**Course-End Project**

**Implementing microservices with CodePipeline and ECS**

**Steps to be followed:**

1. Set Up Docker and GitHub Repositories
2. Deploy Services on Amazon ECS
3. Automate CI/CD with AWS CodePipeline

**Step 1: Set Up Docker and GitHub Repositories**

1. Navigate to GitHub to access the GitHub repositories for both Backend (NodeJS) and Frontend (React) applications

<https://github.com/anujdevopslearn/ReactFrontend.git>

A screenshot of a computer

Description automatically generated

1. Create Dockerfiles for both services with the following configurations and push them to the respective repositories:

Backend Dockerfile:

**FROM node:16**

**ENV HOME=/home/app**

**COPY package.json $HOME/node\_docker/**

**WORKDIR $HOME/node\_docker**

**RUN npm install --silent --progress=false**

**COPY . $HOME/node\_docker**

**EXPOSE 3000**

**CMD ["npm", "start"]**

Frontend Dockerfile:

**FROM node:16**

**ENV HOME=/home/app**

**COPY package.json $HOME/node\_docker/**

**WORKDIR $HOME/node\_docker**

**RUN npm install --silent --progress=false**

**COPY . $HOME/node\_docker**

**EXPOSE 3000**

**CMD ["npm", "start"]**

A screenshot of a computer

Description automatically generated

**A screenshot of a computer

Description automatically generated**

**Step 2: Deploy Services on Amazon ECS**

* 1. Navigate to the AWS Console, open the Containers menu, and select Elastic Container Service (ECS) to proceed with setting up your new ECS cluster for deploying Frontend and Backend Docker containers

A screenshot of a computer

Description automatically generated

* 1. Click on **Create cluster** in the Amazon Elastic Container Service (ECS) dashboard to start the process of setting up a new cluster for deploying your Frontend and Backend Docker containers.

A screenshot of a computer

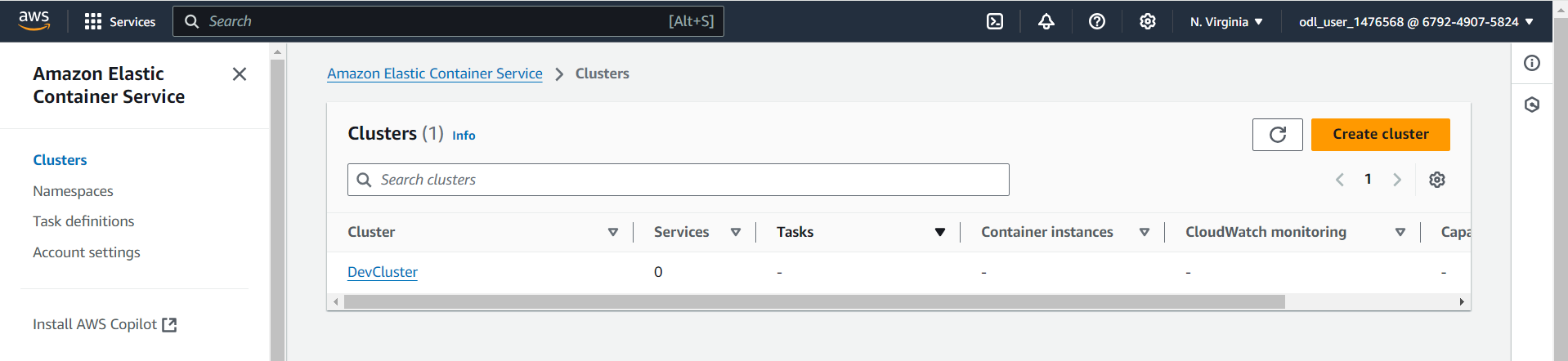
Description automatically generated

* 1. Enter the cluster name as **DevCluster**, select the default namespace, and choose **AWS Fargate (serverless)** as the infrastructure type to set up a serverless environment for your container deployment

A screenshot of a computer

Description automatically generated

* 1. Confirm that the new cluster **DevCluster** has been successfully created and is visible in the list of clusters in the Amazon ECS dashboard.



* 1. Verify that the cluster **DevCluster** is in an Active state in the cluster overview section of Amazon ECS. Check the details such as ARN, CloudWatch monitoring status, and the registered container instances before proceeding with service and task definitions

A screenshot of a computer

Description automatically generated

**Note:** We need to create task definition and basic service to start with container deployment

* 1. Click on **Create new task definition** in the Task Definitions section of Amazon ECS to start configuring a new task definition for your Docker containers, specifying details such as launch type, container definitions, and resource requirements

A screenshot of a computer

Description automatically generated

**Note**: Provide information regarding docker container to be deployed. We need to create two task definition one for backend and another one for frontend

* 1. Enter **backendcontainer** as the task definition family name, select **AWS Fargate** as the launch type, and choose the operating system as **Linux/X86\_64** with the network mode set to awsvpc for a serverless and secure container deployment

A screenshot of a computer

Description automatically generated

* 1. Set the container name to **backend**, use the image URI **docker.io/anujsharma1990/backend:latest**, mark it as **Yes** in Essential container, and enable port mapping with container port **3000**, protocol TCP, and application protocol HTTP. Ensure that resource limits are defined as required for CPU and memory allocation.

