULL,
  contact BIGINT
);
```

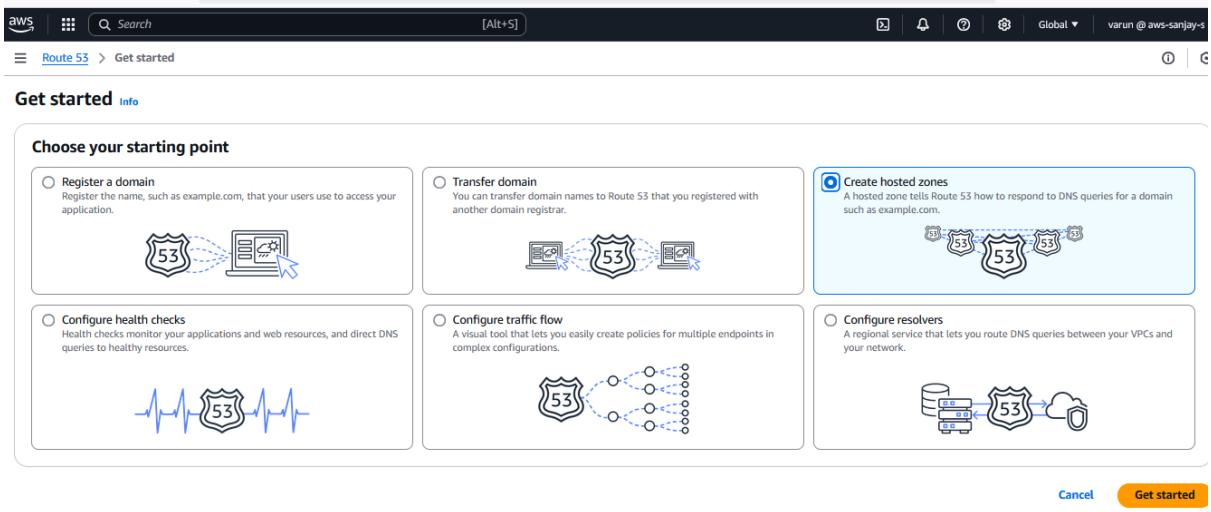
Module 03

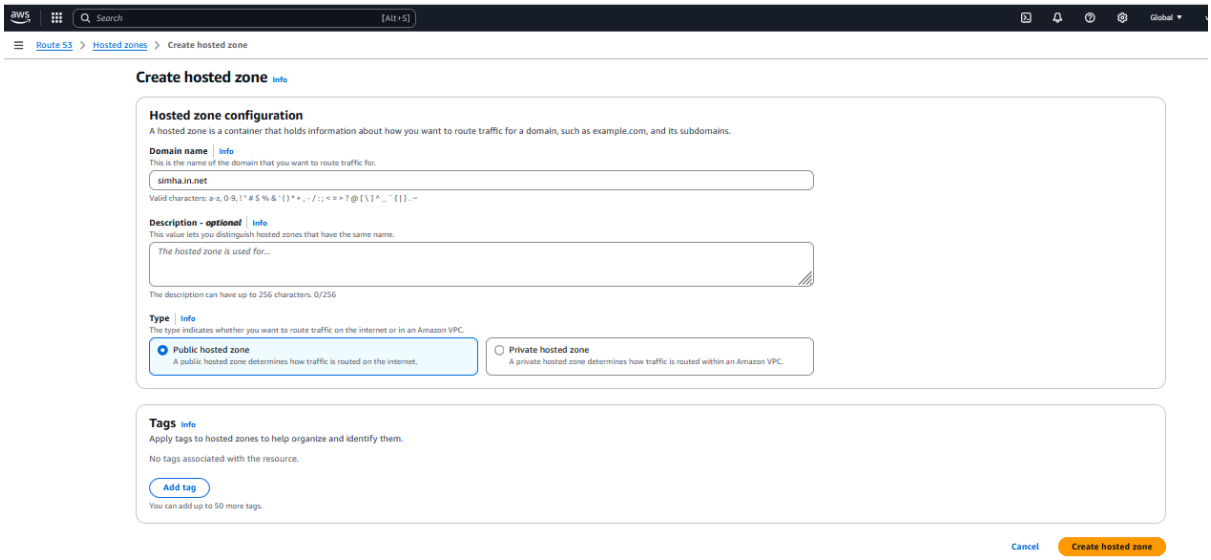
Setting up a secure web application using Route 53, a Load Balancer, and ACM

1. Purchased or registered a domain (in this case, it was purchased through BigRock, a third-party provider).

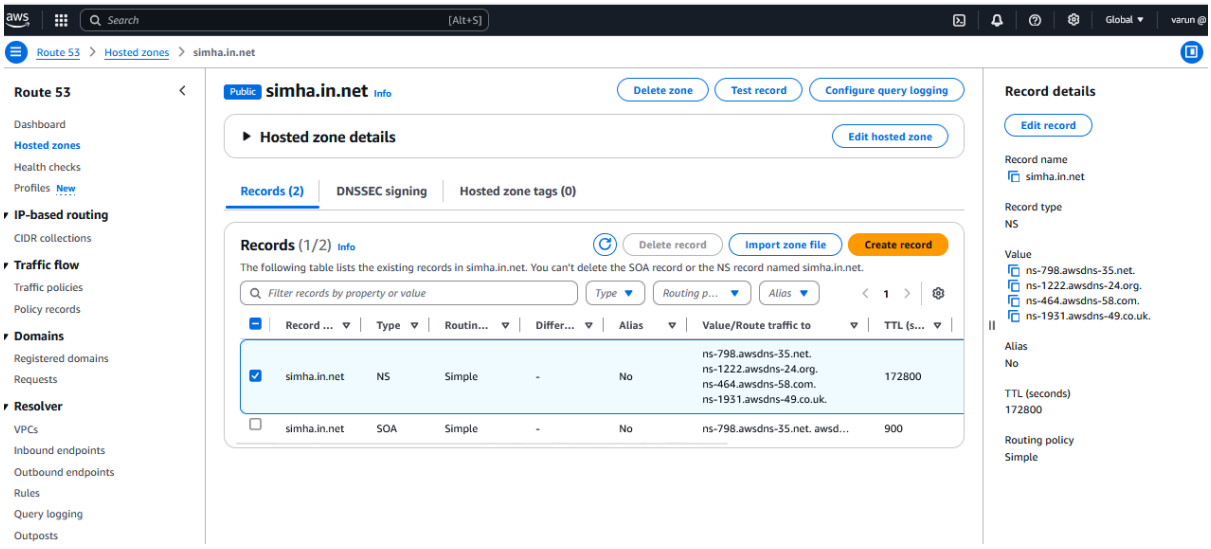


2. Create a hosted zone on Route 53 by providing the domain name you purchased and selecting the type as Public Hosted Zone.

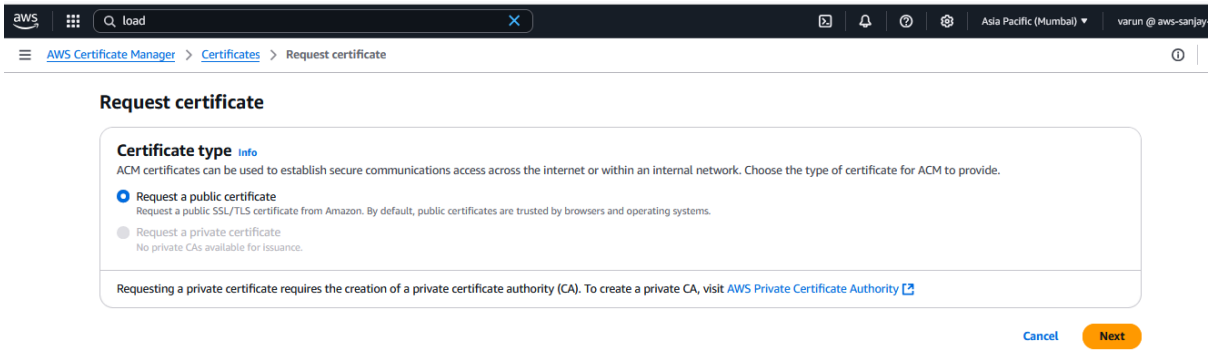


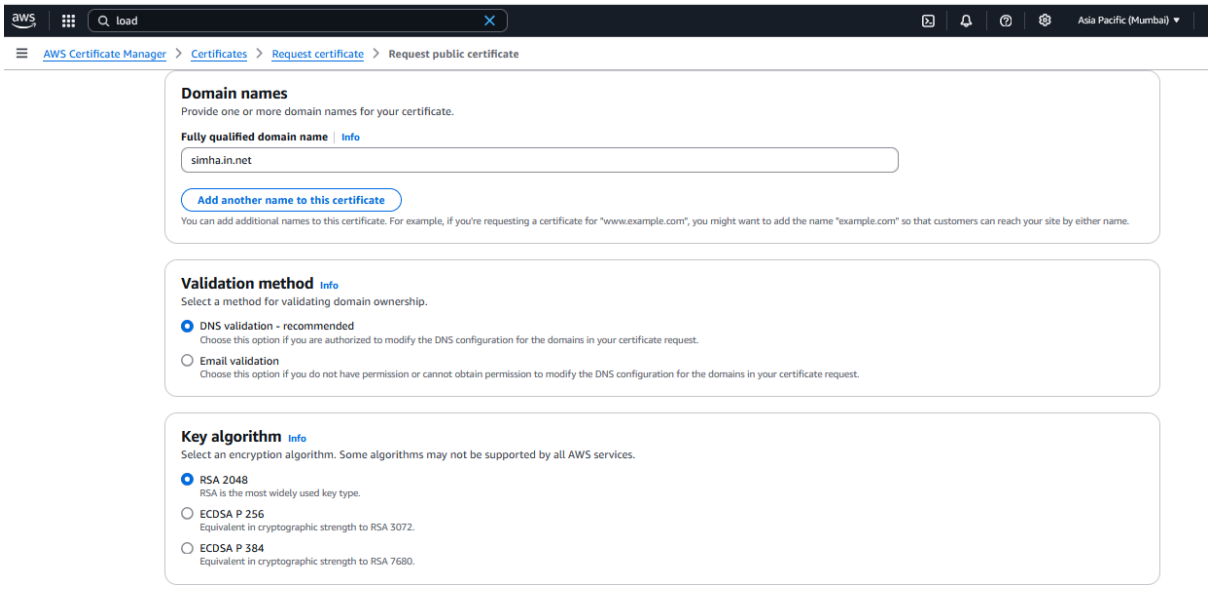


3. After creating the hosted zone, copy the NS (Name Server) record values and paste them into the Name Server settings where you purchased the domain

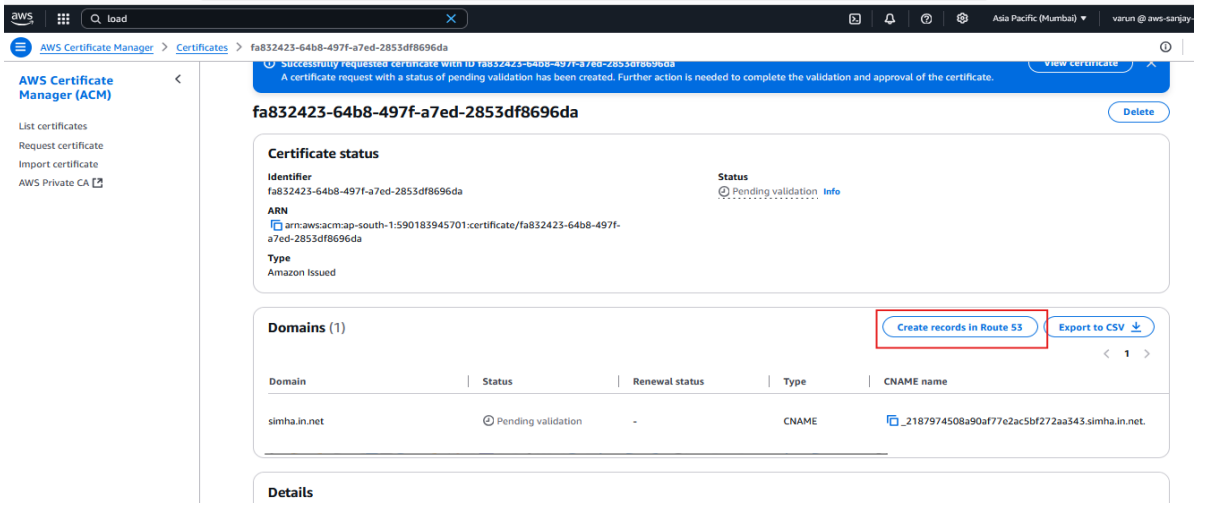


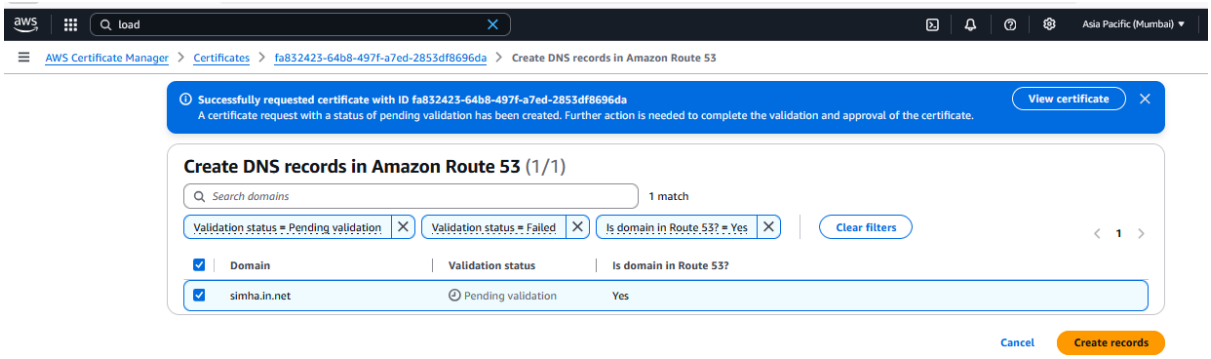
4. Now, create a certificate in ACM for the region where the public instance is located. Choose the certificate type as Public and leave the remaining settings as default

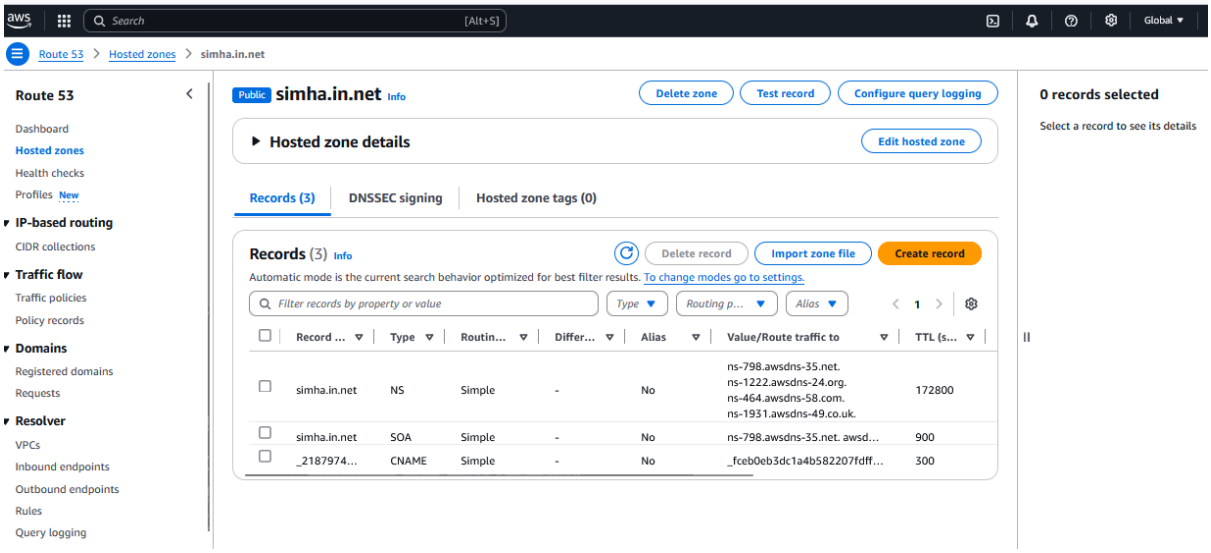




5. After the certificate is created, copy the CNAME record and then add it to Route 53.

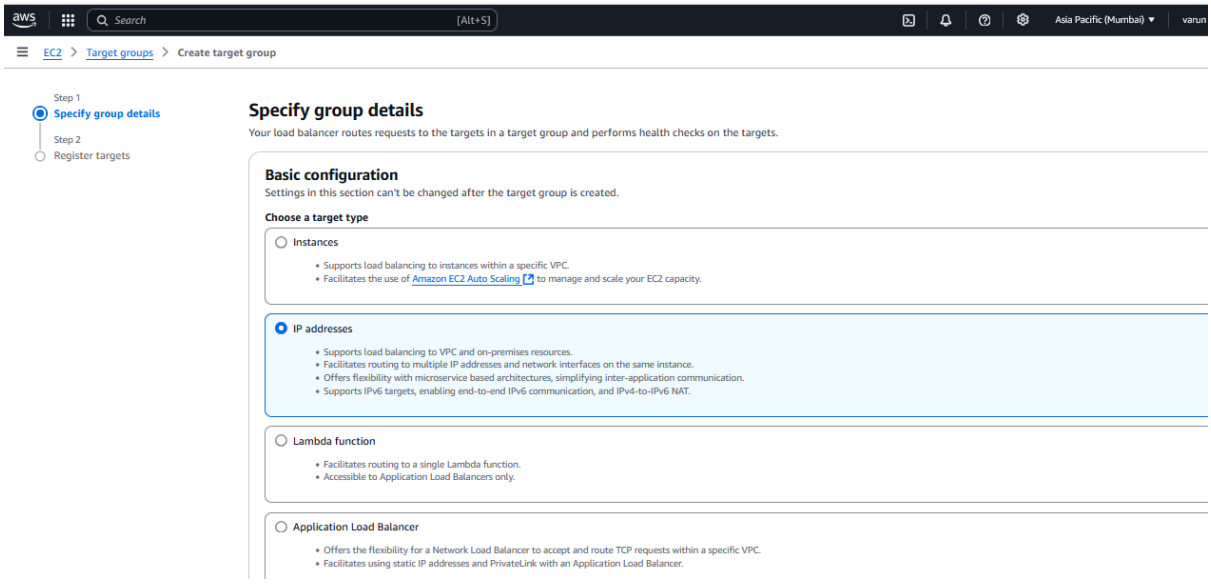




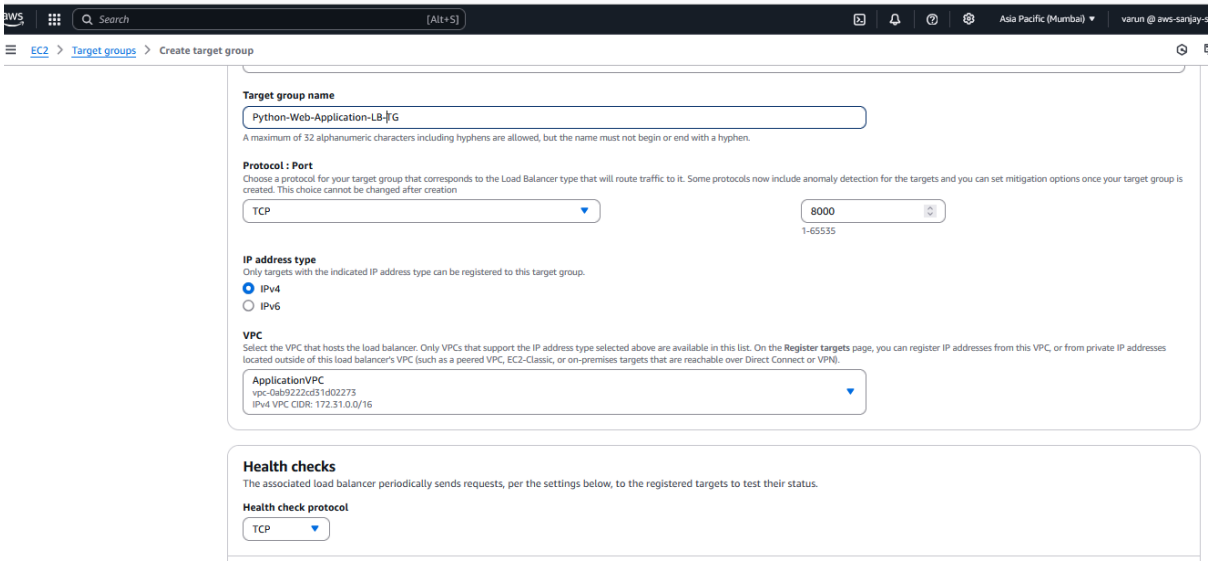


Wait a few minutes for the validation to complete, and the status in the **ACM service** will change to **"Issued."**

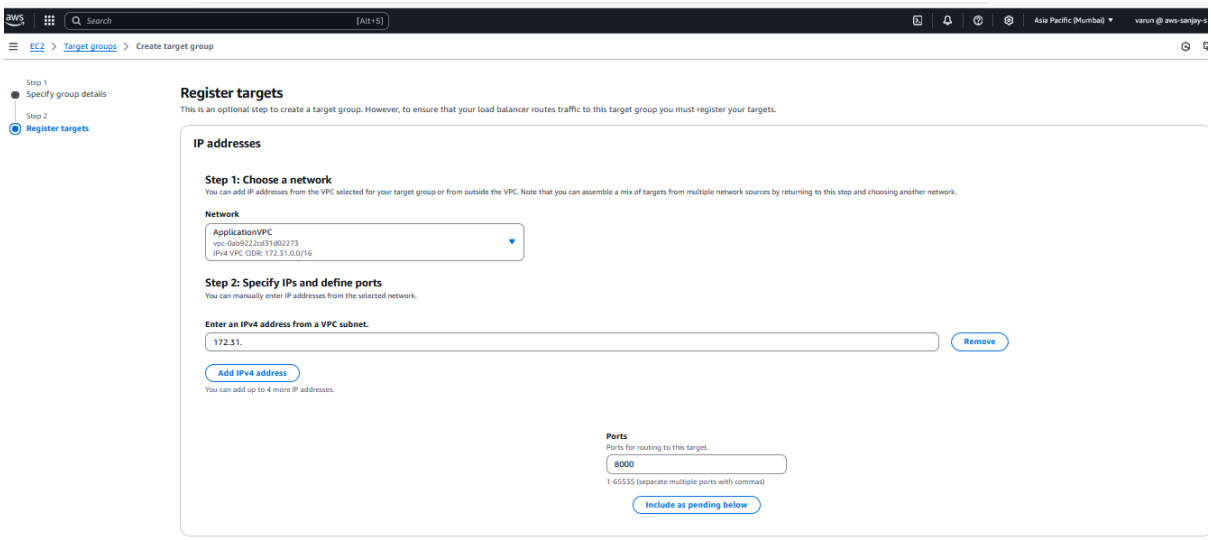
6. Create a **Target Group** for the **Load Balancer**, selecting **"IP Address"** as the target type since we are using **serverless Fargate**.

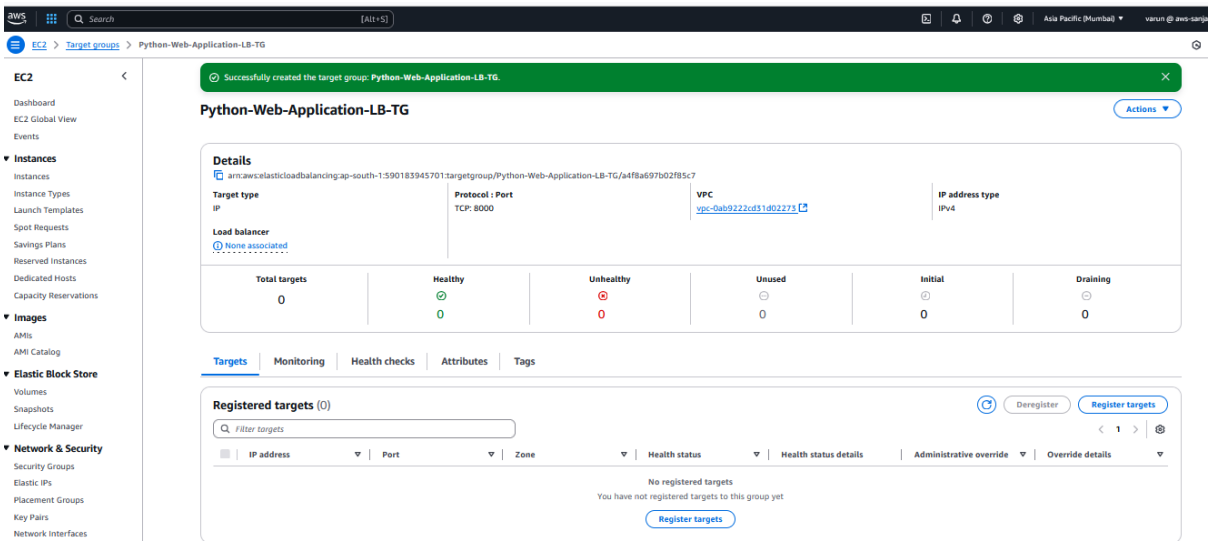


7. Set the **protocol** to **TCP** and the **port** to **8000**, which is exposed/open in the target container.

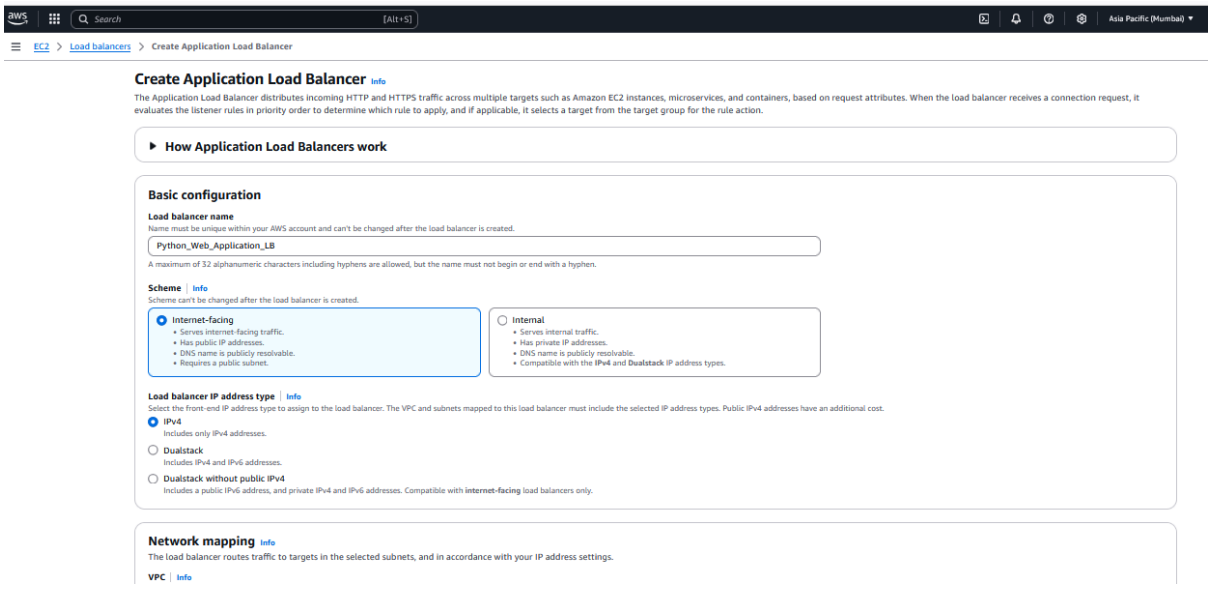


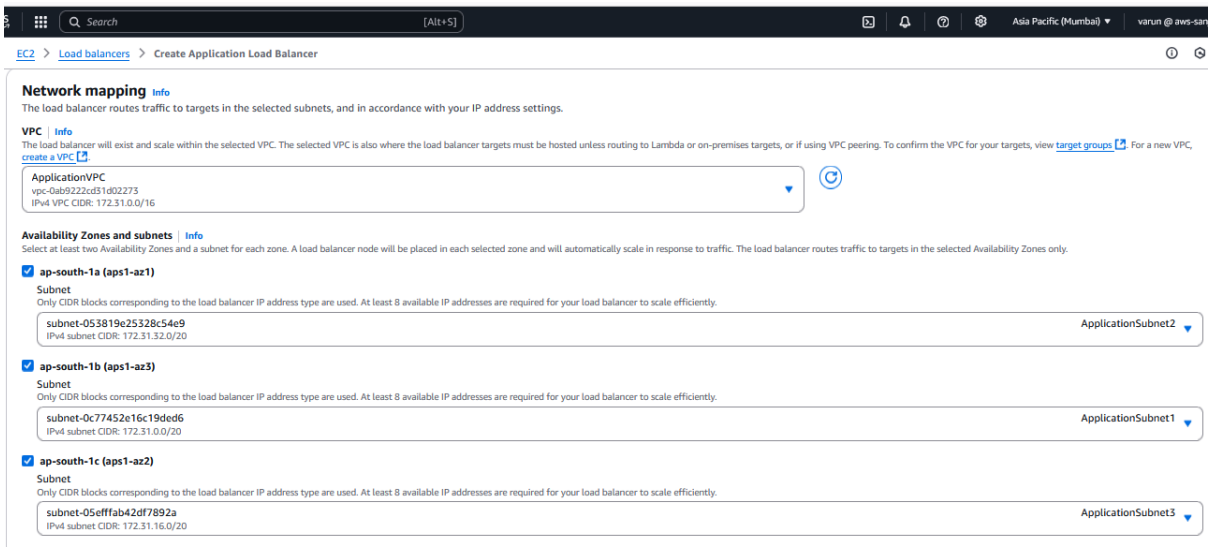
8. In the **"Register Target"** section, specify the **subnet** that needs to be targeted.



