IITR-FSD-Content Integration and Deployment

Jenkins ECS Terraform Pipeline

v1.0

Contents

[1. Introduction 2](#_Toc740583350)

[2. CICD Tool Stack 3](#_Toc1996204434)

[3. Pre-requisites 3](#_Toc1143240837)

[4. CI-CI Architecture 4](#_Toc1745217351)

[Deployment strategy: Rolling updates 4](#_Toc92769586)

[5. ECS Cluster architecture 4](#_Toc1063393744)

[6. Terraform files 5](#_Toc297752875)

[7. Deployment Procedure 6](#_Toc1881668924)

[8. References 14](#_Toc809203027)

[9. Revision Sheet 15](#_Toc1625764766)

# Introduction

|  |  |
| --- | --- |
| **Team** | |
| SANGRAM KESARI SAHU | Jenkins & AWS CLI |
| AKSHIT BAUNTHIYAL | Docker & NPM |
| ANSHUL VANAWAT | ECR, Terraform & Documentation |
| MALCON LOBO | ECS & Documentation |
| NARAYANAN KESAVAN | GitHub, Pipeline & Integration |

In this project, we will learn how to set up a continuous integration and continuous delivery (CI/CD) pipeline on AWS.  A pipeline helps us to automate steps in your software delivery process, such as introducing automatic builds and then deploying to Amazon EC2 instances. We will use Jenkins, a service that builds, tests, and deploys the code every time there is a code change, based on the release process models defined. As part of our setup, we will plug other dev-ops tools into Jenkins to complete your software delivery pipeline.  The other tools which are used are GitHub, Terraform, Docker, ECS, Fargate and ECR. GitHub acts as source code repository, ECR will be our docker repository, cluster will be built on ECS. The complete infrastructure will be created by Terraform using Fargate. Jenkins will be manually configurate and integrated with GitHub for CI-CD pipeline.