A screenshot of a computer

Description automatically generated

* 1. Click on **Create** to create task definition

A screenshot of a computer

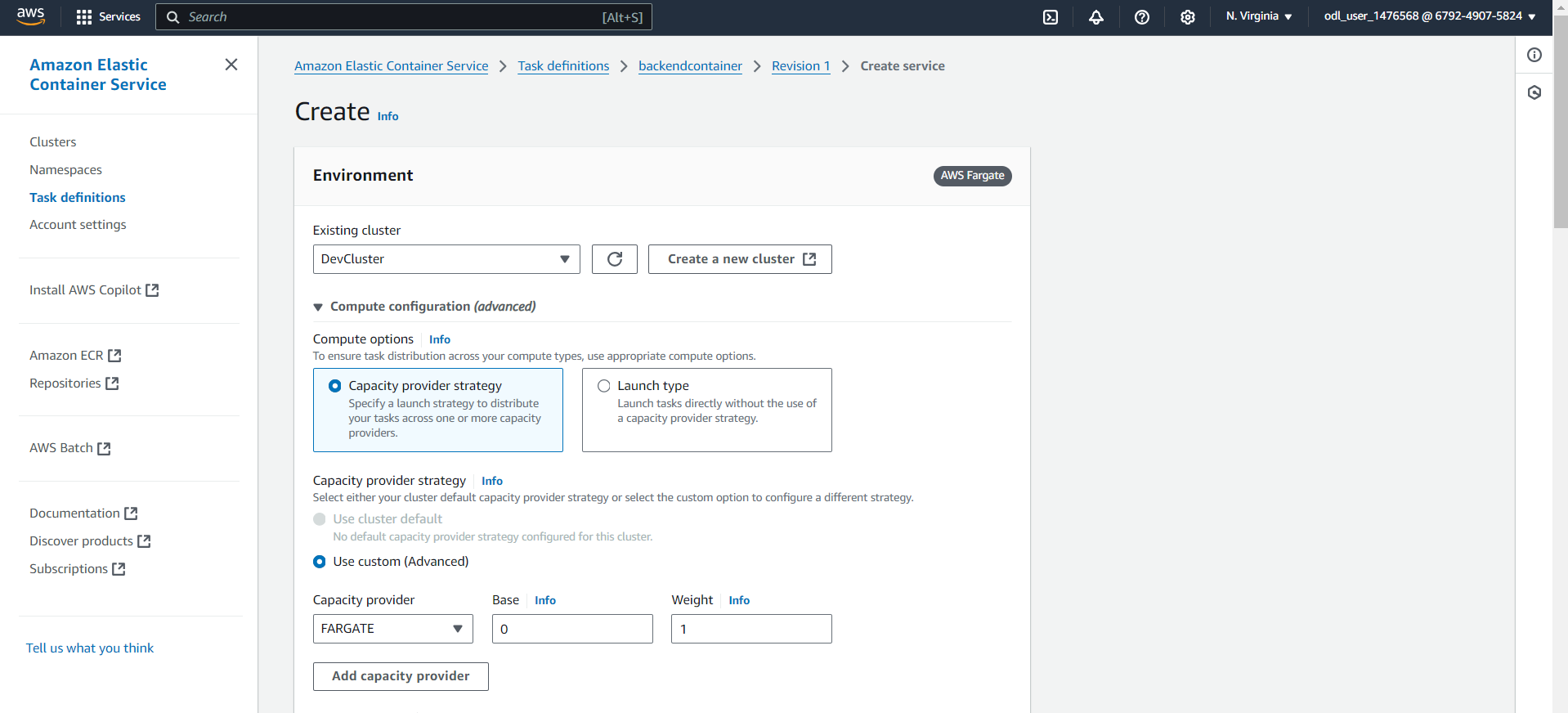
Description automatically generated

A screenshot of a computer

Description automatically generated

**Note**: Once task definition is created, we need to deploy the service.

* 1. Select the existing cluster **DevCluster**, choose **Capacity provider strategy** for compute configuration, and set the capacity provider to **FARGATE** with a weight of **1** to ensure tasks are deployed using AWS Fargate



* 1. For the deployment configuration, select **Service** as the application type, choose the task definition **backendcontainer** with revision **1**, set the service name as **backend-service**, and select **Replica** as the service type. Define the desired number of tasks as required for the deployment.

A screenshot of a computer

Description automatically generated

* 1. Choose the default **VPC**, select multiple subnets for high availability, and create a new security group named **sg\_backend\_service** with a description **Created in ECS Console**. This setup ensures secure networking and resource placement within the specified subnets

A screenshot of a computer

Description automatically generated

* 1. Set up an inbound rule in the security group **sg\_backend\_service** to allow TCP traffic on port **3000** from any source (0.0.0.0/0, ::/0) and ensure that the option for assigning a **Public IP** is turned on to enable external access to the backend service

A screenshot of a computer

Description automatically generated

* 1. Confirm that the deployment of **backend-service** is in progress and that the cluster DevCluster is in an **Active** state, as indicated in the cluster overview. Monitor the status of tasks and services to ensure successful deployment and proper operation

A screenshot of a computer

Description automatically generated

**Note:** We need to repeat same steps for frontend application

* 1. Create a new task definition named **frontend**, select **AWS Fargate** as the launch type, and set the operating system architecture to **Linux/X86\_64** with the network mode configured as **awsvpc** for a serverless and secure container environment

A screenshot of a computer

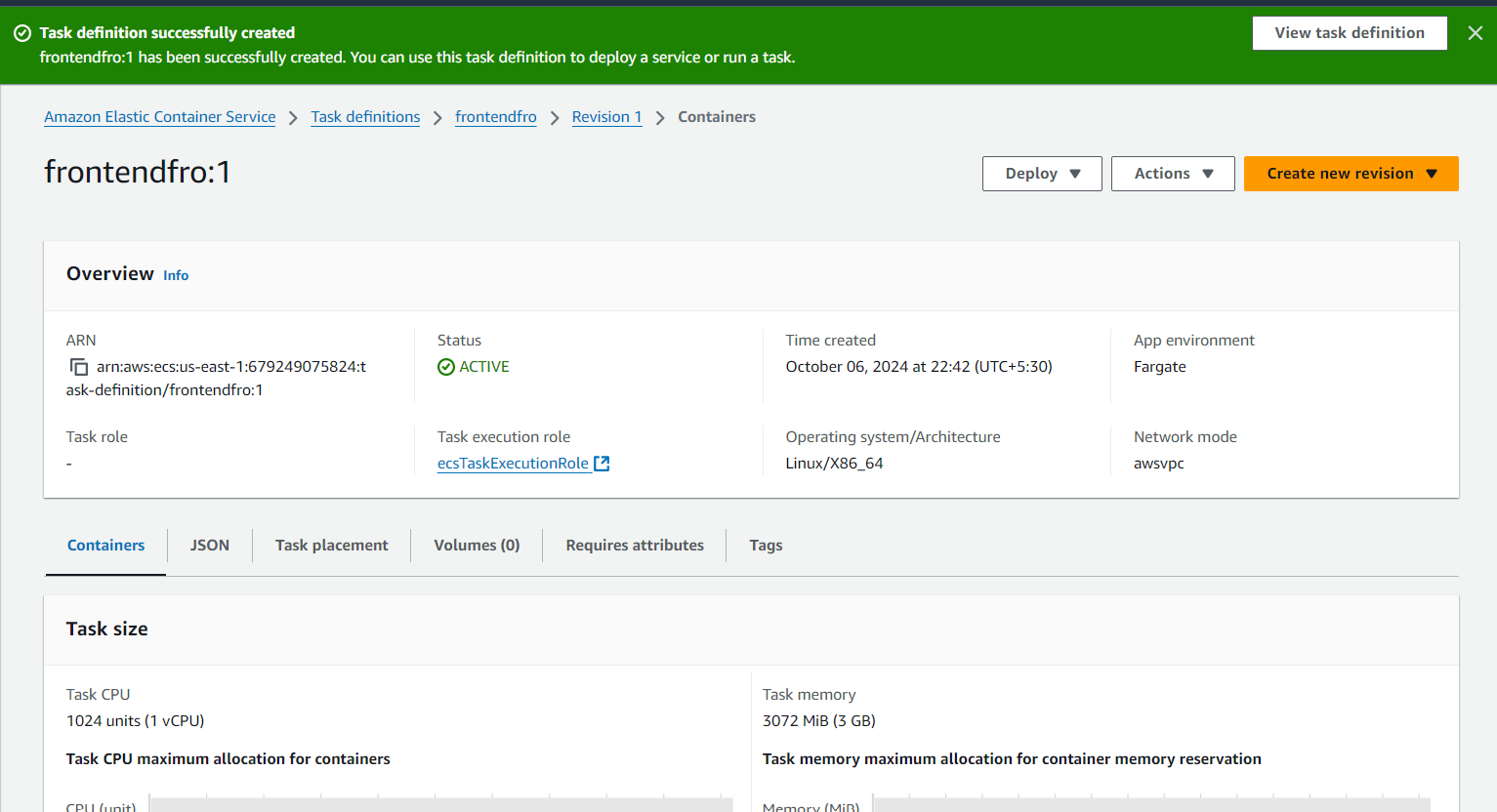
Description automatically generated

* 1. Set up **frontend-container** with image URI **docker.io/anujsharma1990/frontend:latest**, mark it as essential, configure port **80** with TCP and HTTP protocols, and allocate resources with **1 CPU**, **1 GPU**, a hard memory limit of **3 GB**, and a soft limit of **1 GB**