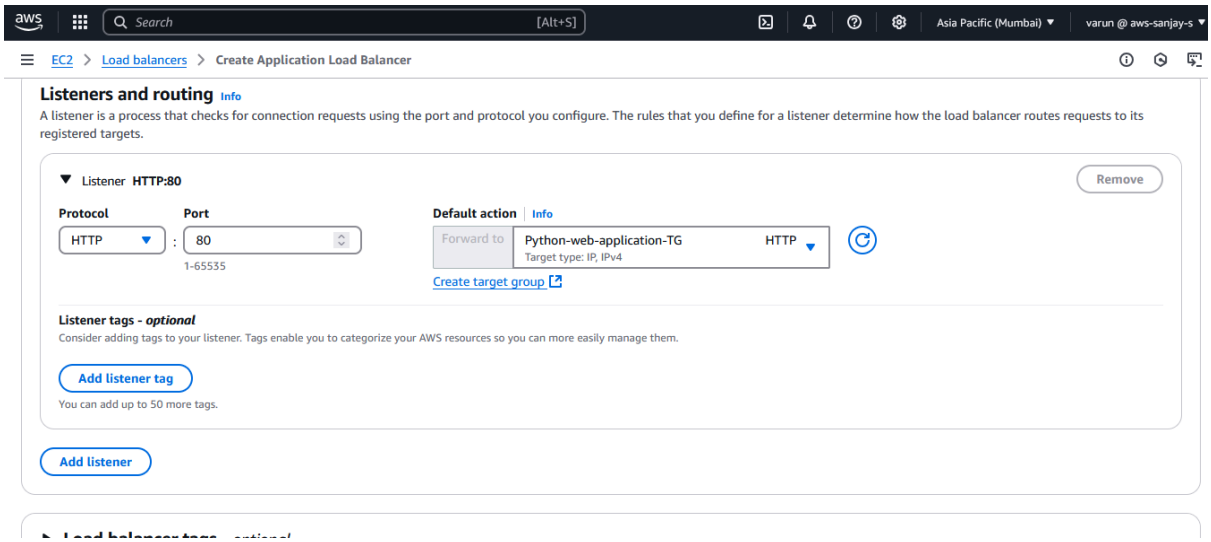


9. After creating the **Target Group**, set up an **Application Load Balancer (ALB)** to enable secure communication with the instance.
    - Configure the **ALB** as **internet-facing**.
    - Select the **VPC** and its **three availability zones** where the container resides.

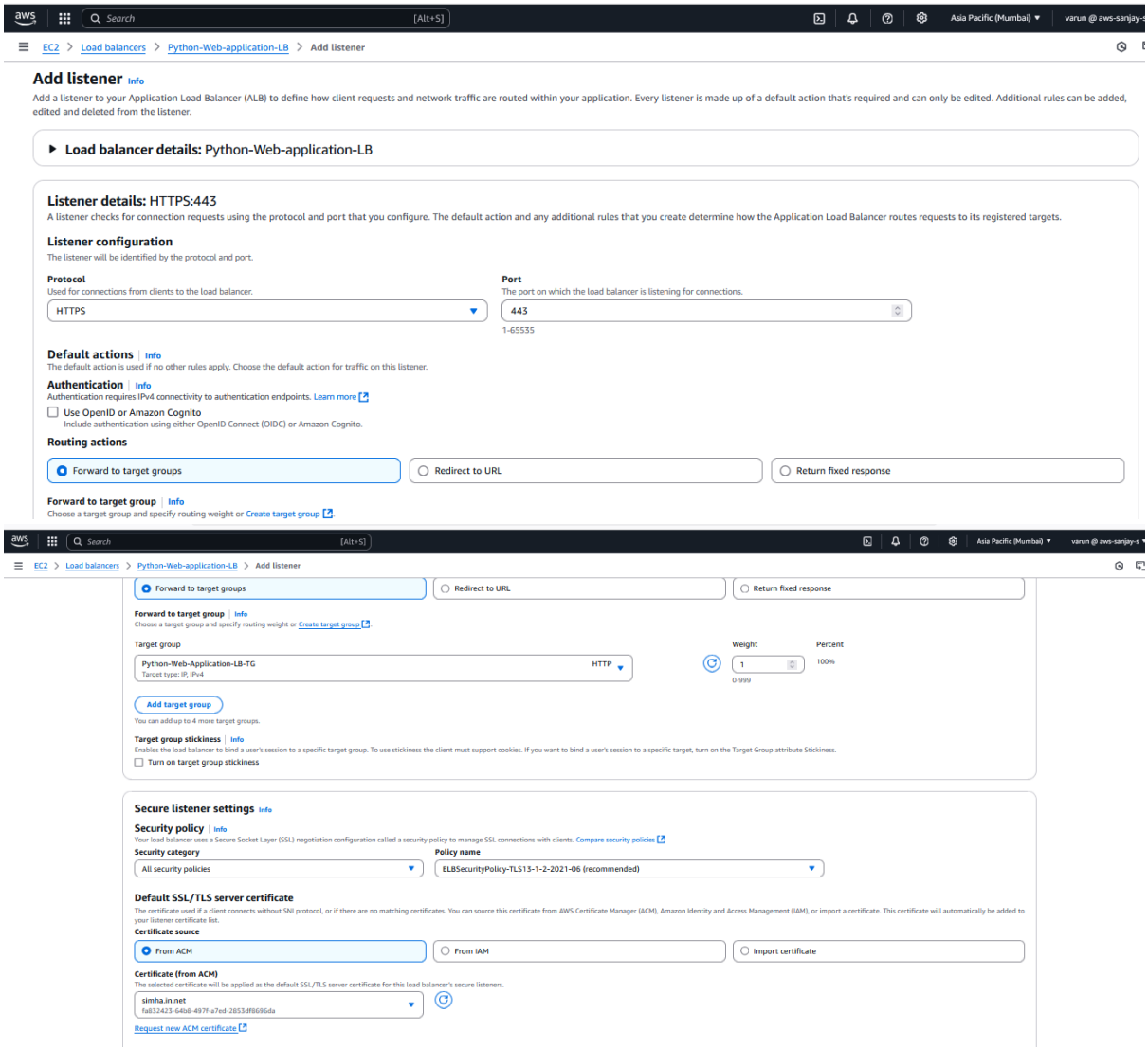




10. In the **Listeners and Routing** section:
    - Add **HTTP** listeners.
    - Set the **default action** for both listeners to **"Forward to Target Group"**.

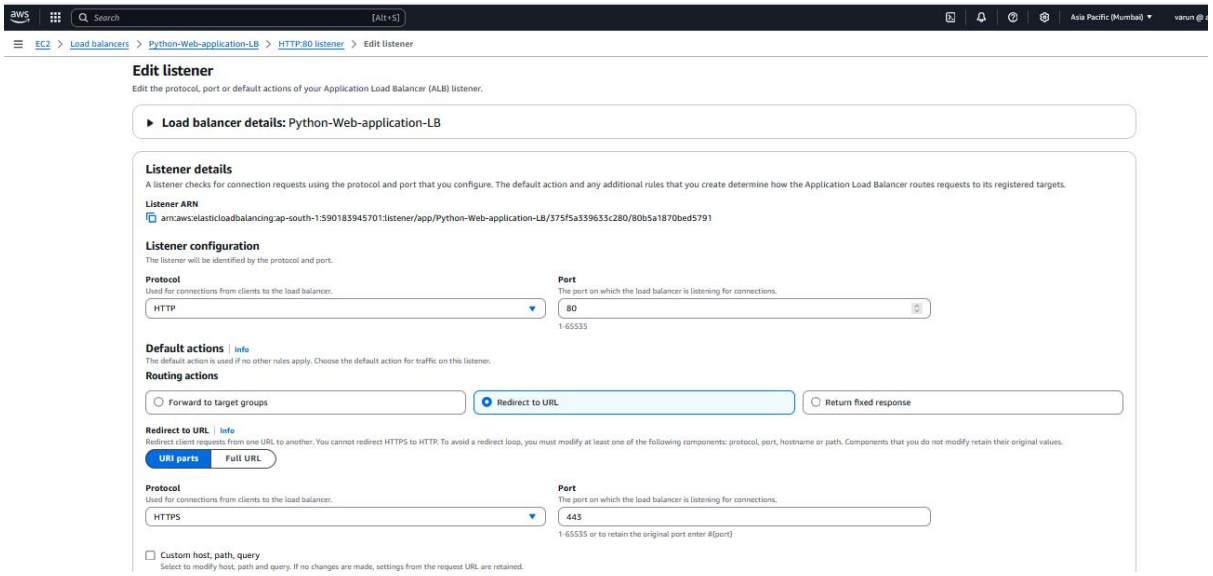


11. After creating the **Load Balancer**:
    - **Add another HTTPS listener**.
    - Choose the **default security policy**.
    - For the **certificate**, select it from **ACM** and choose the certificate created earlier.
    - Click **"Create"** to finalize the setup.

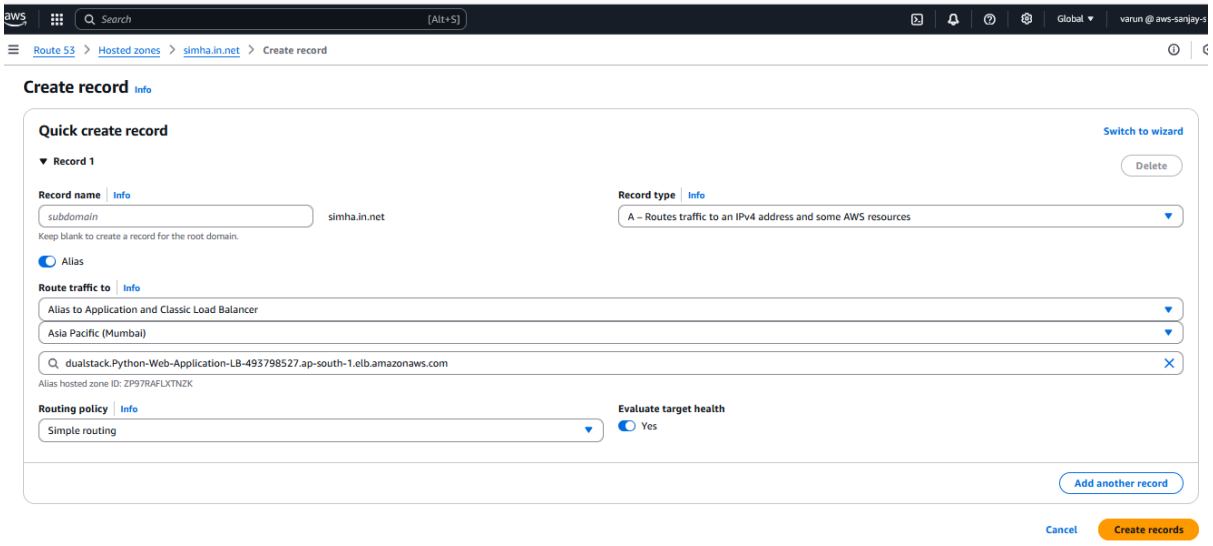


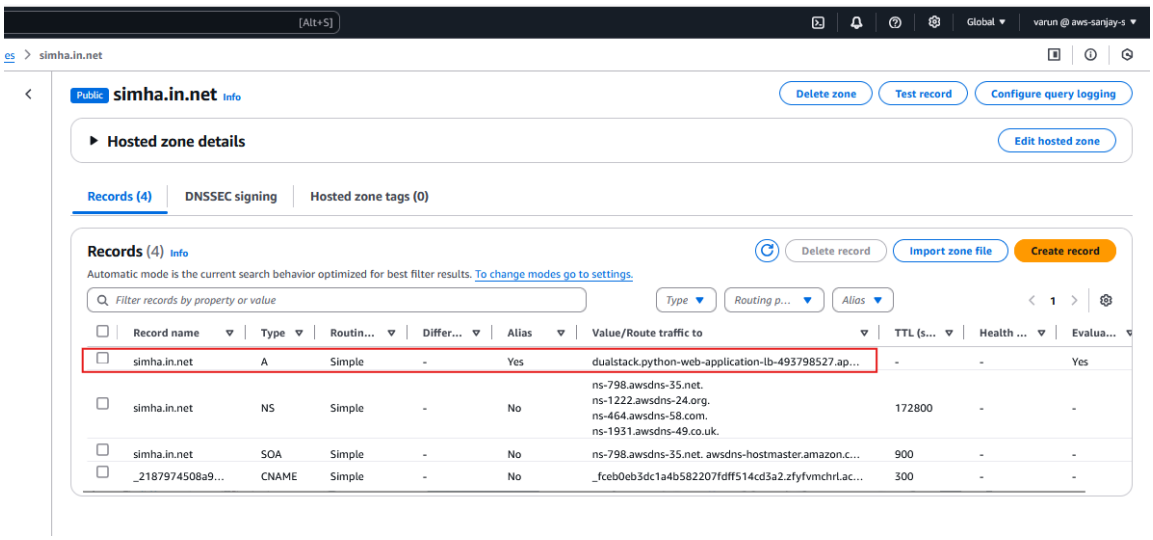
12. Next, **edit the HTTP listener**:
    - Change the **default action** to **"Redirect to URL"**.
    - Set the **URL port** to **HTTPS (443)**.

This ensures that users accessing the application via **HTTP** are automatically redirected to **HTTPS** for a secure connection.



13. Now, in Route 53, create a record for HTTPS by setting the traffic routing value to the application Load Balancer and specifying its region, along with the Load Balancer's DNS name.

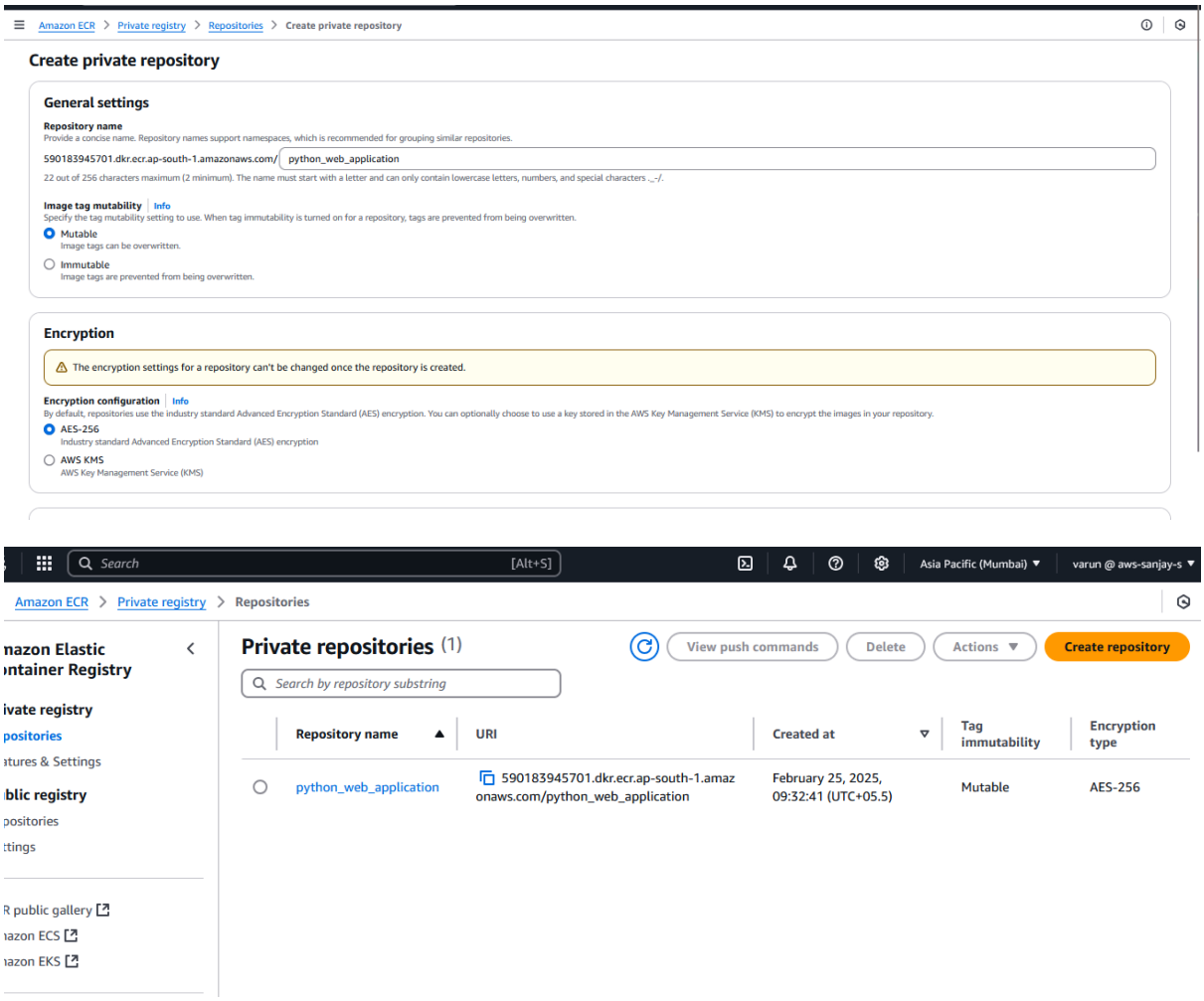




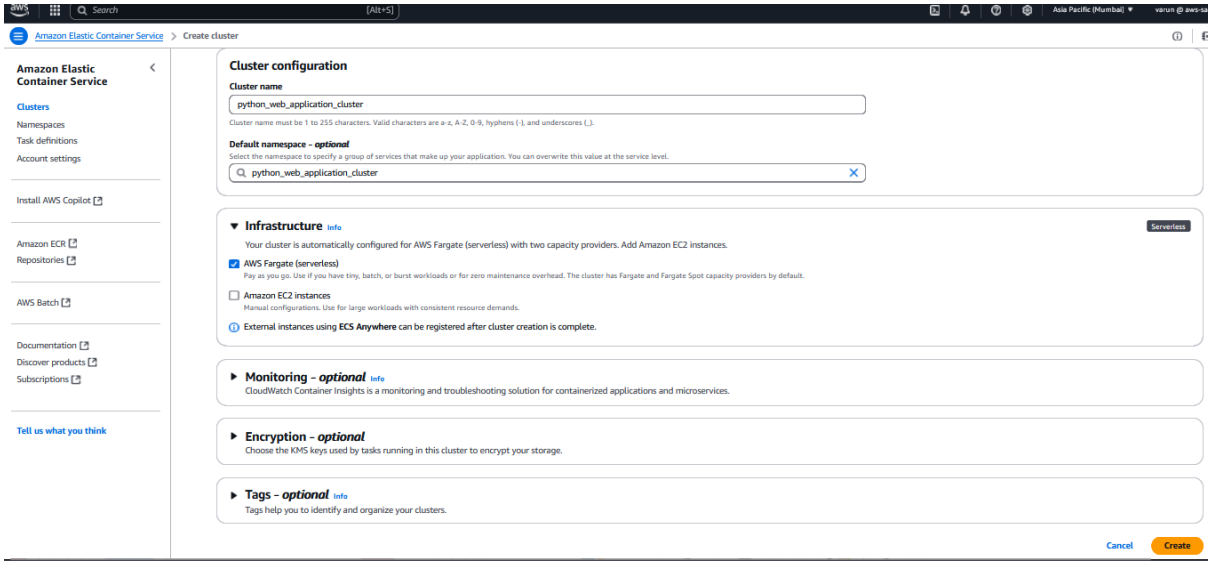
Module – 04

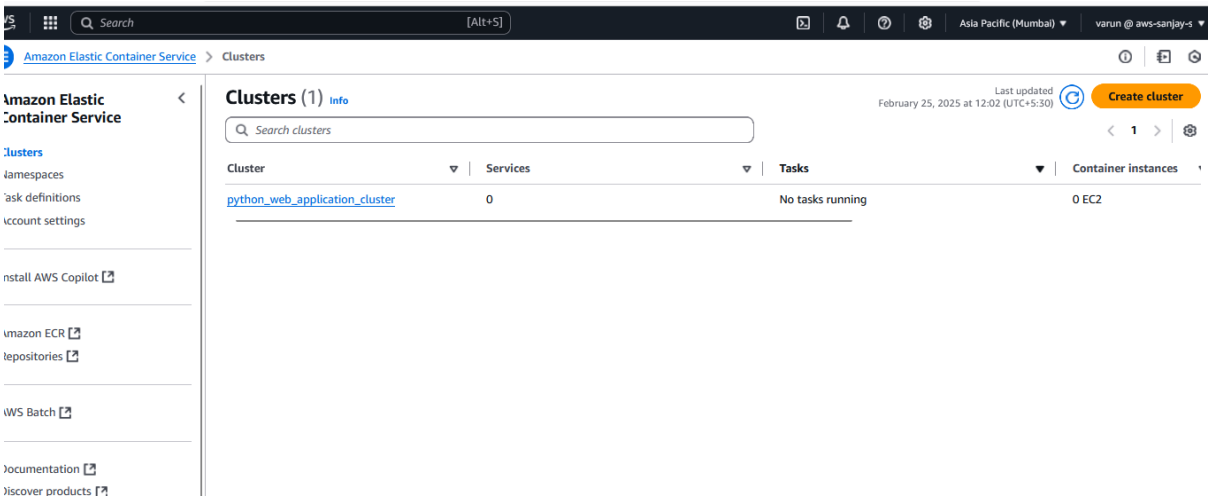
Set up ECR, ECS, task definition, and its service for containerized deployment

1. Create an **Elastic Container Registry (ECR)** on a **private repository**.

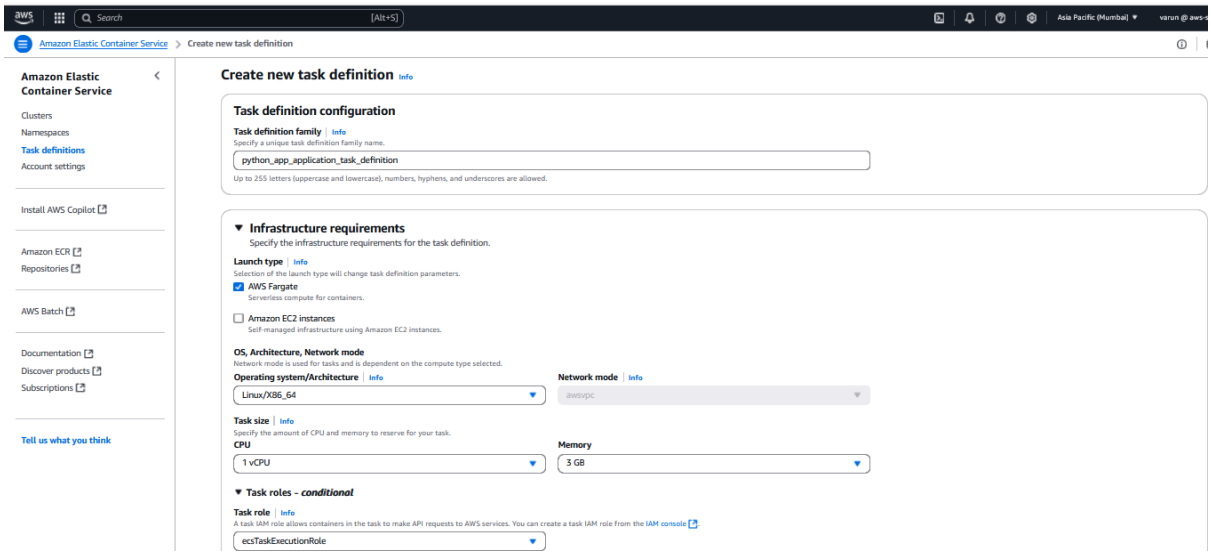


2. Now, create the **Elastic Container Service (ECS) cluster** and select **AWS Fargate** as the infrastructure since we are using a **serverless** approach.