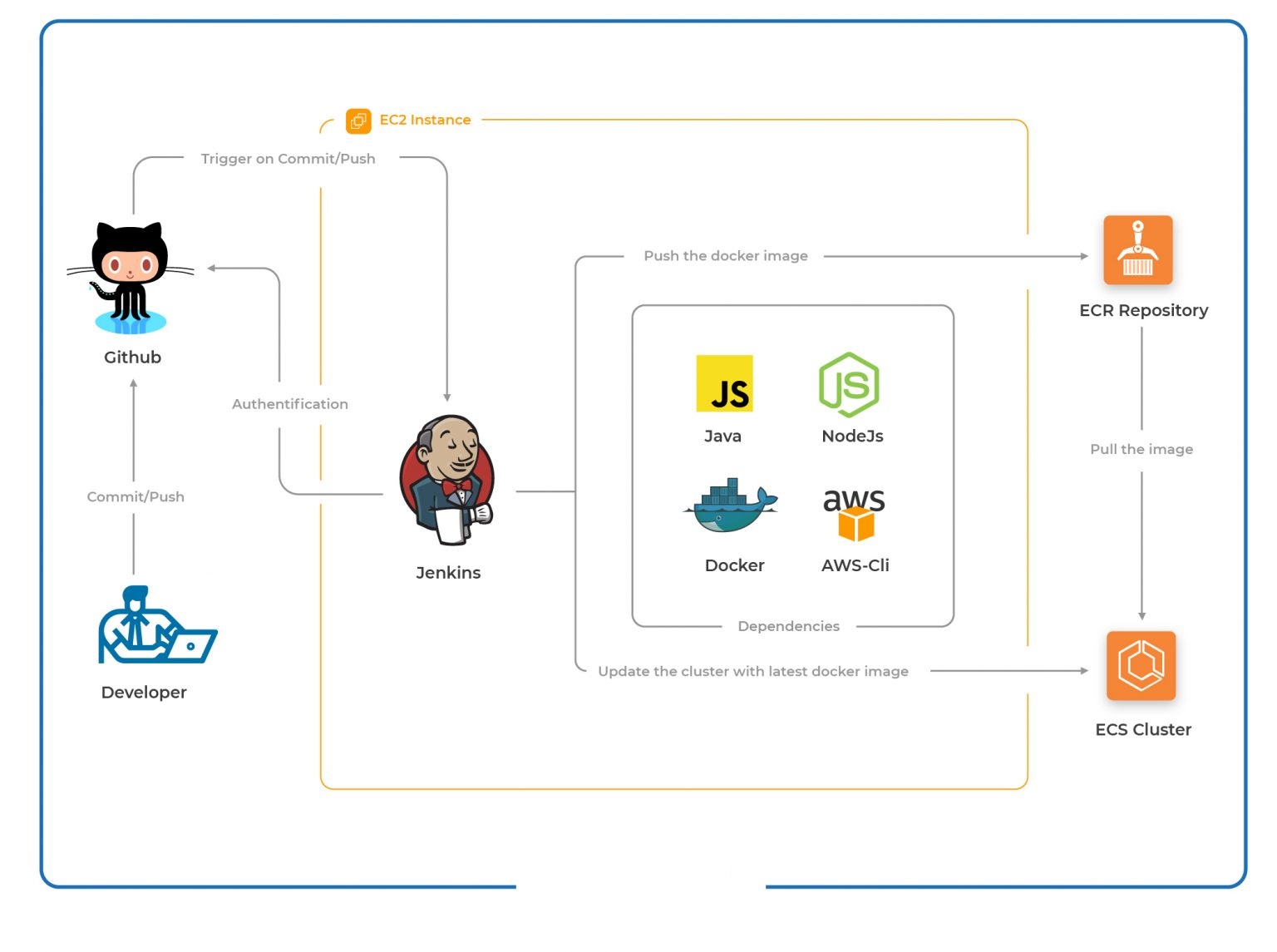
# CICD Tool Stack

1. **GitHub: -** It is a web-based application or a cloud-based service where people or developers collaborate, store, and manage their application code using Git. We will create and store our application code here.
2. **AWS EC2 Instance: -** AWS EC2 is an Elastic Computer Service provided by Amazon Web Services used to create Virtual Machines or Virtual Instances on AWS Cloud. We will create an EC2 instance and install Jenkins and other dependencies in it.
3. **Java: -** This will be required to run Jenkins Server.
4. **AWS CLI: -** AWS-cli i.e., AWS Command Line Interface is a command-line tool used to manage AWS Services using commands.
5. **Docker: -** Docker is an open-source containerization platform used for developing, shipping, and running applications. We will use it to build Docker Images of our sample application.
6. **Jenkins: -** Jenkins is an open-source, freely available automation server used to build, test, and deploy software applications. We will be creating our CI CD Docker Pipeline to build, test and deploy our application on AWS EC2 using Jenkins
7. **ECR: -** ECR is a registry service on AWS that allows you to manage your docker images with versioning and lifecycle policies. We will be using ECR to store Docker Images of our sample application.
8. **ECS: -** AWS ECS with Fargate is a serverless computing platform that makes running containerized services on AWS easier than ever before.
9. **Terraform:** - Terraform is an infrastructure-as-code tool created by Hashicorp to make handling infrastructure more straightforward and manageable.
10. **Nginx:** - It is an open-source software for web serving, reverse proxying, caching, load balancing, etc. We’ll be using in our docker container to host a React.js project.

# Pre-requisites

1. System with AWS-CLI installed.
2. Configure AWS-CLI with command ‘AWS configure’
   1. -set access key id
   2. -set secret access key
   3. -set default region
3. Source code of the web application with Docker file, Jenkins file and task definitions.
   1. <https://github.com/anshulv1401/capstone_frontend> (Download from here).
4. Public and private key pair (DemoKeyPair.pub, DemoKeyPair.pem) to access Jenkins-server.

# CI-CD Architecture

This is how our architecture will look like after setting up the CI CD Pipeline with Docker.