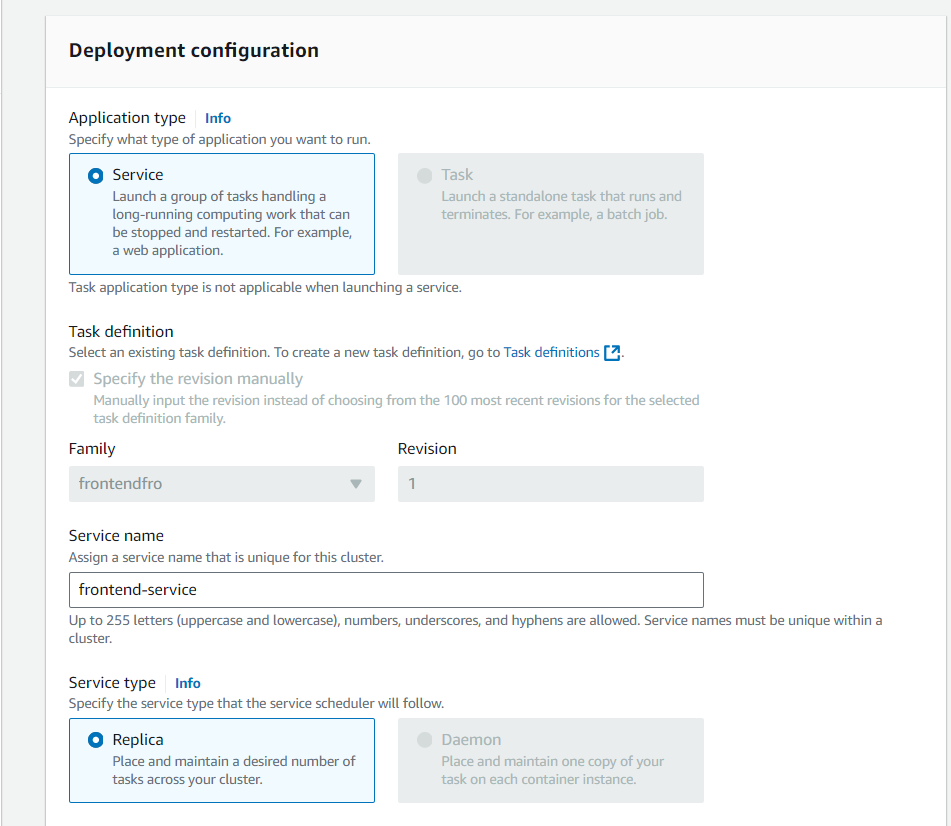
A screenshot of a computer

Description automatically generated

Confirm that the task definition **frontendfro:1** has been successfully created and is in an **Active** state, with 1 vCPU, 3 GB memory, using **Fargate** launch type, **Linux/X86\_64** architecture, and awsvpc network mode, ready for deployment



* 1. Choose **Service** as the application type, select the task definition **frontendfro** with revision **1**, set the service name to **frontend-service**, and choose **Replica** as the service type to maintain the desired number of tasks in the cluster.



* 1. Create a new security group named **sg\_frontend\_security** with a description **Created in ECS Console**, add an inbound rule to allow TCP traffic on port **3000** from any source (0.0.0.0/0, ::/0), and ensure that the option for assigning a **Public IP** is turned on.

A screenshot of a computer

Description automatically generated

Once both backend and frontend services are created you can see them created as per below screenshot.

A screenshot of a computer

Description automatically generated

**Step 3: Automate CI/CD with AWS CodePipeline**

**Note:** Once cluster is provisioned, we can proceed with configuring AWS Code pipeline to automate CI CD workflow

* 1. Navigate to the **AWS Management Console**, go to **Developer Tools**, and select **CodePipeline** to start configuring your continuous delivery pipeline for software release.