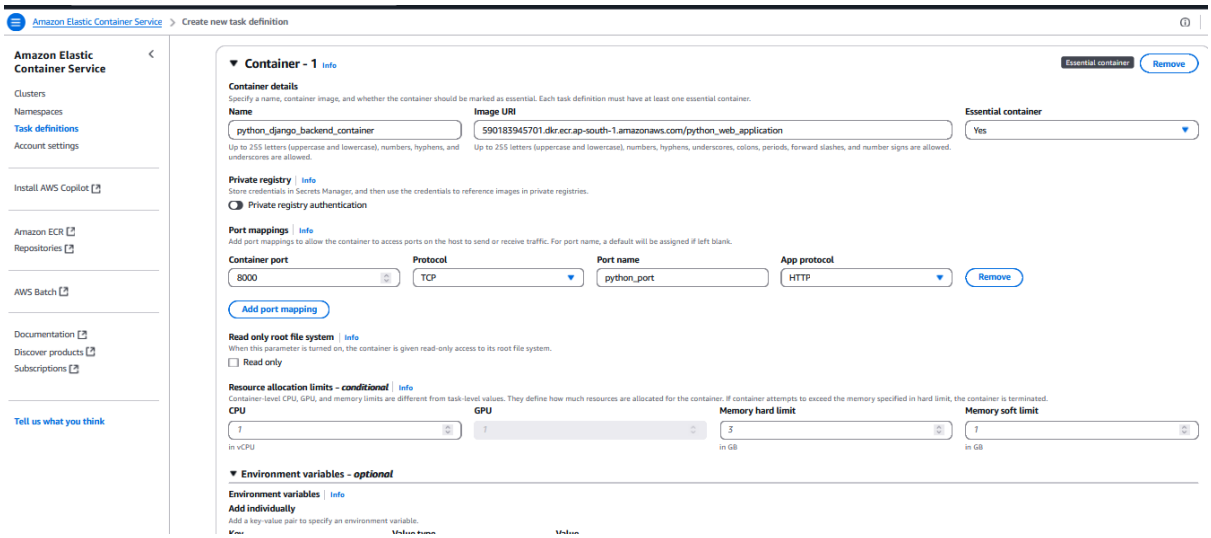




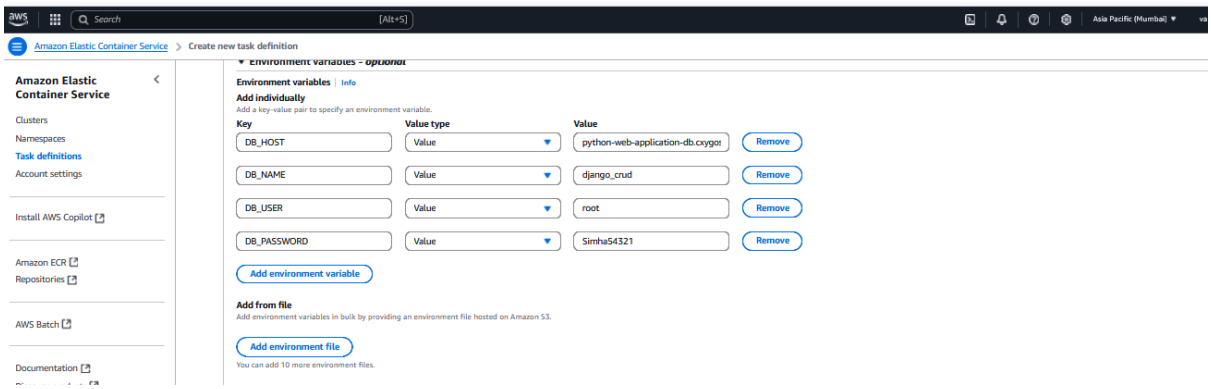
3. After creating the **ECR**, proceed with creating the **Task Definition**:
    - **Launch Type**: **Fargate** (as we are using serverless).
    - **Operating System**: **Linux x86_64**.
    - **Task Size**: Define the required **CPU and memory** for the containers.



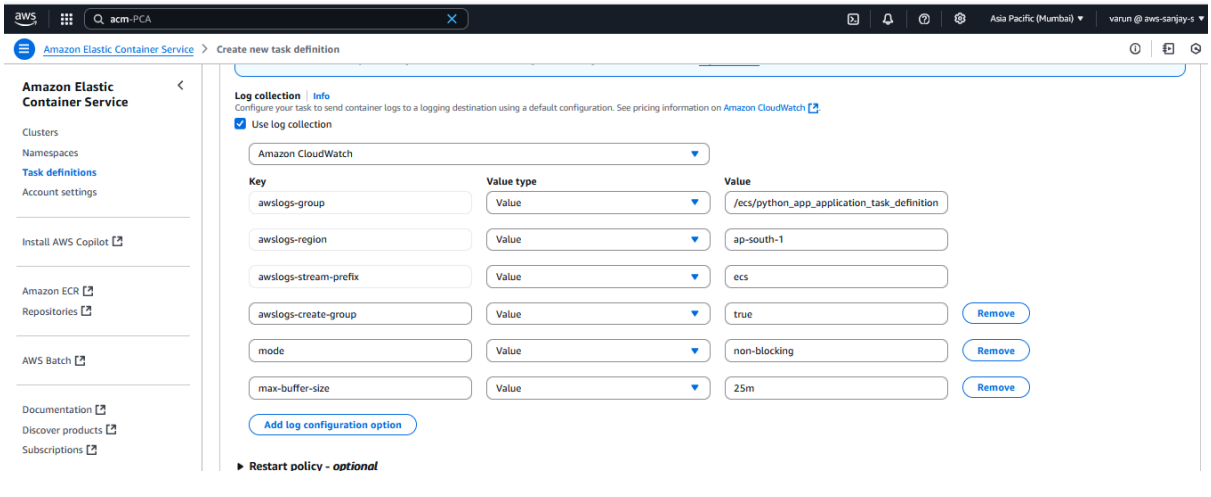
4. Now, configure the **container details**:
    - **Image URI**: Use the **ECR URI** created earlier.
    - **Container Port**: **8000** (as exposed by the application).
    - **Application Protocol**: **HTTP**.

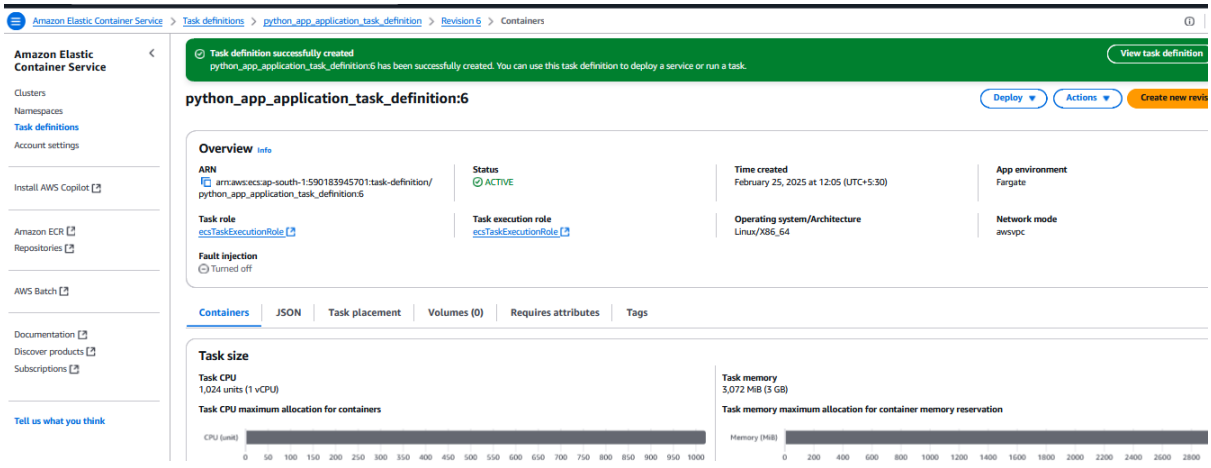


5. Set up the **environment variables** to enable communication between the **container** and **RDS** or other containers:
    - **DB_HOST** = *RDS Endpoint* (created earlier)
    - **DB_NAME** = *django_crud* (database created on RDS)
    - **DB_USER** = *Username set during RDS creation*
    - **DB_PASSWORD** = *Password set during RDS creation*

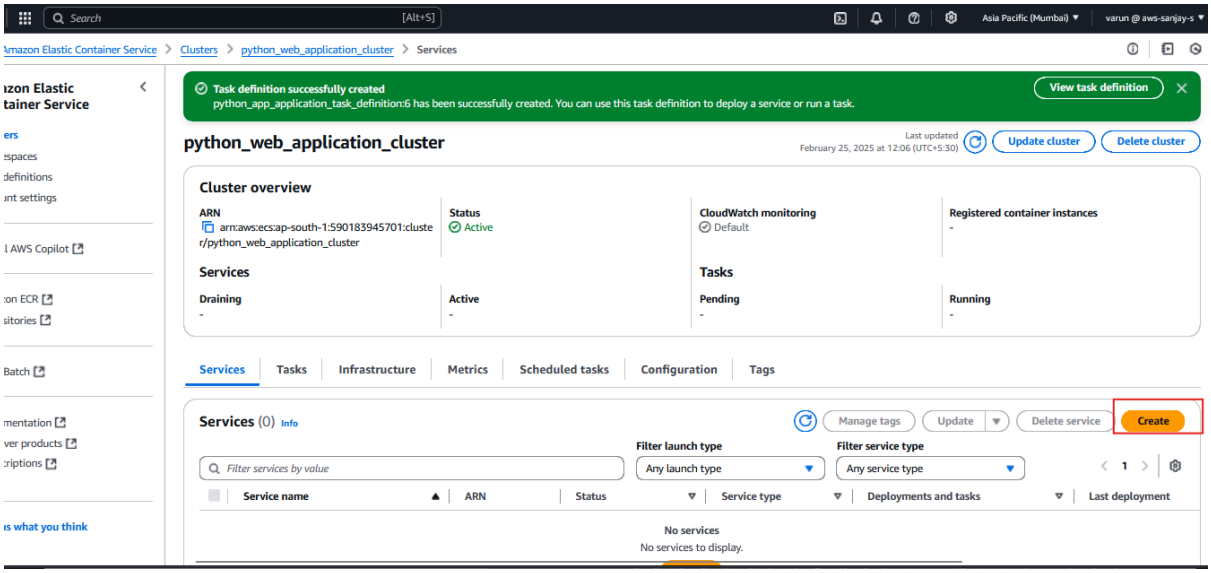


6. **Enable log collection** to store **application container logs** in an **Amazon CloudWatch Logs group**.

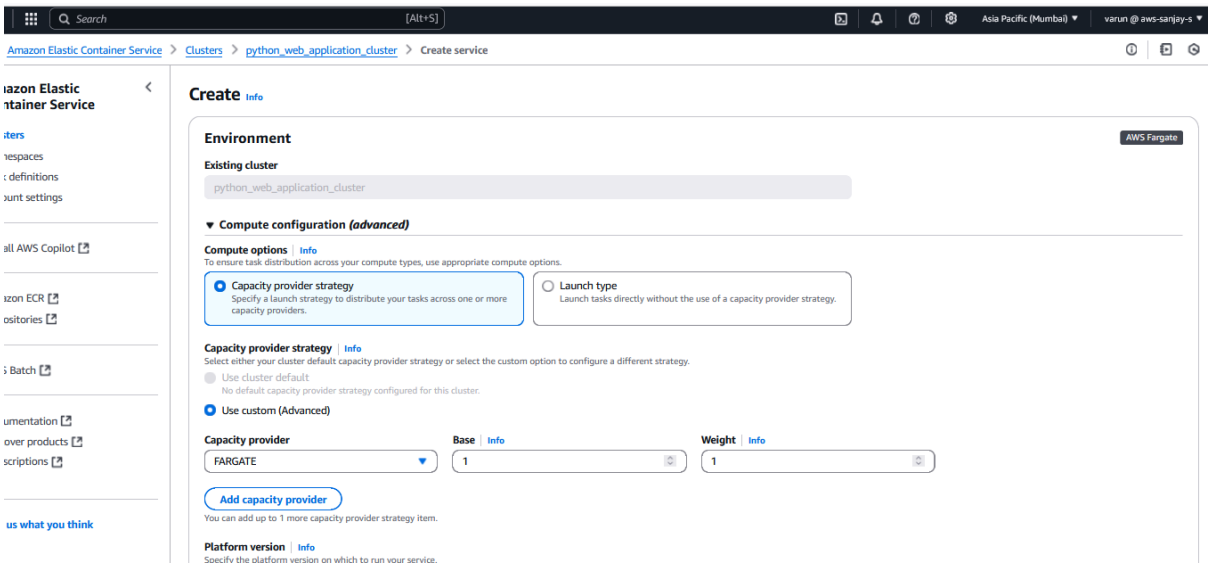




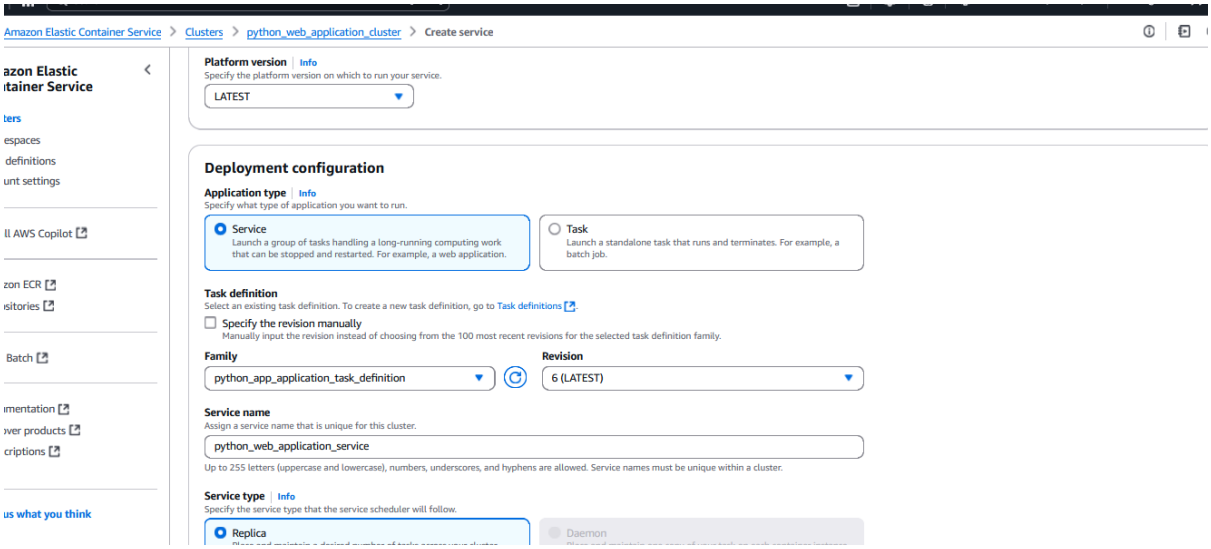
7. Once the **Task Definition** is created, proceed with setting up the **Service**:
   - Go to the **ECS Cluster**.
   - Click on **"Create Service"** to begin the setup.



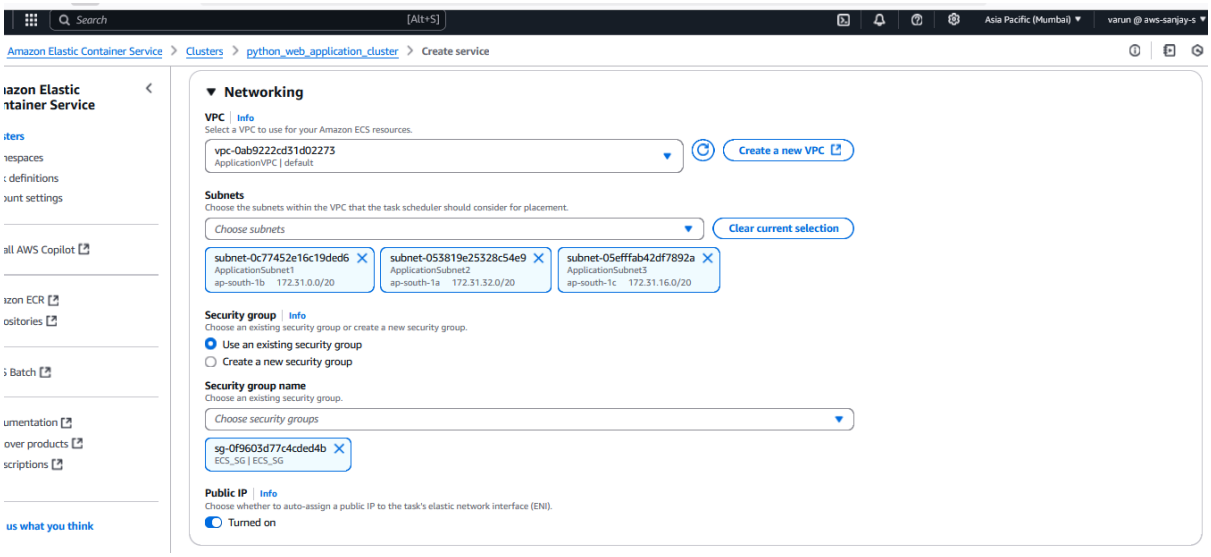
8. Set the **Capacity Provider** to **FARGATE** and set the **Base Value** to **1**, ensuring that at least **one task** is always running.



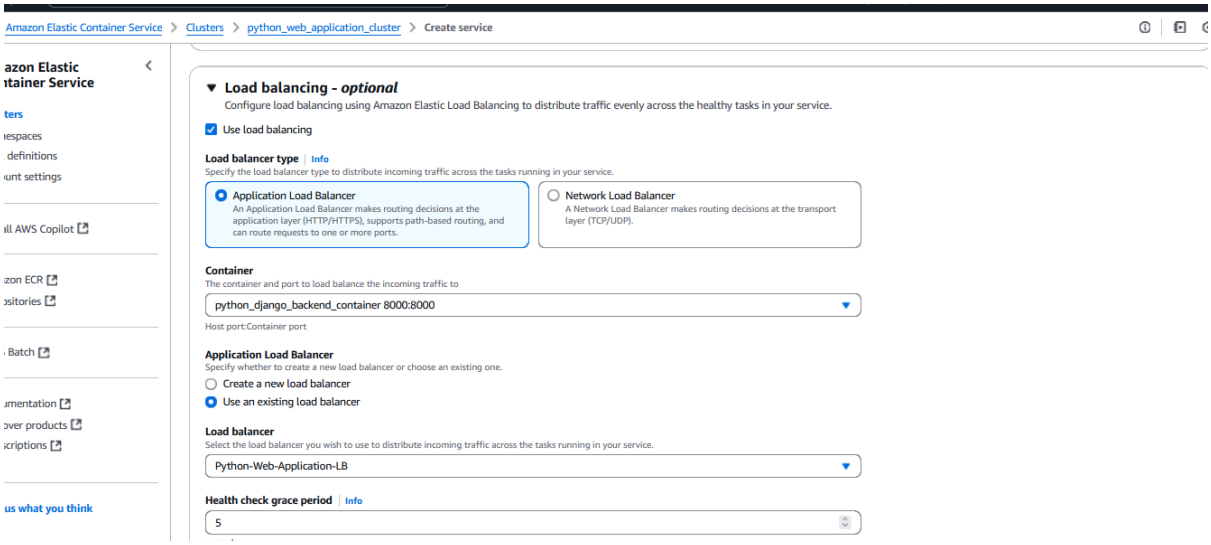
9. Set the **Platform Version** to **"LATEST"**, as the **ECR** stores only the **latest image** for deployment.