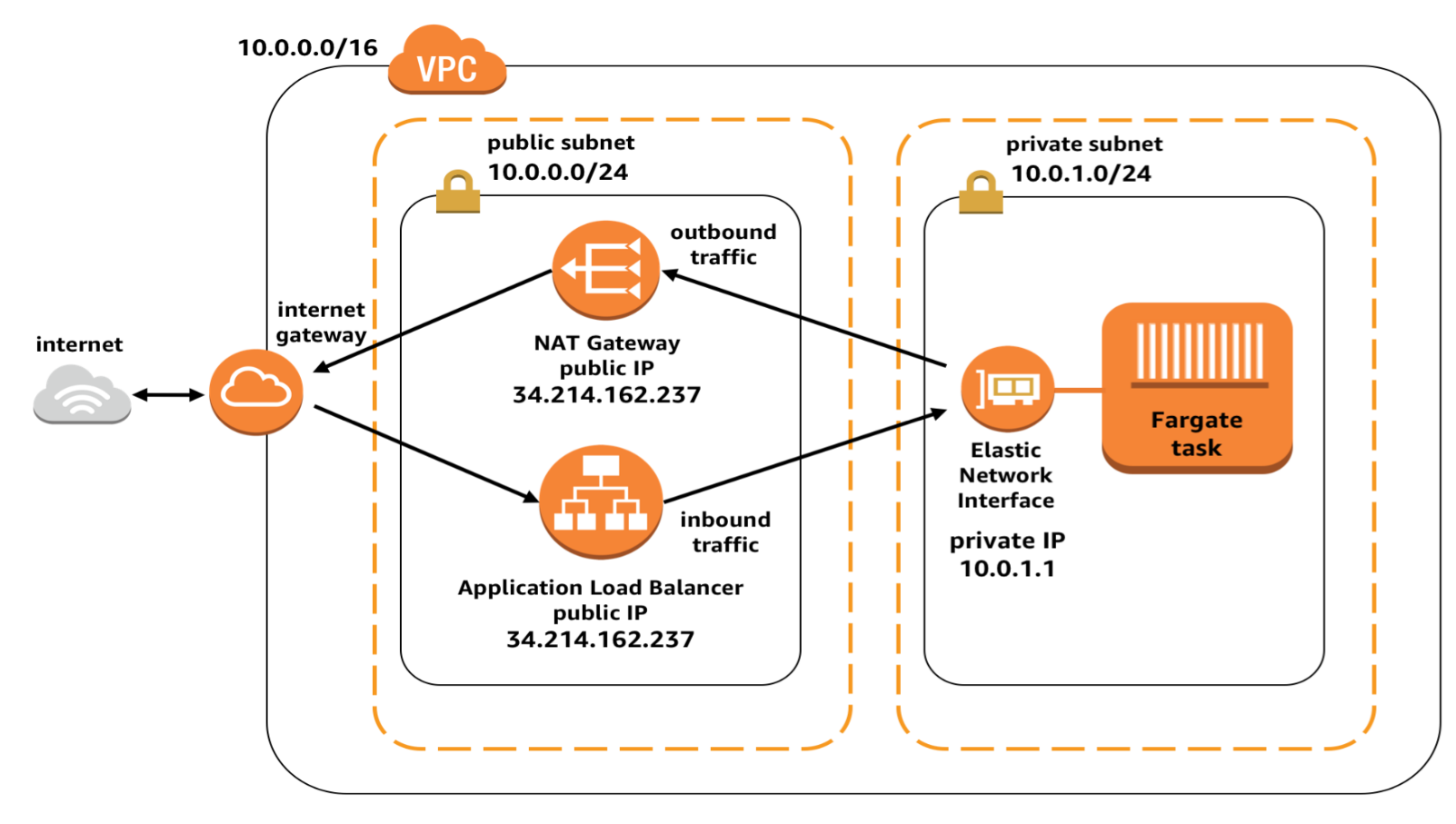
After the CI CD Docker Pipeline is successfully set up, we will push commits to our GitHub repository and in turn, GitHub Webhook will trigger the CI CD Pipeline on Jenkins Server. Jenkins Server will then pull the latest code, carry out unit tests, build a docker image and push it to AWS ECR. After the image is pushed to AWS ECR, the same image will be deployed in AWS ECS by Jenkins.

### **Deployment strategy: Rolling updates**

In rolling updates, the scheduler in the ECS Service replaces the currently running tasks with new ones. The tasks in the ECS cluster are nothing but running containers created out of the task definition. Deployment configuration controls the number of tasks that Amazon ECS adds or removes from the service. The lower and the upper limit on the number of tasks that should be running is controlled by minimumHealthyPercent and maximumPercent respectively.

1. minimumHealthyPercent example: If the value of minimumHealthyPercent is 50 and the desired task count is 4, then the scheduler can stop 2 existing tasks before starting 2 new tasks
2. maximumPercent example: If the value of maximumPercent is 200 and the desired task is 4 then the scheduler can start 4 new tasks before stopping 4 existing tasks.

# ECS Cluster architecture

Docker container would be run in a Fargate task, and this task is executed in a private subnet, talking to the outside world via the NAT gateway in the public subnet. The ALB in the public subnet funnels HTTP requests to the task, and the response is coming via the NAT gateway