A screenshot of a computer

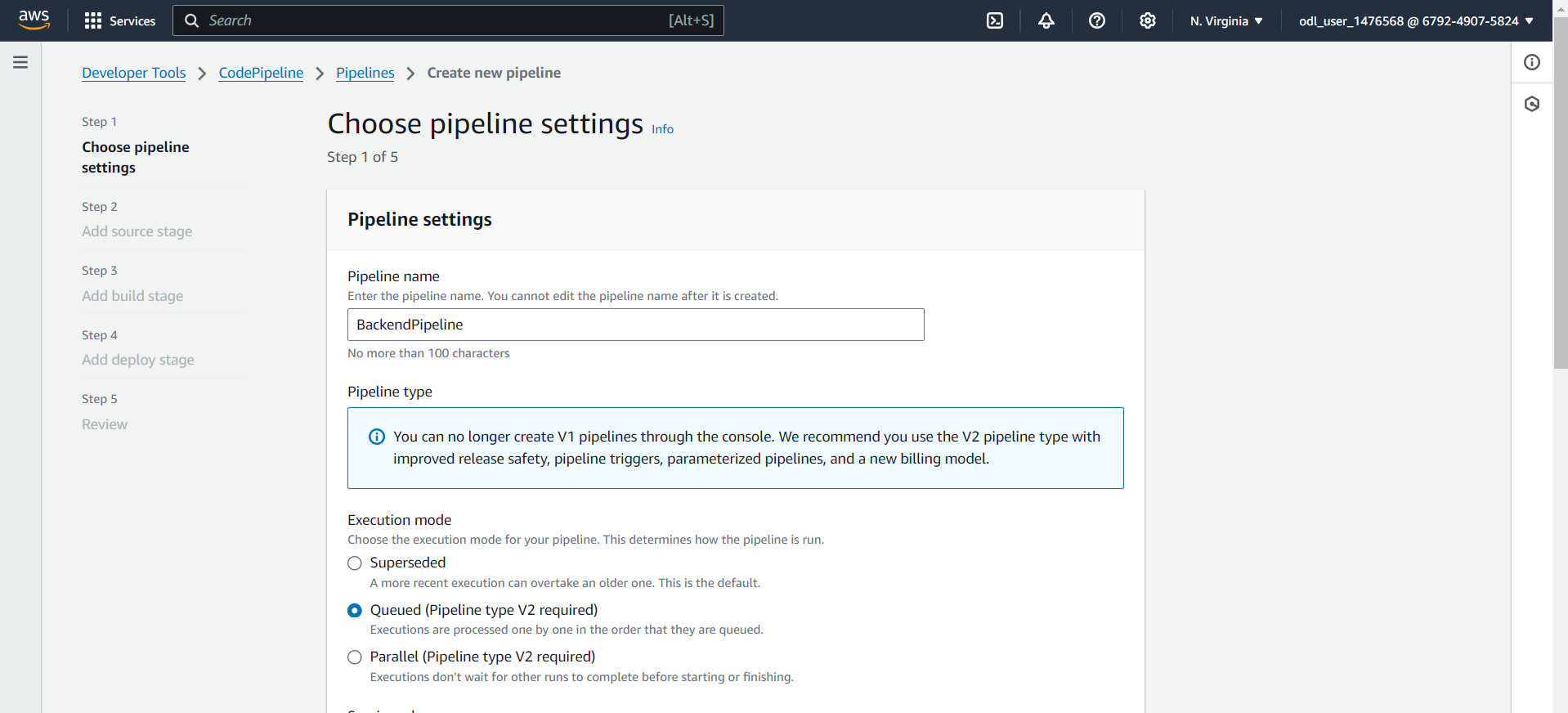
Description automatically generated

* 1. Click on **Create pipeline** in the CodePipeline dashboard to begin setting up a new continuous delivery pipeline.

A screenshot of a computer

Description automatically generated

* 1. Set the **Pipeline name** to **BackendPipeline**, select the pipeline type as **V2**, and choose the **Queued** execution mode for processing executions in the order they are triggered.



* 1. Set **Source provider** to **GitHub (Version 2)**, click **Connect to GitHub** to create a connection, specify the **Connection**, input the **Repository name**, and select the **Default branch** for the source stage.

A screenshot of a computer

Description automatically generated

* 1. Enter the **Connection name** as **connection**, add optional **Tags** if needed, and click **Connect to GitHub** to proceed with creating the GitHub App connection

A screenshot of a computer

Description automatically generated

* 1. Click **Authorize AWS Connector for GitHub** to grant permissions for verifying GitHub identity, accessing resources, and acting on your behalf, completing the integration between AWS and GitHub

A screenshot of a computer

Description automatically generated

* 1. Ensure the **Connection name** is set to **connection**, select the **App installation** with ID **17235361**, and click **Connect** to finalize the GitHub connection setup in AWS

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* 1. Confirm the **GitHub connection**, set the **Repository name** to **anujdevopslearn/NodeBackend**, select **master** as the **Default branch**, and choose **CodePipeline default** for the output artifact format.

A screenshot of a computer

Description automatically generated

Keep the default trigger configurations and continue to the next step

A screenshot of a computer

Description automatically generated

* 1. Choose **Other build providers**, select **AWS CodeBuild** as the build provider, click **Create project** to define a new build project, and select **Single build** as the build type

A screenshot of a computer

Description automatically generated

A screenshot of a computer

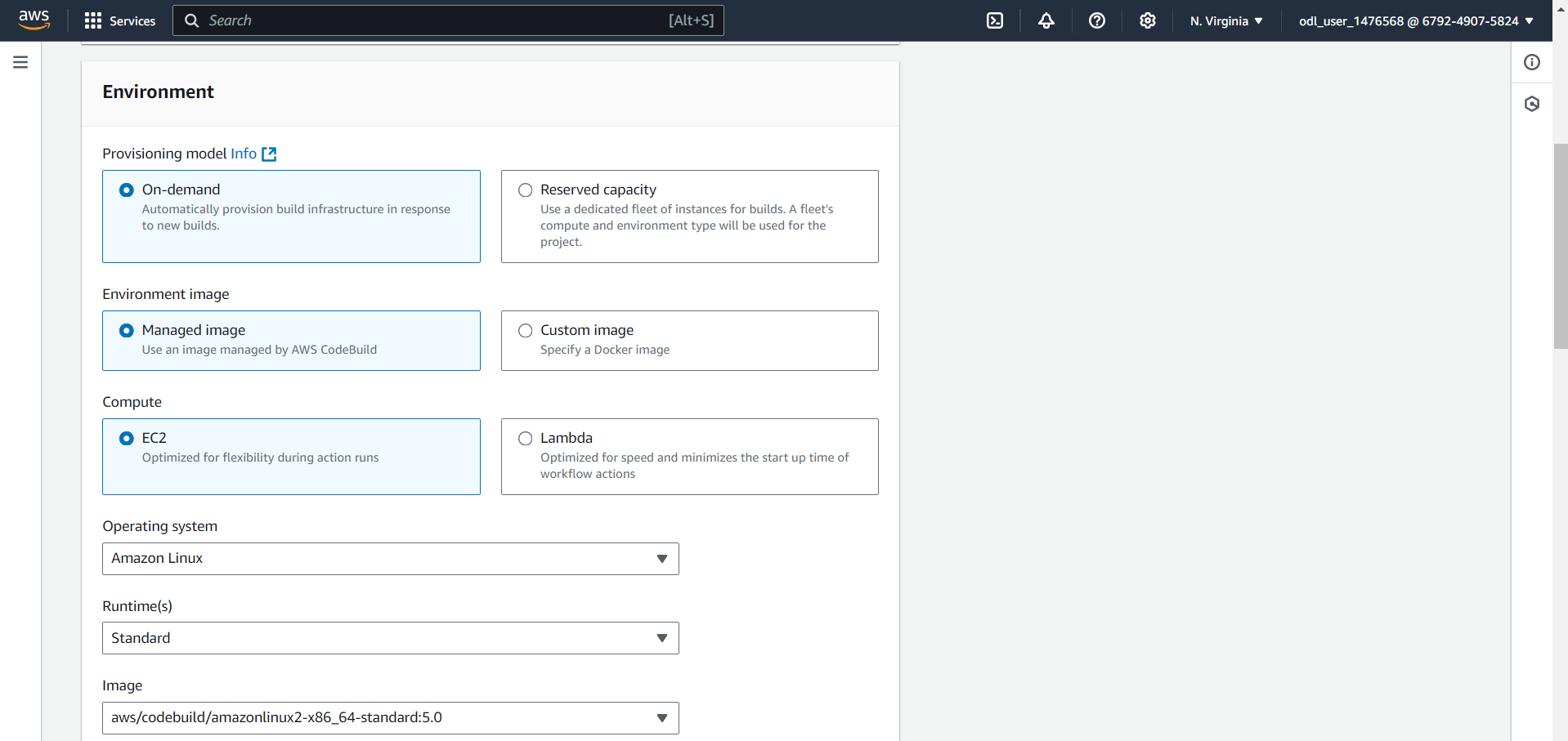
Description automatically generated

* 1. Enter the **Project name** as **BuildProject**, select **On-demand** for the provisioning model, and choose **Managed image** for the environment image to use an AWS CodeBuild managed image

A screenshot of a computer

Description automatically generated

* 1. Select **On-demand** for the provisioning model, choose **Managed image** for the environment image, set **EC2** for compute, select **Amazon Linux** as the operating system, choose **Standard** for runtime(s), and pick the image **aws/codebuild/amazonlinux2-x86\_64-standard:5.0**



* 1. Select **New service role**, set the role name to **codebuild-BuildProject-service-role**, configure the timeout to **1 hour**, set the queued timeout to **8 hours**, and enable the **Privileged** flag for elevated build permissions if needed