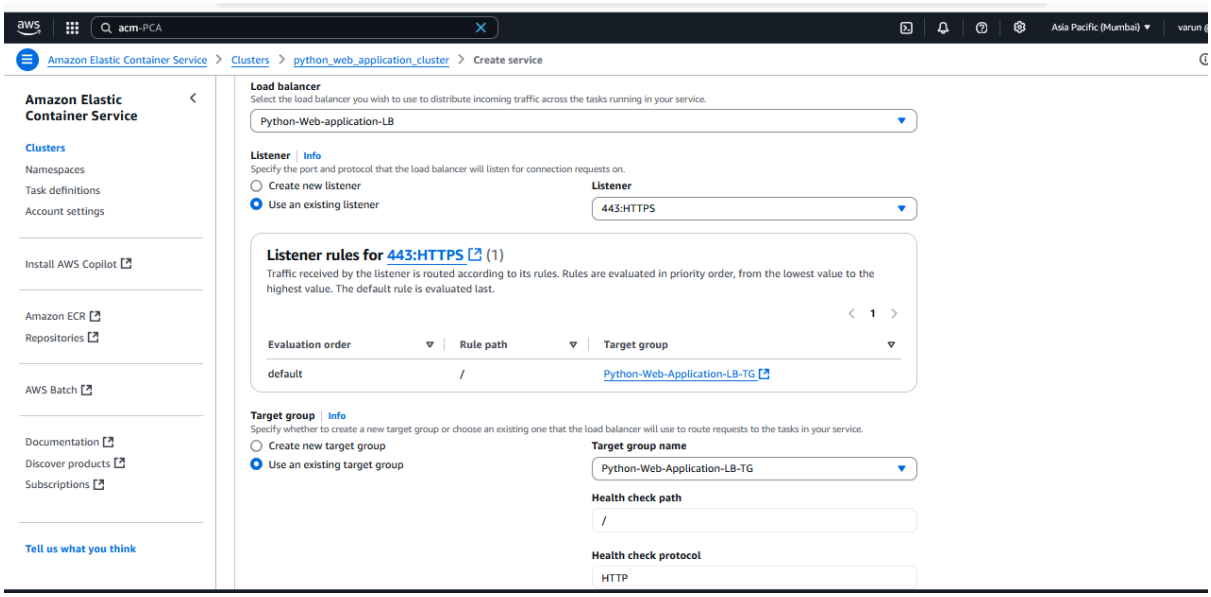
10. Keep the **remaining settings as default** and select the **same VPC** where the **RDS** and **EC2 instance** are set up. This ensures seamless connectivity without needing additional configurations.



11. Now, select the **Application Load Balancer (ALB)** that was created earlier.



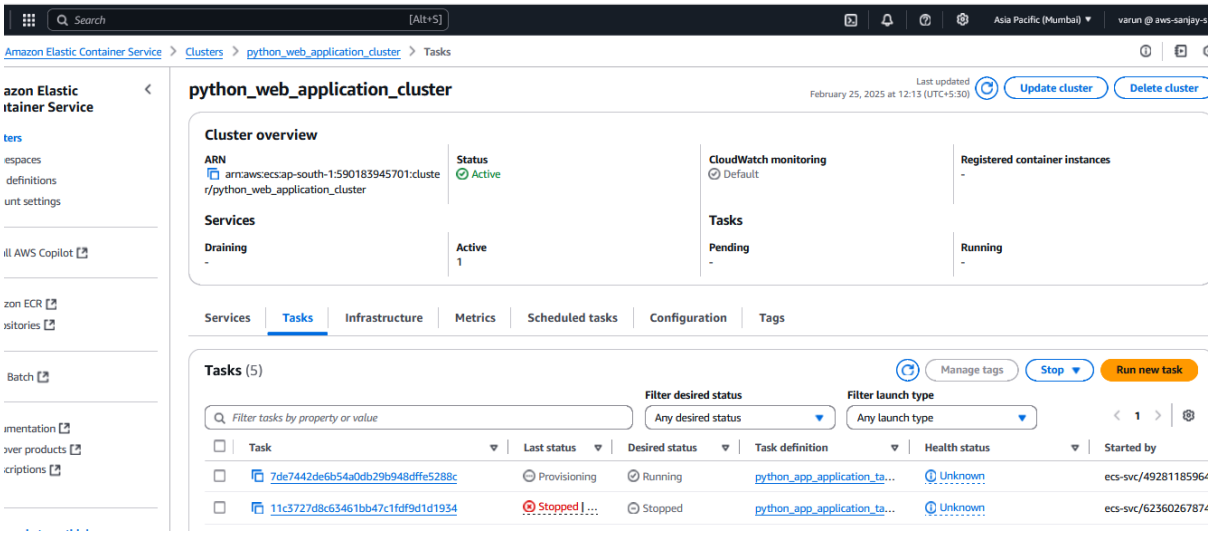
12. Here, select **HTTPS** instead of **HTTP** to ensure secure communication.



Note: Initially, the task will **fail** since no image is stored in **ECR**.

To prevent unnecessary failures:

- **Stop the running task** under the **Task** section in ECS.


Search
[Alt+S]
Asia Pacific (Mumbai)
Amazon Elastic Container Service
Clusters
python_web_application_cluster
Tasks
python_web_application_cluster
Last updated
February 25, 2025 at 12:13 (UTC+5:30)
Update cluster
Delete cluster
Cluster overview
ARN
arn:aws:ecs:ap-south-1:590183945701:cluster/python_web_application_cluster
Status
Active
CloudWatch monitoring
Default
Registered container instances
-
Services
Draining
-
Active
1
Tasks
Pending
-
Running
-
Services
Tasks
Infrastructure
Metrics
Scheduled tasks
Configuration
Tags
Tasks (5)
Manage tags
Stop
Run new task
Filter tasks by property or value
Filter desired status
Any desired status
Filter launch type
Any launch type
Task
Last status
Desired status
Task definition
Health status
Started by
7de7442de6b54a0db29b948dffe5288c
Provisioning
Running
python_app_application_ta...
Unknown
11c3727d8c63461bb47c1fdf9d1d1934
Stopped
Stopped
python_app_application_ta...
Unknown

Module – 05

Overview of the Web Application and Implementing Necessary Modifications

Before building and pushing the container, let's go through an **overview of the application**.

1. The **GitHub repository** contains a **Python web application** structured as follows:

```
├── docker-compose.yml
├── backend
│   ├── App
│   │   ├── Templates
│   │   │   ├── edit.html
│   │   │   ├── insert.html
│   │   │   ├── show.html
│   │   ├── Migrations
│   │   ├── admin.py
│   │   ├── apps.py
│   │   ├── models.py
│   │   ├── tests.py
│   │   ├── urls.py
│   │   ├── views.py
│   ├── Basic
│   │   ├── settings.py
│   │   ├── urls.py
│   │   ├── wsgi.py
├── Dockerfile
├── manage.py
```

This structure includes:

- **Docker-related files** (docker-compose.yml, Dockerfile) for containerization.
- **Backend application** containing **Django models, views, templates, and configurations**.
- **Templates folder** for HTML views.
- **Settings and URLs** for application configuration.

Docker-compose.yml:

```
version: '3.8'
services:
  backend:
    build: ./backend
    ports:
      - "8000:8000"
    environment:
      - DB_HOST= RDS-EndPoint( Which is created earlier)
      - DB_NAME=django_crud
      - DB_USER=Admin
      - DB_PASSWORD=Simha
    networks:
      - my_networks

networks:
  my_networks:
    driver: bridge
```

dockerfile (inside the bankend directory):

```
FROM python:3.9

WORKDIR /app

RUN apt-get update

COPY . /app

RUN pip install --no-cache-dir django mysqlclient

EXPOSE 8000

CMD ["python", "manage.py", "runserver", "0.0.0.0:8000"]
```

---

2. Now, modify the necessary configurations in the **settings.py** file.
- The application should be accessible from anywhere.
- Update the ALLOWED_HOSTS in the settings.py file as follows:

ALLOWED_HOSTS = ['*', 'localhost', '127.0.0.1']

3. The application should only accept **CSRF tokens** from the specified domains. Update the **CSRF_TRUSTED_ORIGINS** in the **settings.py** file:

```
CSRF_TRUSTED_ORIGINS = [
  "https://simha.in.net",
  "https://www.simha.in.net"  # Add both with and without "www"
]
```

4. Update the **database connection settings** in the **settings.py** file to connect to the **RDS MySQL database**:

```
DATABASES = {
  'default': {
    'ENGINE': 'django.db.backends.mysql',
    'NAME': os.environ.get('DB_NAME', 'django_crud'),
    'USER': os.environ.get('DB_USER', 'admin'),
```

```
        'PASSWORD': os.environ.get('DB_PASSWORD', 'Simha54321'),
        'HOST': os.environ.get('DB_HOST', 'python-web-application-db.cxygosic8ugs.ap-south-1.rds.amazonaws.com'),
        'PORT': os.environ.get('DB_PORT', '3306'),
    },
}
```

Module – 06

Configure the Jenkins console and set up its pipeline code

1. Once the **Jenkins service** is running successfully on the server, access it using the **public IP address** with port **8080** in the browser.

- **Example:** http://3.108.53.207:8080

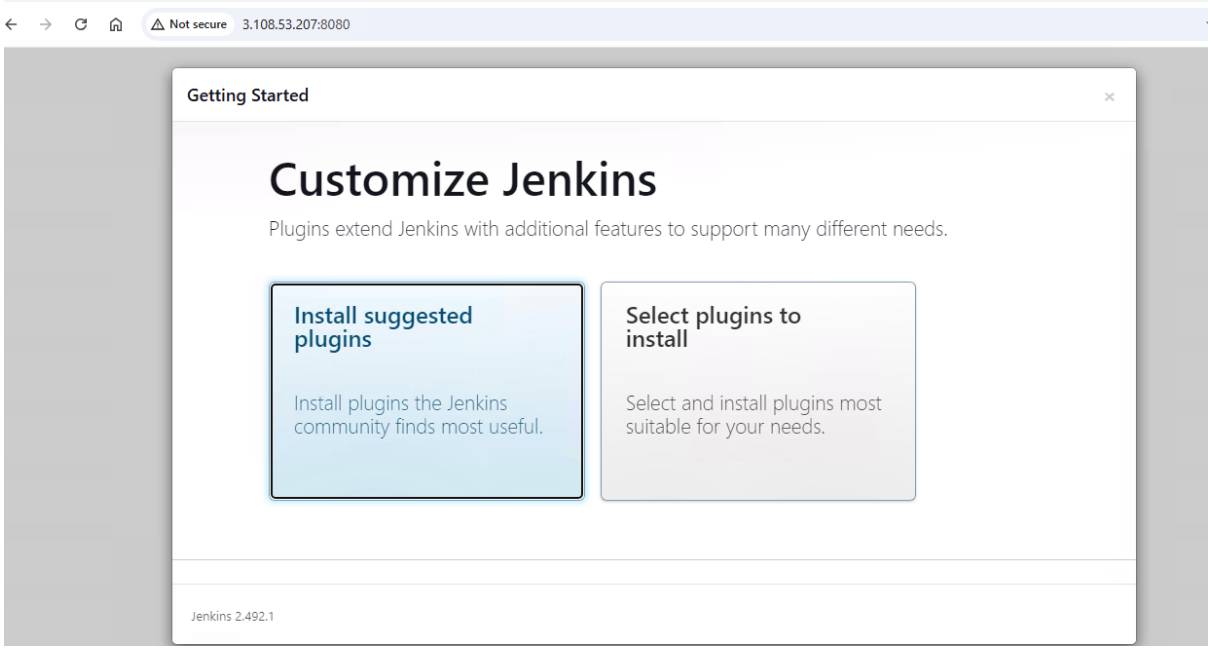
2. Upon initialization, Jenkins will prompt for an **admin password**. Retrieve it using the following command:

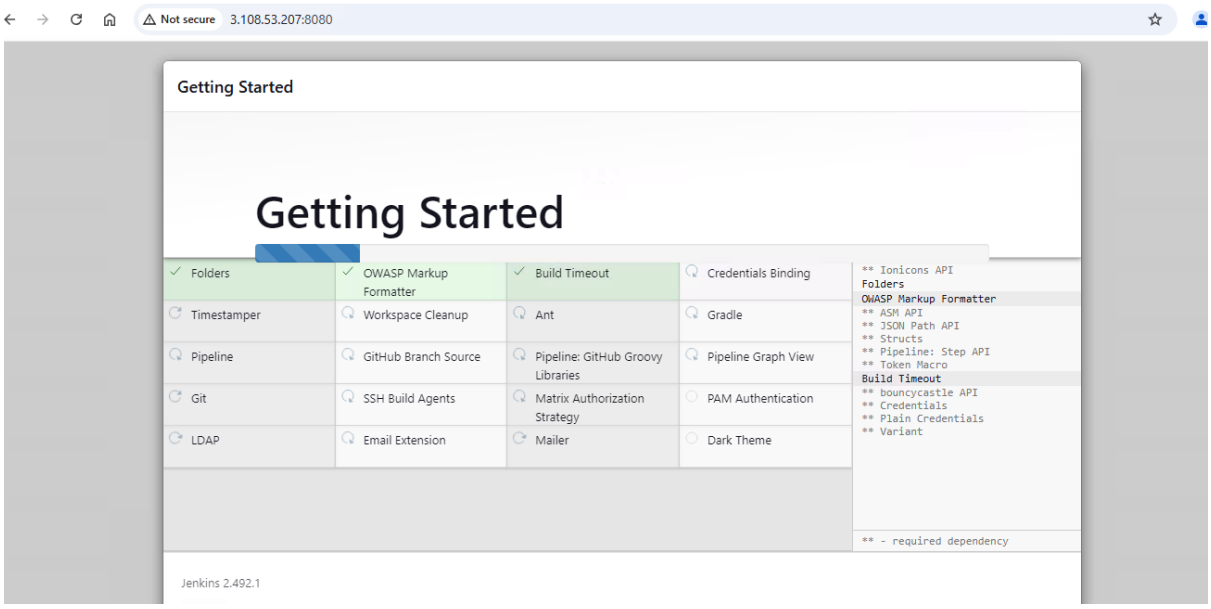
   Retrieve Jenkins Initial Admin Password

   sudo cat /var/lib/jenkins/secrets/initialAdminPassword

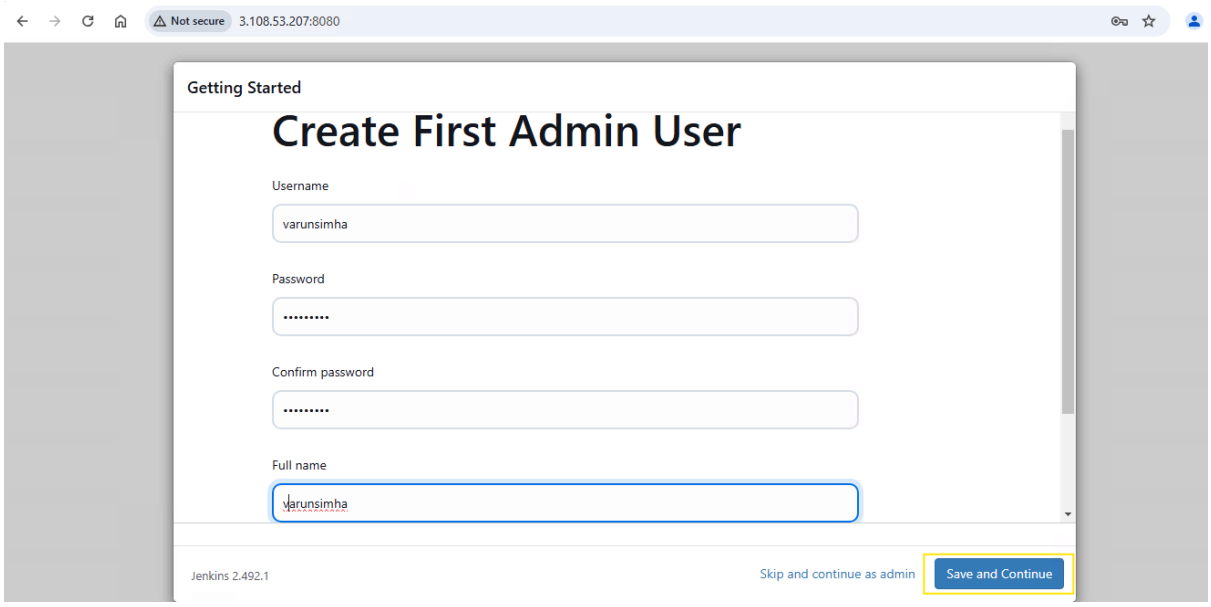
```
root@ip-172-31-7-152:/my_project# sudo cat /var/lib/jenkins/secrets/initialAdminPassword
7bb00c327e92411fab36e9509f05abd9
root@ip-172-31-7-152:/my_project#
```

3. Next, install the **suggested plugins** by clicking on the **"Install suggested plugins"** option during the Jenkins setup.





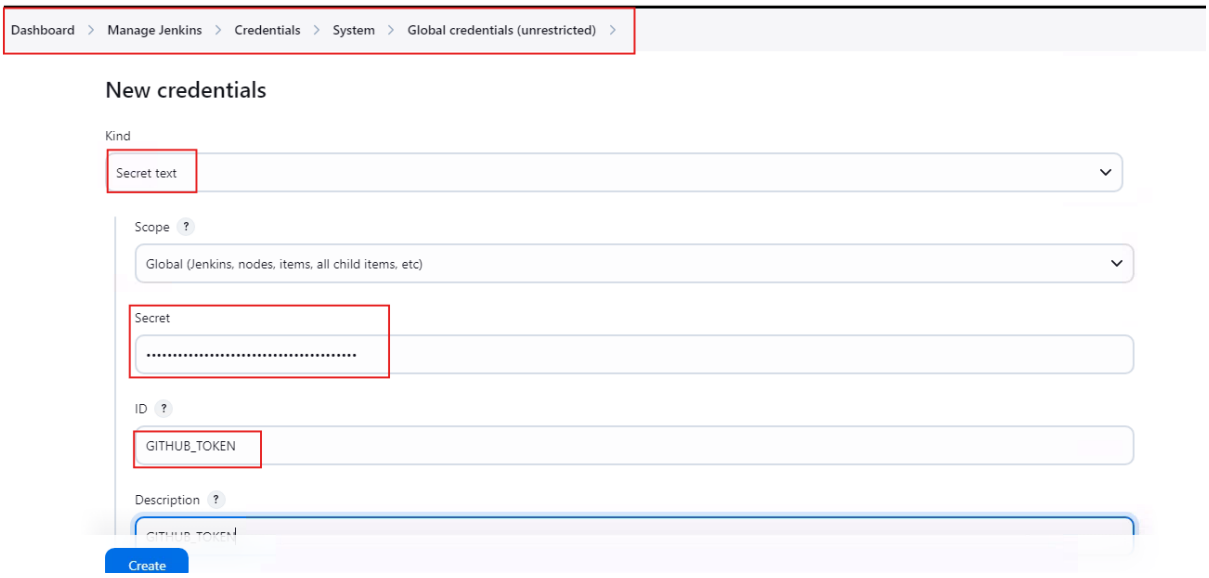
4. After the installation, enter the **required details** such as **username, password, full name, and email**, then click on **"Save and Continue"** to proceed.



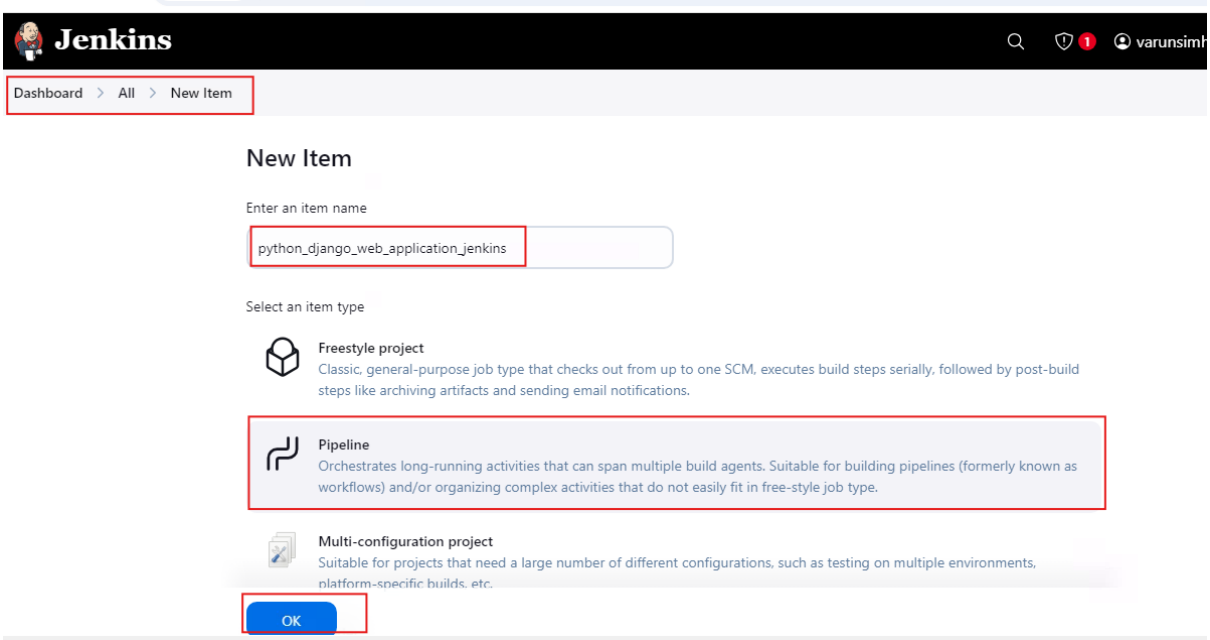
5. Since the GitHub repository is **private**, instead of hardcoding the token in the code, save it in **Jenkins Global Credentials**:

- Go to **Jenkins Dashboard** → **Manage Jenkins** → **Manage Credentials**.
- Under **Stores scoped to Jenkins**, click on **(global)** → **Add Credentials**.
- Choose **"Secret Text"** as the credential type.
- Enter the **GitHub Personal Access Token (PAT)** in the **Secret** field.
- Provide an **ID** (e.g., github-token) for reference.
- Click **"OK"** to save the credential.

This ensures secure authentication without exposing the token in the pipeline script.



6. Once the **Global Credentials** for the token are created, follow these steps to set up a **new Jenkins pipeline**:
    - Go to **Jenkins Dashboard** and click on **"New Item"**.
    - Enter the **Item Name** (e.g., MyPipeline).
    - Select **"Pipeline"** as the item type.
    - Click **"OK"** to proceed.



7. In the **Pipeline** section:
    - Scroll down to the **Pipeline** configuration.
    - Select **"Pipeline script"** as the definition.
    - Enter the **pipeline script** to automate the workflow.
    - Click **"Save"** to finalize the setup.

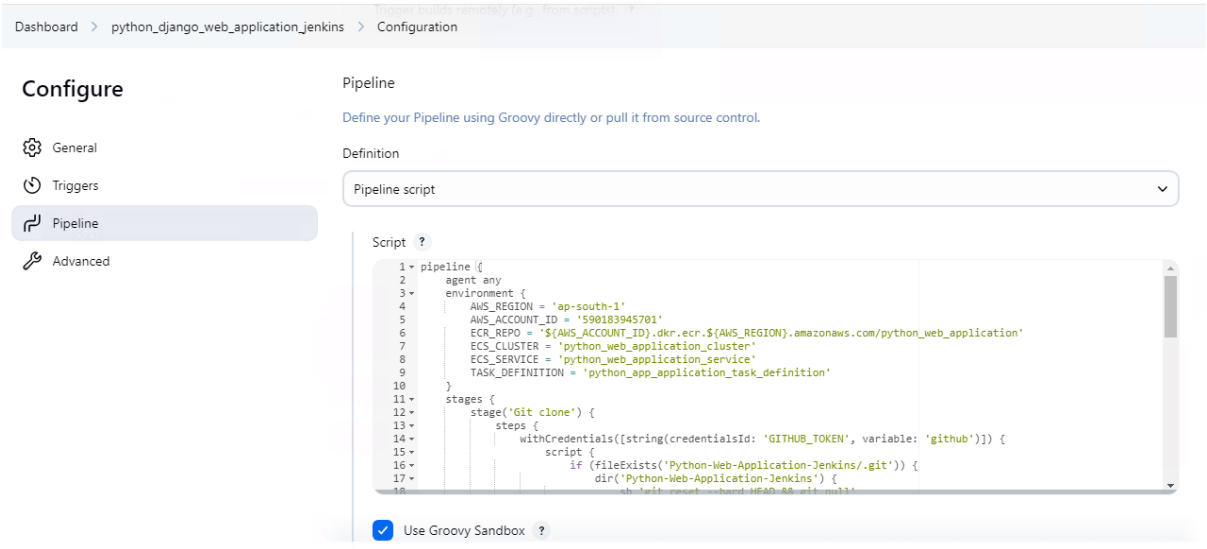
This script will define the build, test, and deployment steps for your Jenkins automation.

Pipeline script:

```
pipeline {
  agent any
  environment {
    AWS_REGION = 'ap-south-1'
          AWS_ACCOUNT_ID = '590183945701'
    ECR_REPO =
'${AWS_ACCOUNT_ID}.dkr.ecr.${AWS_REGION}.amazonaws.com/python_web_applicat
ion'
    ECS_CLUSTER = 'python_web_application_cluster'
    ECS_SERVICE = 'python_web_application_service'
    TASK_DEFINITION = 'python_app_application_task_definition'
  }
  stages {
    stage('Git clone') {
      steps {
        withCredentials([string(credentialsId: 'GITHUB_TOKEN', variable: 'github')]) {
                    script {
                        if (fileExists('Python-Web-Application-Jenkins/.git')) {
                            dir('Python-Web-Application-Jenkins') {
                                sh 'git reset --hard HEAD && git pull'
                            }
                        }
                        else {
                          sh 'git clone
https://$github@github.com/varunsimha-MP/Python-Web-Application-Jenkins.git'
```

```
                    }
                }
            }
        }
    }
        stage('Clean and build Docker Images on Jenkin server') {
                steps {
                  dir('Python-Web-Application-Jenkins') {
                        sh """
                            docker-compose down --rmi all --remove-orphans
                            docker-compose build
                        """
                    }
                  }
                }
    stage('Push Docker Images') {
      steps {
        dir('Python-Web-Application-Jenkins') {
          sh """
            docker login -u AWS -p \$(aws ecr get-login-password --region
${AWS_REGION}) ${ECR_REPO}
            #docker build -t ${ECR_REPO}:latest .
            docker tag python_web_application:latest ${ECR_REPO}:latest
            docker push ${ECR_REPO}:latest
          """
        }
      }
    }
```

```
    stage('Deploy to ECS') {
      steps {
        sh """
          aws ecs update-service --cluster ${ECS_CLUSTER} --service ${ECS_SERVICE} --force-new-deployment
        """
      }
    }
  }
}
```

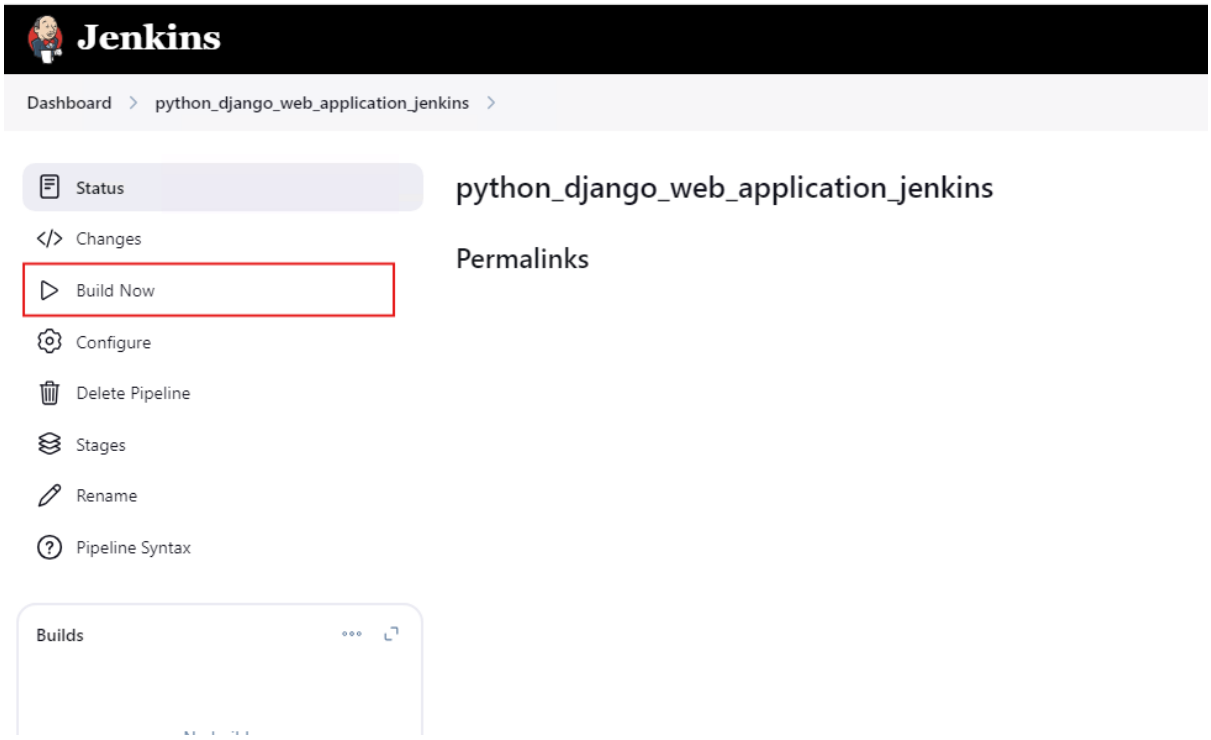


Module 7

Test and Validating the Setup

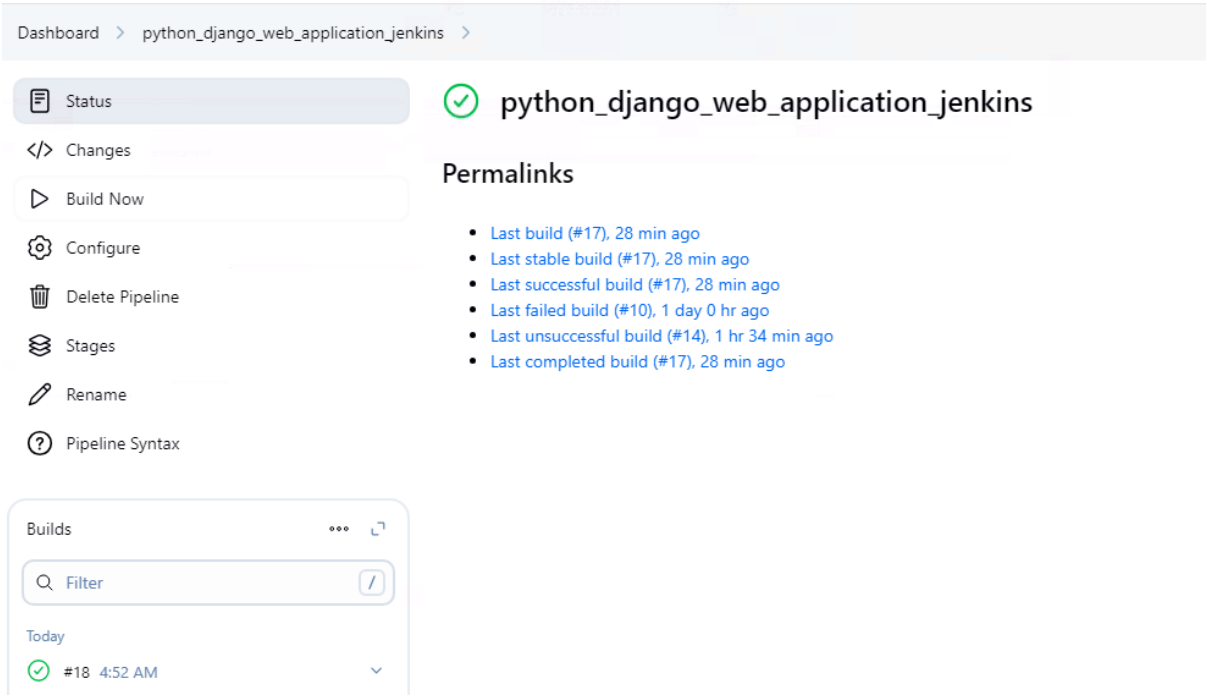
1. Once the **Pipeline** item is ready:
   - Click **"Build Now"** to trigger the pipeline.
   - Monitor the build progress in the **Build History** section.
   - Wait until the build completes successfully.

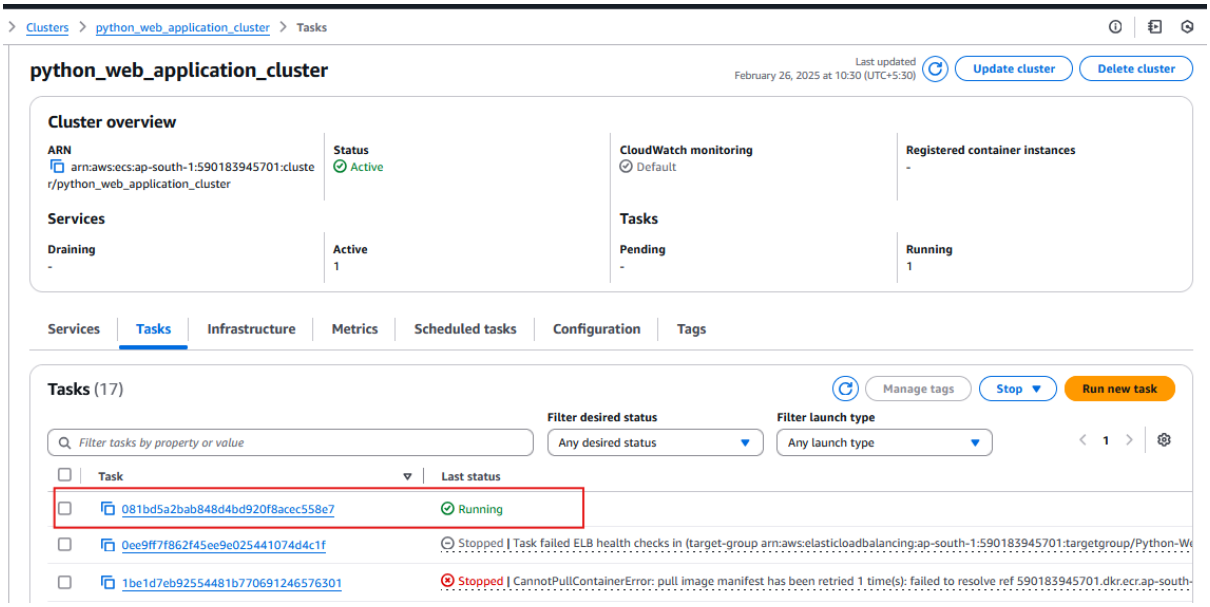
If any issues arise, check the **Console Output** for debugging. 🚀



2. After successfully **pushing/publishing** from the pipeline:
   - Go to the **AWS ECS Console**.
   - Navigate to your **Cluster**.
   - Click on the **Services** tab and locate your service.
   - Check the **"Run Task"** status.
   - You should see a **running task** after the successful deployment.

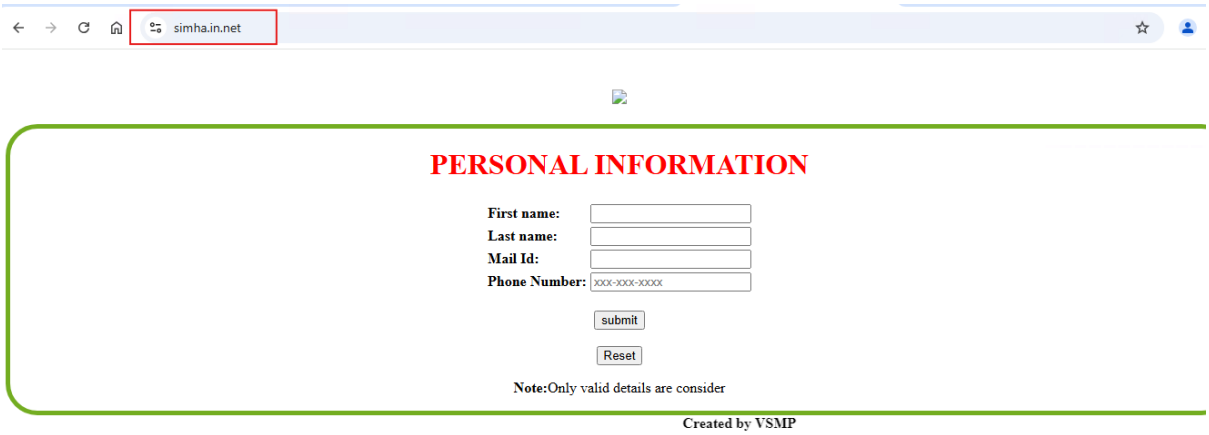
If the task is not running, check the **logs** in **CloudWatch** or the **ECS Events** for debugging. 🚀

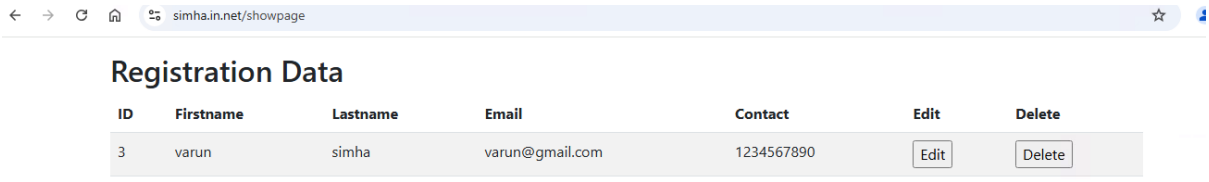




3. Once the task is running successfully:
    - Open your web browser.
    - Enter the **DNS name** (simha.in.net).
    - Access the application and navigate to the required page.
    - Fill in the form with some values.
    - Click **Submit** to test the functionality.

If any issues occur, check the **CloudWatch logs**, **RDS connection**, or **ECS task logs** for debugging.

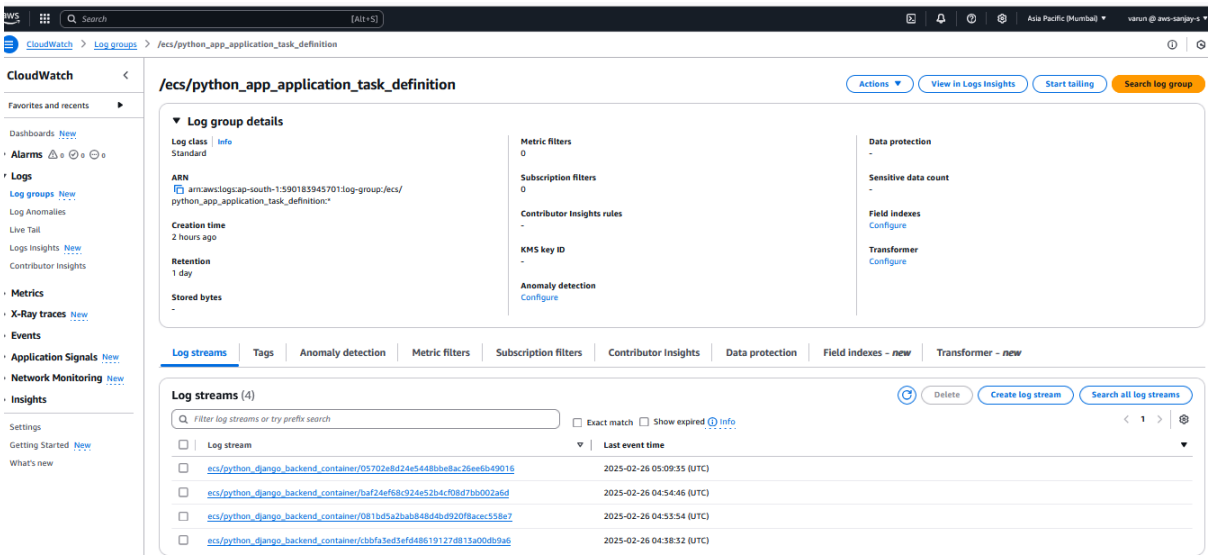


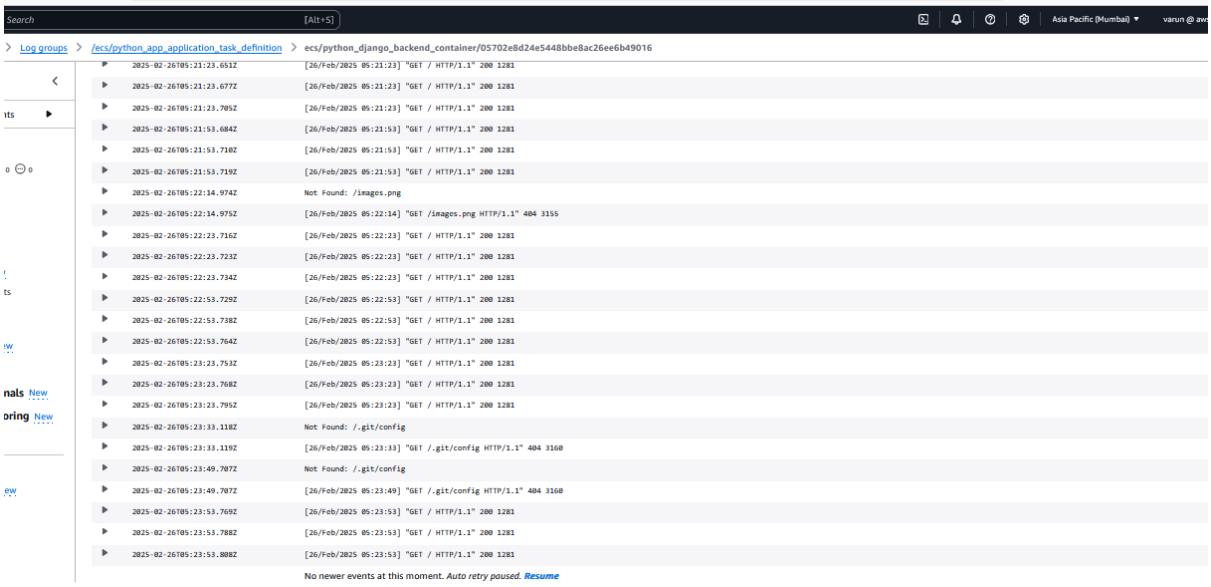


4. While working on the application, the logs are stored in **CloudWatch Logs Group**, allowing us to monitor the application's activity.

- For **troubleshooting container issues**, these logs can be helpful in:
    - Identifying errors or failures.
    - Monitoring application behavior and performance.
- To access logs:
    - Go to **AWS CloudWatch Console**.
    - Navigate to **Logs Groups**.
    - Select the appropriate **log group** associated with your ECS container.
    - View and analyze the logs for any errors or warnings.

These logs will be crucial for **debugging** and ensuring smooth application operations.





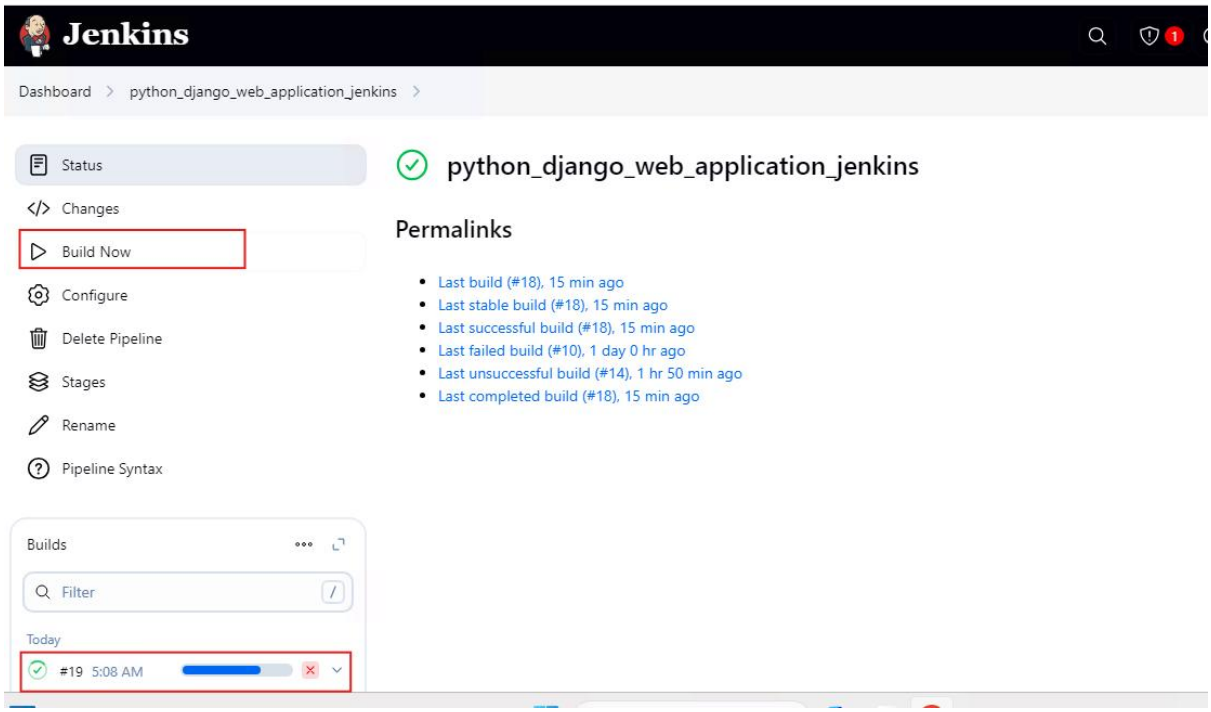
5. Now, let's update the application and redeploy it using Jenkins:

- We will modify the text **"Registration Data"** to **"Registered Data"** without causing any downtime.
- Using **Git** from a local machine, we will:
  - Make the necessary changes in the application code.
  - Commit and push the changes to the **GitHub repository**.
  - Trigger the Jenkins pipeline to redeploy the updated application seamlessly.

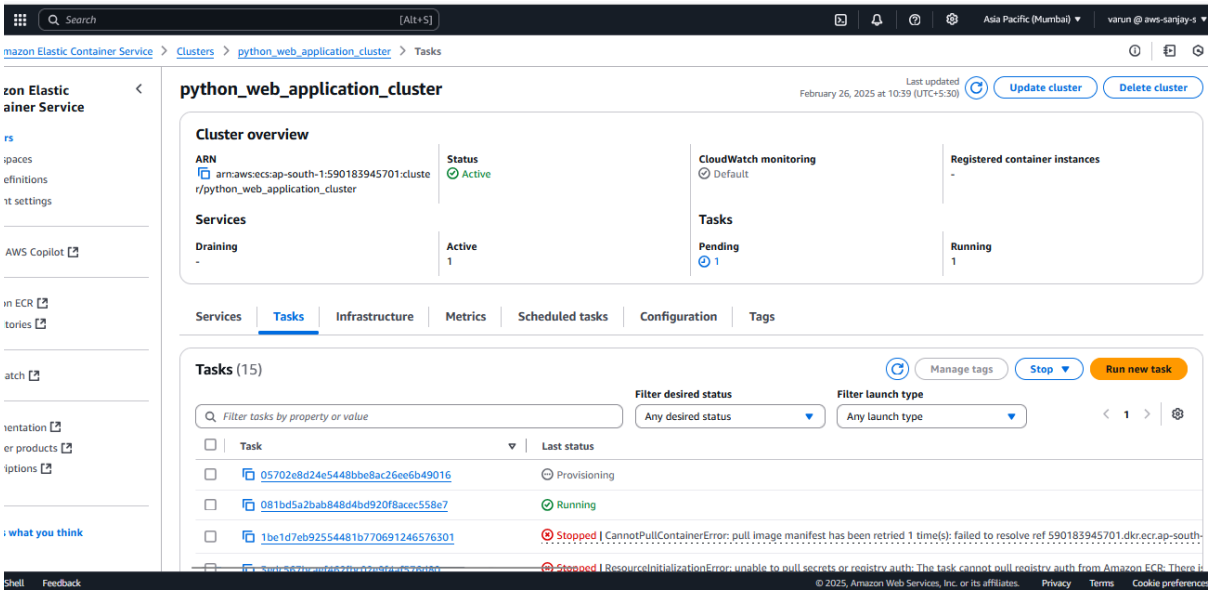
```
MINGW64:/c/Users/varunsimha.premanara/Downloads/new/Python-Web-Application-Jenkins