# Terraform files

1. **Jenkins-server**.
   1. main.tf: - Terraform provider, with region.
   2. DemoKeyPair.pub: - Public key which is used in key.tf
   3. key.tf: - Public key for ec2 instance. Private key is should already be available with you to login into the instance.
   4. jenkins-server.tf: - EC2 instance, with Ubuntu and t3.medium type.
   5. sg\_ec2.tf: - Security group for Jenkins-server. Open ports 80, 8080, 22, etc.
   6. script-jenkins.sh: - Installing required dependencies.
   7. variables.tf: - Contains the variables which are getting used for building Jenkins server.
2. **ECS-ECR**.
   1. provider.tf: - Terraform provider, with region.
   2. vpc.tf: - VPC with 10.0.0.0/16 cidr block is created in the given region.
   3. internet-gateway: - Creates an internet gateway attached to the newly created vpc.
   4. subnets: - Public and private subnets are created under 3 availability zones. ("us-east-1a", "us-east-1b", "us-east-1c")
   5. routing-table-for-public-subnets: - Creates routing tables, connecting each public subnet with Internet gateway.
   6. natgateway-for-private-subnets.tf: - Creates Nat-gateway.
   7. routing-table-for-private-subnets: - Creates routing tables, connecting each private subnet with the Nat gateway
   8. security-group: - Create security group for application load balancer allowing 80 and 443 ports, and security group for ecs-task, allowing container port mentioned in the variable file.
   9. ecr-creation.tf: - Creates an ECR repository which will be used to store our docker images. This repository also has a lifecycle policy which clears the older docker images if it goes beyond the max image count.
   10. ecs-with-taskdef.tf: - Create a ECS cluster (*ecs-demo-cluster-test*) with task definition. The definition has the following configuration
       1. CPU = 256
       2. memory = 512
       3. Port-mapping = container port mentioned in the variable file.
       4. Task definition name = “*ecs-terraform-demo*”

This task definition will be updated on each Jenkins pipeline cycle.

As the task definition gets updated, it triggers ECS service, and all the existing tasks will be replaced with the new task definition instances.

* 1. esc-service.tf: - Create a ECS service with name “*ecs-demo-service-test*”, with desired count 2, which may change based on auto scaling group. All ECS task will be running in private subnets.
  2. application-load-balancer.tf: - Creates an application load balancer which listens to port 80 and forwards the request to the target group. This target group is used by ECS service to propagate to available tasks.
  3. auto-scaling-target.tf: - Creates auto-scaling target with max capacity as 3 and min capacity as 2, and a policy to trigger the auto scaling if average CPU utilization reaches 60%.