A screenshot of a computer

Description automatically generated

* 1. Choose **Use a buildspec file**, specify the buildspec name as **buildspec.yml**, and optionally enable **CloudWatch logs** for build output monitoring

A screenshot of a computer

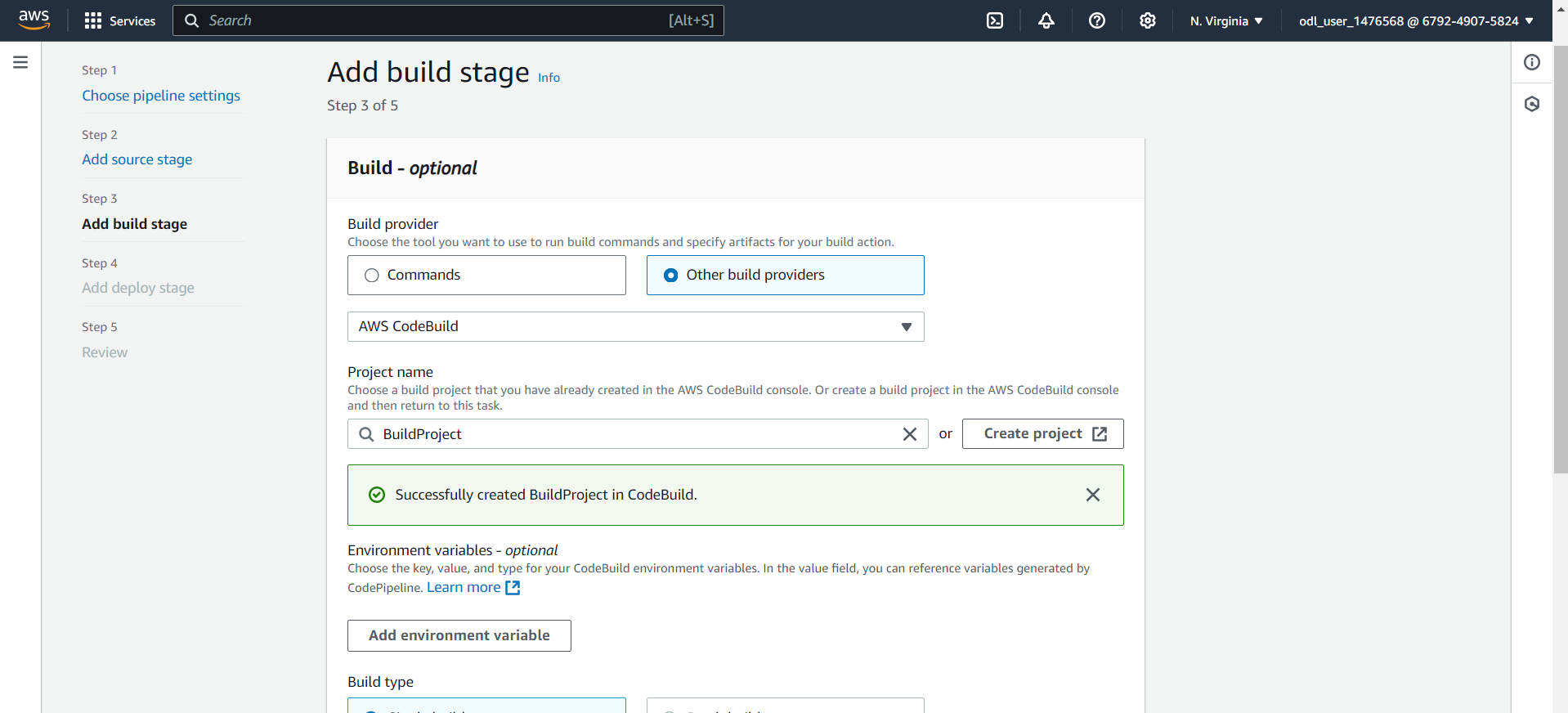
Description automatically generated

* 1. Choose project name and then proceed with adding environment variables for connecting with Docker hub

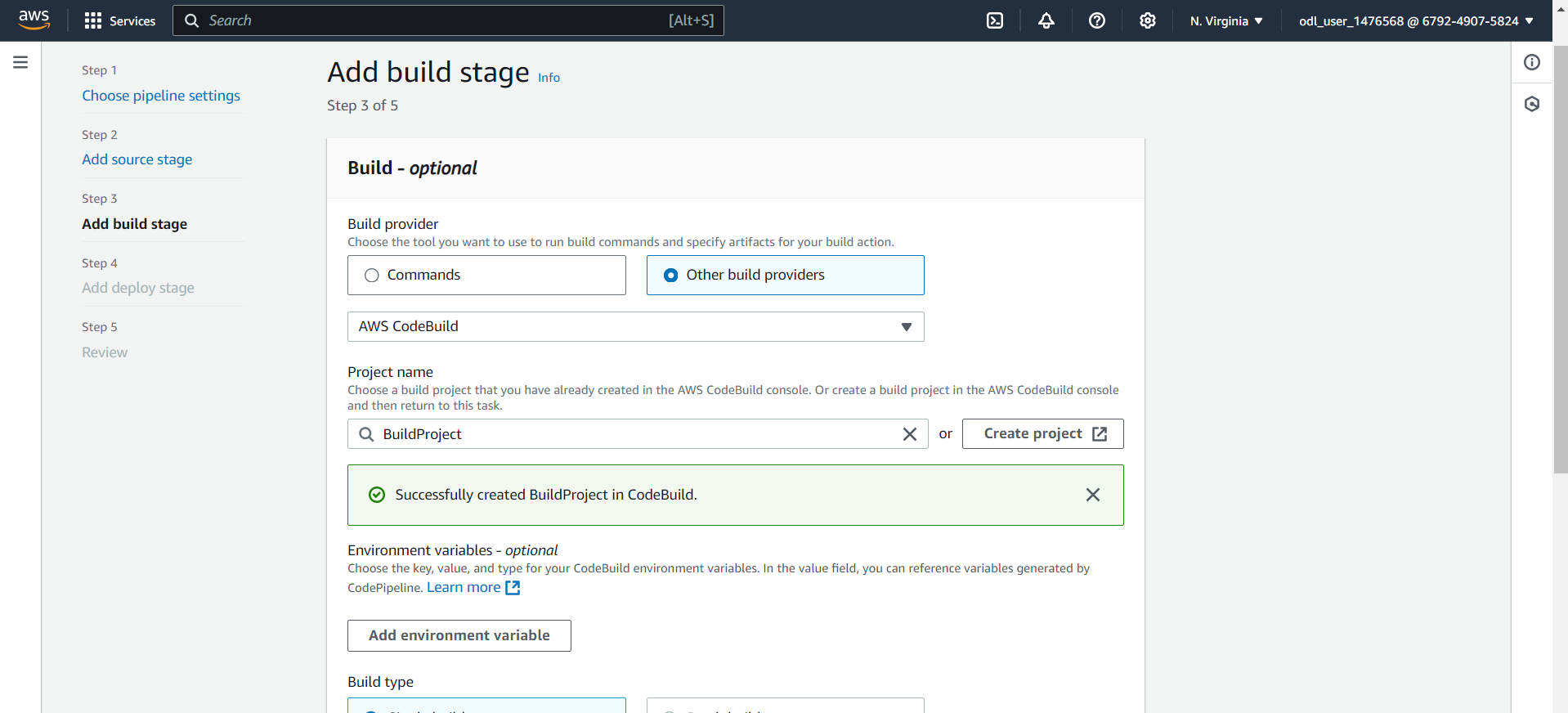
A screenshot of a computer

Description automatically generated

* 1. Confirm that **BuildProject** is successfully created, select it as the **Project name**, and choose **Single build** for the build type to proceed with the build stage setup in CodePipeline