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ ls
backend/  docker-compose.yml

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ vi backend/app/templates/show.html

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ git add .

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ git commit -m "Updated the Registration Data name to Registered data on show.html page"
[master ec39d99] Updated the Registration Data name to Registered data on show.html page
 Committer: Varunsimha Premanarasimha <varunsimha.premanarasimha@stratogent.com>
Your name and email address were configured automatically based
on your username and hostname. Please check that they are accurate.
You can suppress this message by setting them explicitly. Run the
following command and follow the instructions in your editor to edit
your configuration file:

    git config --global --edit

After doing this, you may fix the identity used for this commit with:

    git commit --amend --reset-author

 1 file changed, 1 insertion(+), 1 deletion(-)

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$ git push
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 554 bytes | 138.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (4/4), completed with 4 local objects.
To https://github.com/varunsimha-MP/Python-Web-Application-Jenkins.git
   fff1d42..ec39d99  master -> master

varunsimha.premanara@STL-LAP-352 MINGW64 ~/Downloads/new/Python-Web-Application-Jenkins (master)
$
```

6. Once the changes are pushed to the **GitHub repository**, we will trigger the deployment from **Jenkins** by simply clicking the **"Build Now"** button.

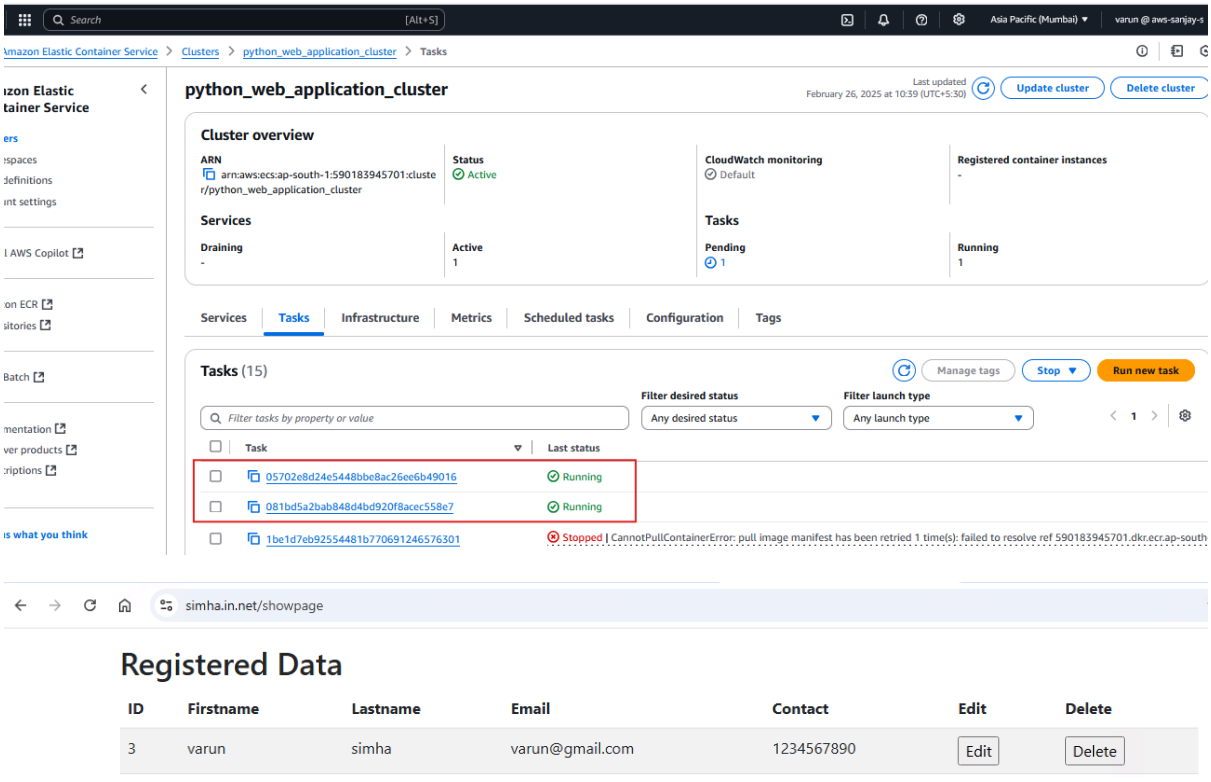
   There is no need to modify the pipeline script, as it will automatically fetch and deploy the updated code.



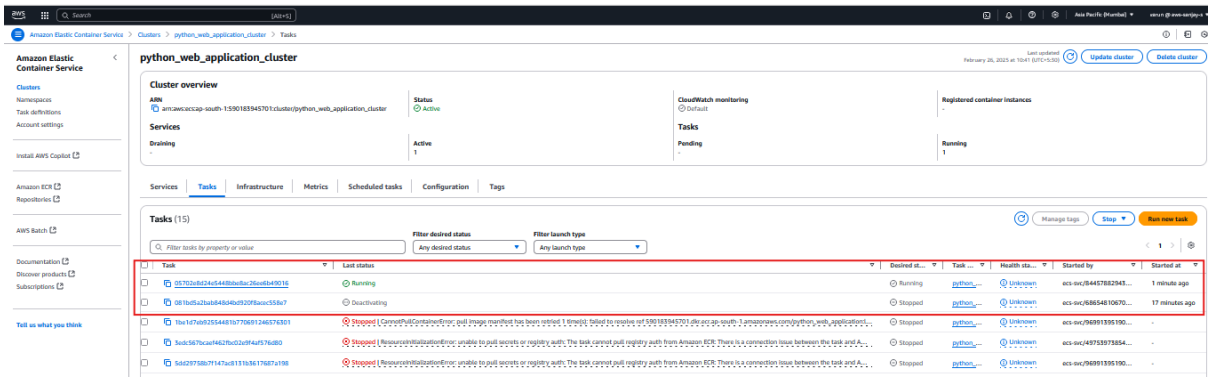
7. After the pipeline successfully completes, navigate to the **tasks** under the **ECS cluster**. You will see a new container being deployed, and its status will indicate that it is in the **provisioning** phase.



8. Once the new container is up and running, the **Load Balancer** will automatically switch traffic to it, ensuring the updated changes are reflected seamlessly without any downtime.



9. Once the new container goes live, the previous container will automatically stop, ensuring a smooth and seamless deployment process.



5. Overall, of this project:

This architecture diagram represents a **Fully Automated Scalable CI/CD Pipeline for Serverless Django on AWS**, integrating multiple AWS services for scalability, security, and automation. Here's a breakdown of the components:

1. User Access via Route 53

- End-users access the application using the domain simha.in.net, managed by AWS Route 53.
- The traffic is routed through a Load Balancer, which:
    - Redirects HTTP (port 80) to HTTPS (port 443) for security.
    - Dynamically routes traffic to target groups on port 8000.

2. AWS Fargate for Containerized Django Application

- The application is containerized using Docker and deployed on AWS Fargate (serverless container orchestration via ECS).
- The Elastic Container Registry (ECR) stores the Docker images for deployment.
- The Django application inside the container runs on port 8000.

3. Database - Amazon RDS

- The Django application connects to an Amazon RDS instance as its backend database.

4. Logging and Monitoring

- Amazon CloudWatch monitors and logs container activities for debugging and performance tracking.

5. CI/CD Pipeline using Jenkins

- Jenkins Server runs on an EC2 instance to automate the deployment process.

- Developers write and push code from VS Code to GitHub.
- Jenkins pulls code from GitHub, builds the Docker image, and pushes it to ECR.
- The new image is deployed on Fargate automatically.

Key Takeaways:

- **Scalable Django Deployment**: Using **Fargate** to avoid managing EC2 instances manually.
- **Security**: HTTPS enforcement and centralized authentication.
- **Automation**: Jenkins handles CI/CD, reducing manual deployments.
- **Monitoring**: Logs and performance tracking via CloudWatch.

This setup ensures an efficient, scalable, and automated deployment pipeline for Django applications on AWS. 🚀