* 1. Click on Add environment variable and add variables as per below screenshot



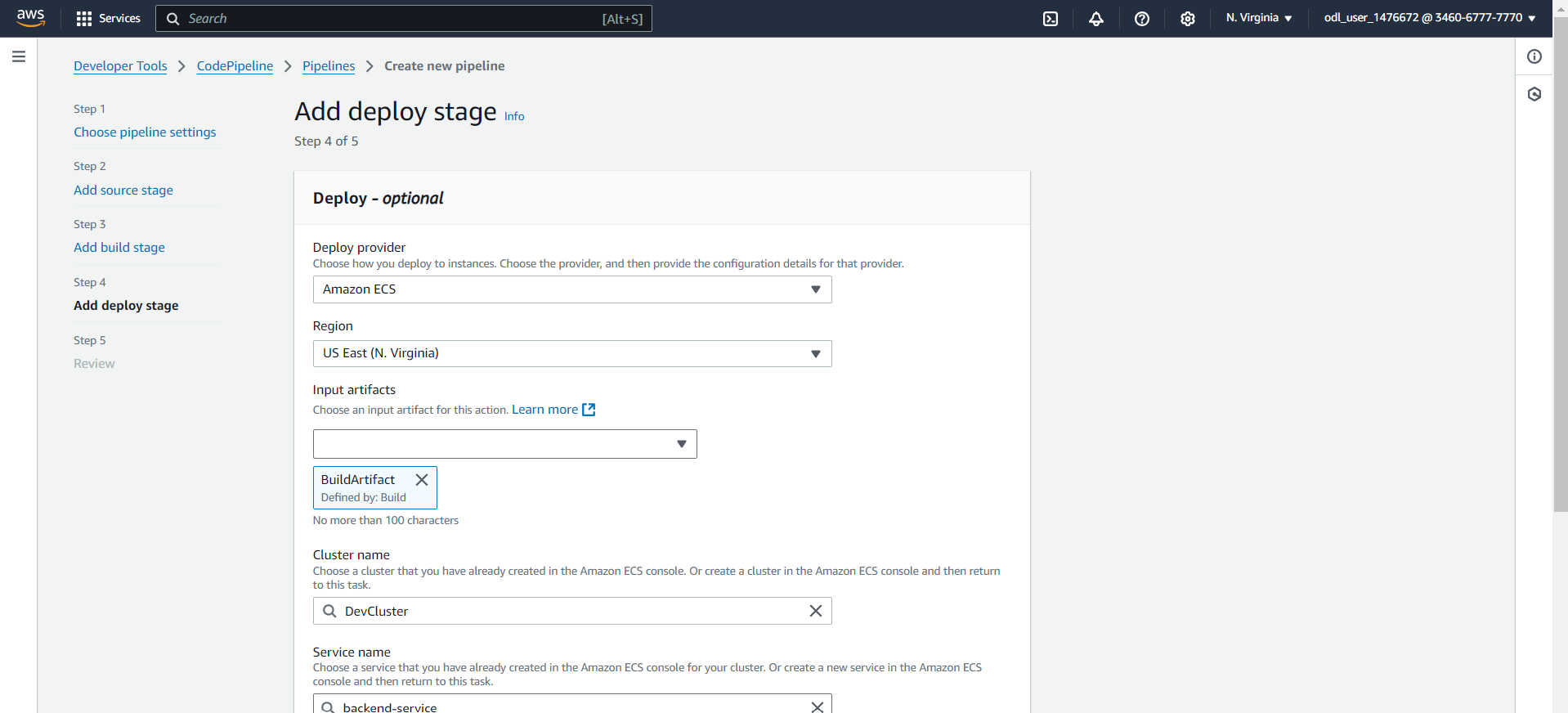
* 1. Navigate to Docker hub and create a new free account and keep Docker hub username and password with you before configuring Jenkins credential

<https://hub.docker.com/>

A screenshot of a login page

Description automatically generated

* 1. Set the **Deploy provider** to **Amazon ECS**, choose the **Region** as **US East (N. Virginia)**, select **BuildArtifact** as the input artifact, set the **Cluster name** to **DevCluster**, and specify the **Service name** as **backend-service**



* 1. Set the **Service name** to **backend-service**, specify the image definitions file as **imagedefinition.json**, leave the **Deployment timeout** optional, and click on **Next**

A screenshot of a computer

Description automatically generated

**Note:** Initial Pipeline execution would be failure since we do not have buildspec.yml stored in Backend repository, we must proceed with that

* 1. Navigate to Backend repository on GitHub and create buildspec.yml using below content

**version: 0.2**

**phases:**

**pre\_build:**

**commands:**

**- echo Logging in to DockerHub...**

**- docker login -u $DOCKERHUB\_USER -p $DOCKERHUB\_PASS**

**- REPOSITORY\_URI="docker.io/anujsharma1990/backend"**

**- CONTAINER\_NAME="backend-container"**

**- IMAGE\_TAG="latest"**

**build:**

**commands:**

**- echo Build started on `date`**

**- echo Building the Docker image...**

**- docker build -t $REPOSITORY\_URI:$IMAGE\_TAG .**

**post\_build:**

**commands:**

**- echo Build completed on `date`**

**- docker push $REPOSITORY\_URI:$IMAGE\_TAG**

**- echo Writing image definitions file...**

**- printf '[{"name":"'$CONTAINER\_NAME'","imageUri":"%s"}]' $REPOSITORY\_URI:$IMAGE\_TAG > imagedefinition.json**

**artifacts:**

**files: imagedefinition.json**

A screenshot of a computer

Description automatically generated

Once file is created Code pipeline would be invoked automatically

A screenshot of a computer

Description automatically generated

Once Deploy stage is successful you can see if service is deployed successfully on ECS Cluster

A screenshot of a computer

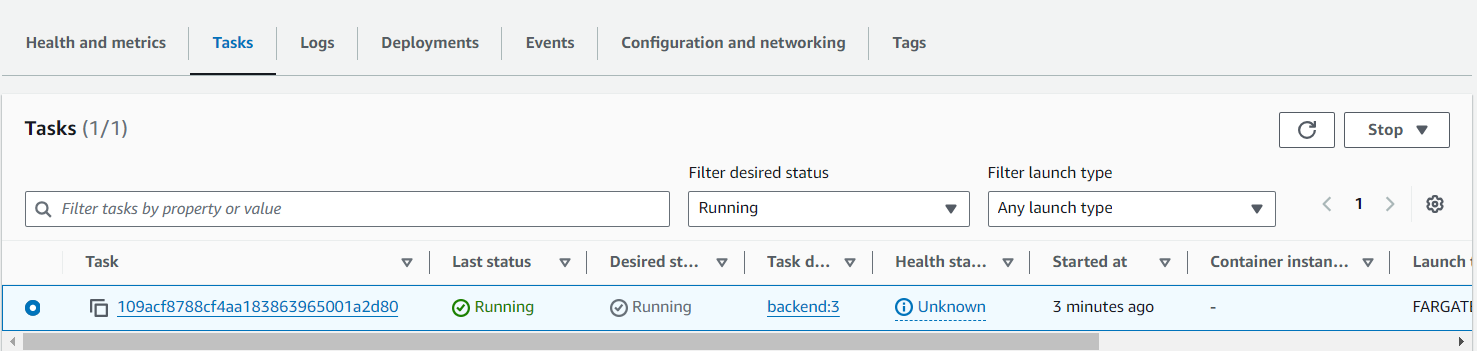
Description automatically generated

Backend Service is up and running post deployment which confirms deployment worked

A screenshot of a computer

Description automatically generated

* 1. Navigate to Tasks tab and select running task. Under networking tab, you will be able to access Public IP using which you can access Backend API



* 1. Use Public IP address and URL as below to access backend application

**http://<public-ip>:3000/testAPI**



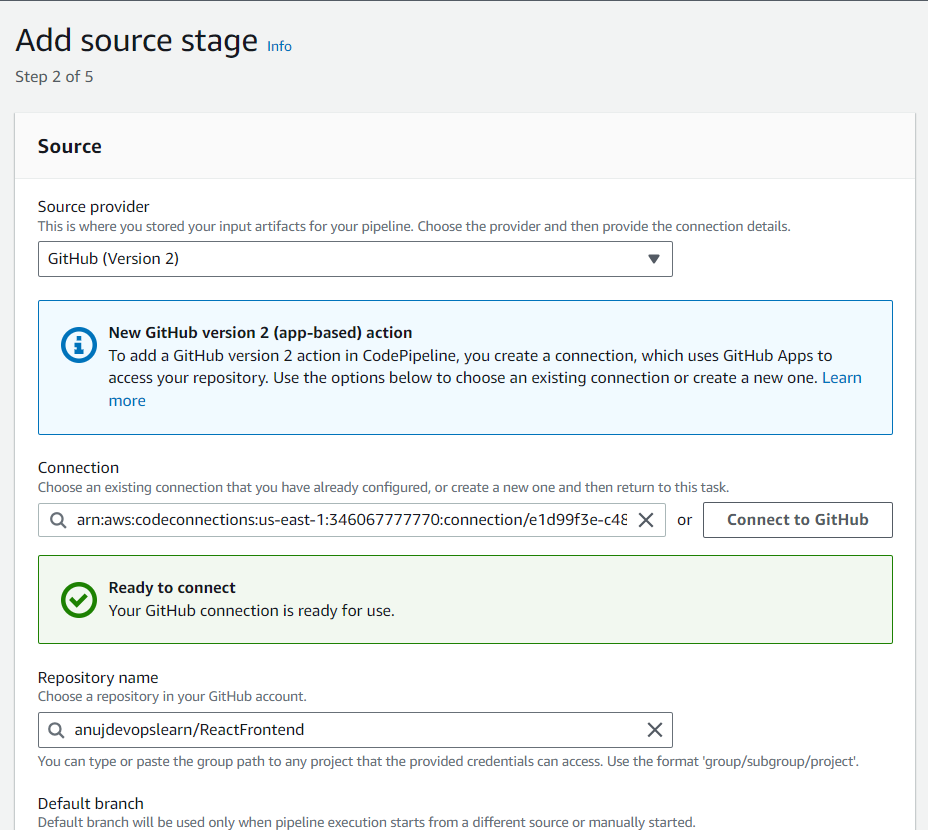
Once Backend pipeline is configured, we need to follow same steps with Frontend application also

* 1. Set the **Pipeline name** to **FrontendPipeline**, select **V2** for the pipeline type, choose **Queued** for the execution mode, and opt for **New service role** for the service role configuration.

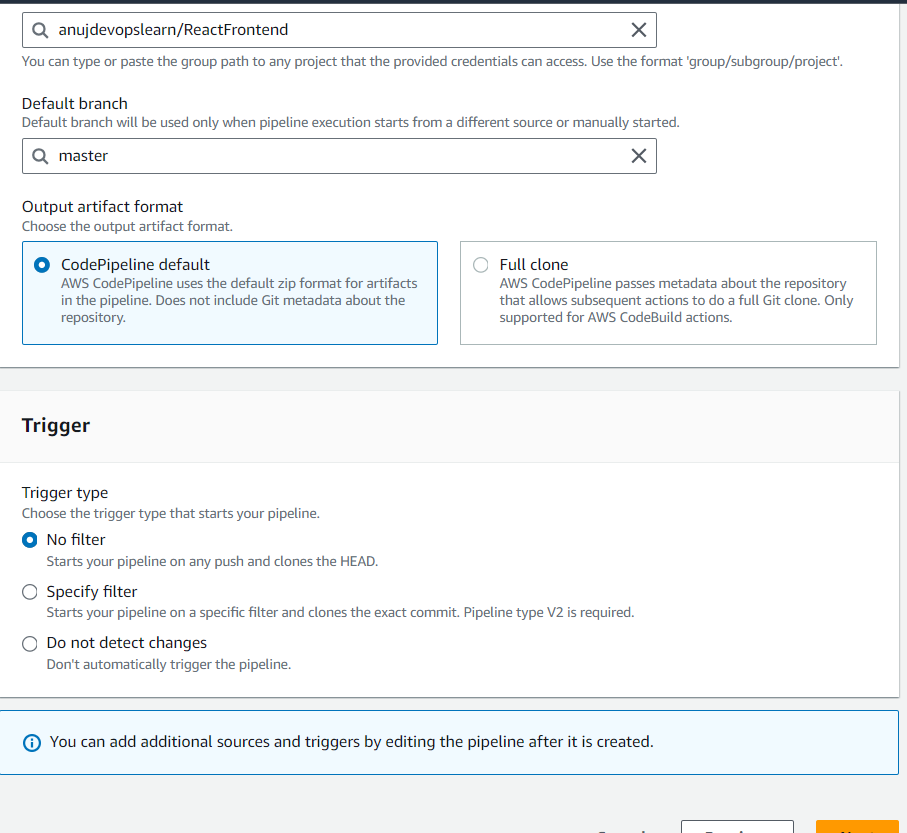
A screenshot of a computer

Description automatically generated

* 1. Set the **Source provider** to **GitHub (Version 2)**, confirm the connection with the provided ARN, use **anujdevopslearn/ReactFrontend** as the **Repository name**, and specify the default branch as needed.



* 1. Set the **Default branch** to **master**, choose **CodePipeline default** for the output artifact format, and select **No filter** as the trigger type to start the pipeline on any push and clone the HEAD.



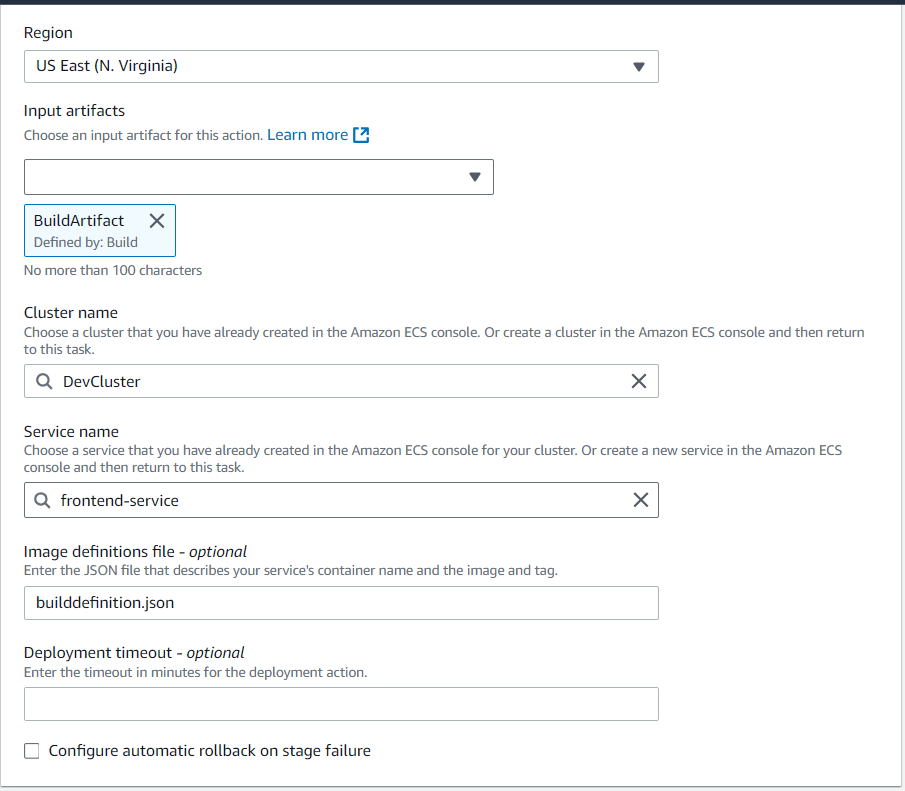
* 1. Select **Use a buildspec file**, set the **Buildspec name** to **buildspec.yml**, and optionally enable **CloudWatch logs** for monitoring build outputs

A screenshot of a computer

Description automatically generated

Once a Code build project is created navigate to Deploy stage to ECS Cluster.

* 1. Select the **Region** as **US East (N. Virginia)**, choose **BuildArtifact** as the input artifact, set the **Cluster name** to **DevCluster**, specify the **Service name** as **frontend-service**, use **builddefinition.json** for the image definitions file, leave the **Deployment timeout** optional, and enable **Configure automatic rollback on stage failure** if needed



* 1. We need to modify .env file where backend API URL will be added

A screenshot of a computer

Description automatically generated

Once the build spec file is created, you will see pipeline getting invoked automatically.

A screenshot of a computer

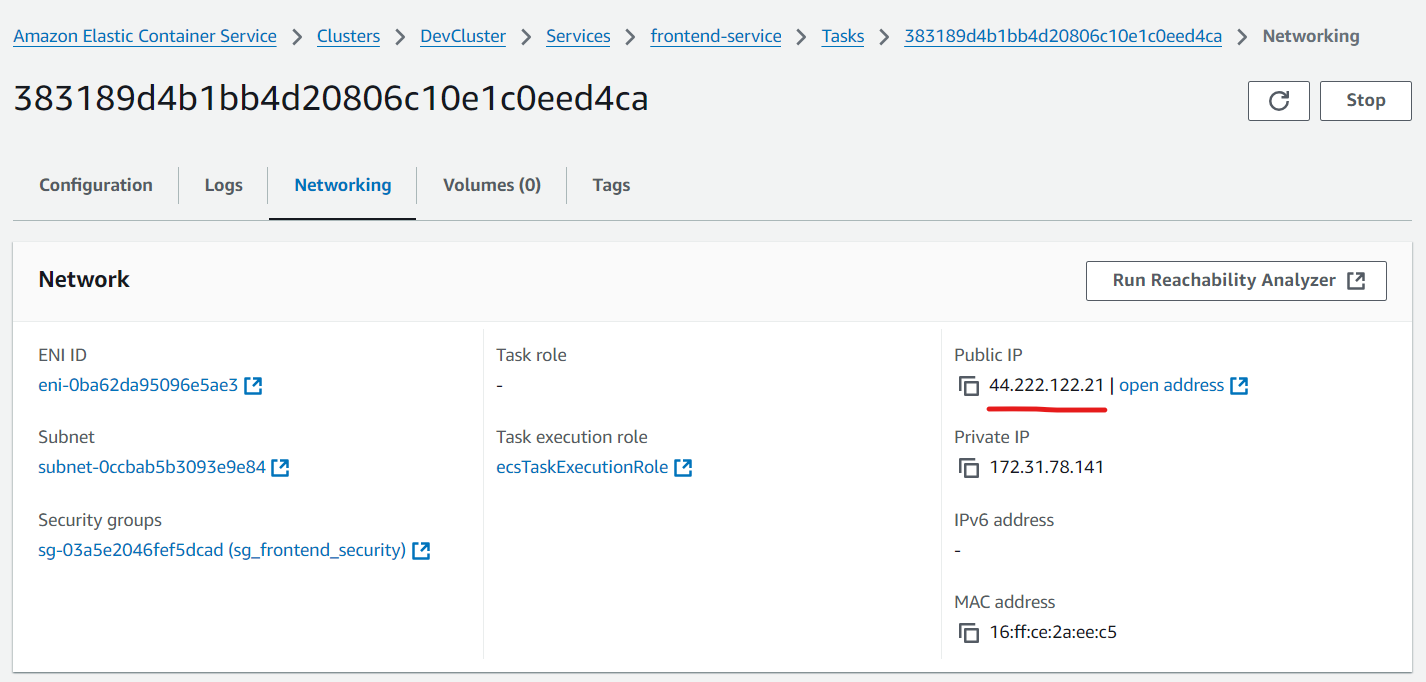
Description automatically generated

* 1. Once pipeline is successful, we can see both backend and frontend tasks deployed on ECS cluster and up and running

A screenshot of a computer

Description automatically generated

Navigate to frontend service and select task which you want to access.



* 1. Use system default browser to access application deployed on Docker container frontend application

**http://<public-ip>:3000**

A screen shot of a computer

Description automatically generated

By following the above steps, you have successfully implemented microservices architecture using AWS CodePipeline and ECS, enabling automated CI/CD for seamless deployment of both backend and frontend applications, ensuring they are up and running on the ECS cluster with full integration and public accessibility.