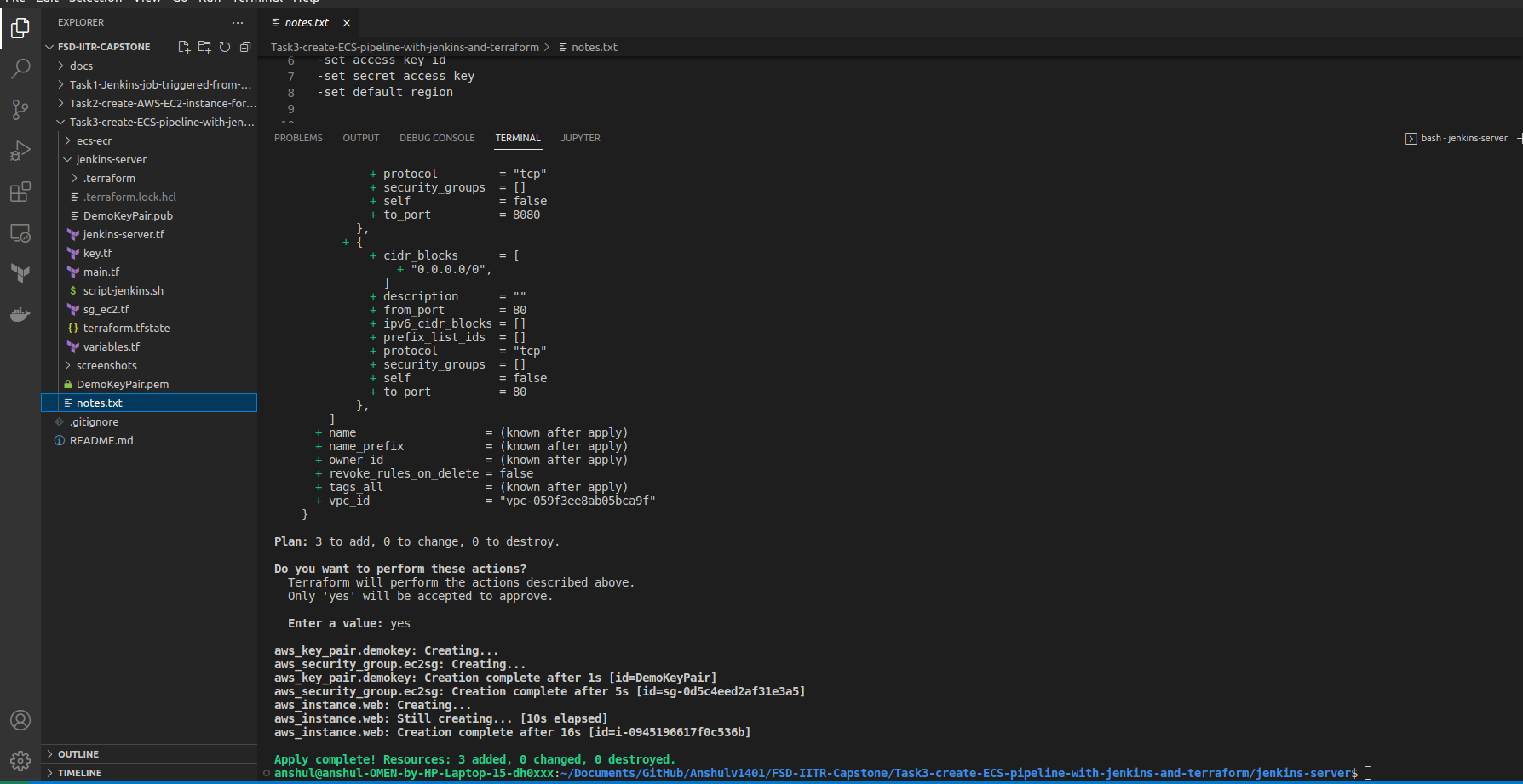
# Deployment Procedure

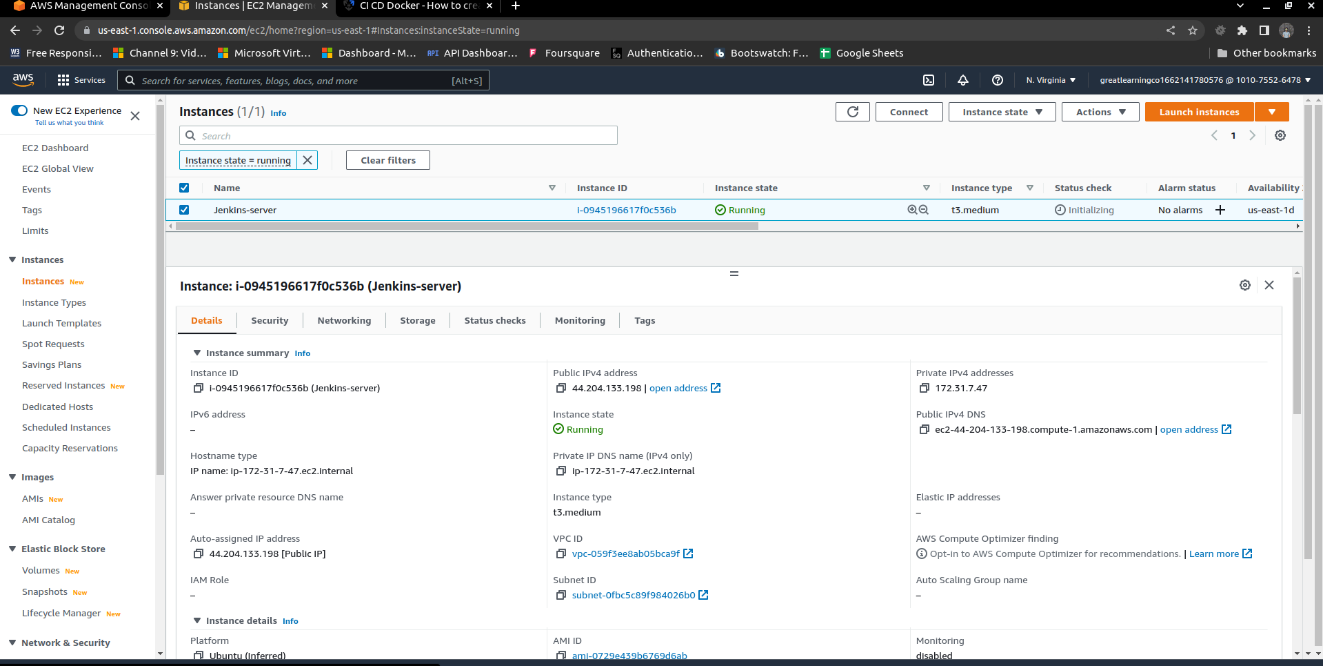
1. Jenkins-server **(t3.medium, Ubuntu EC2 instance)**
   1. On your system, in the repository that is shared, go inside folder Jenkins-server
   2. Run command *‘terraform init’* to initialize terraform.

Text

Description automatically generated

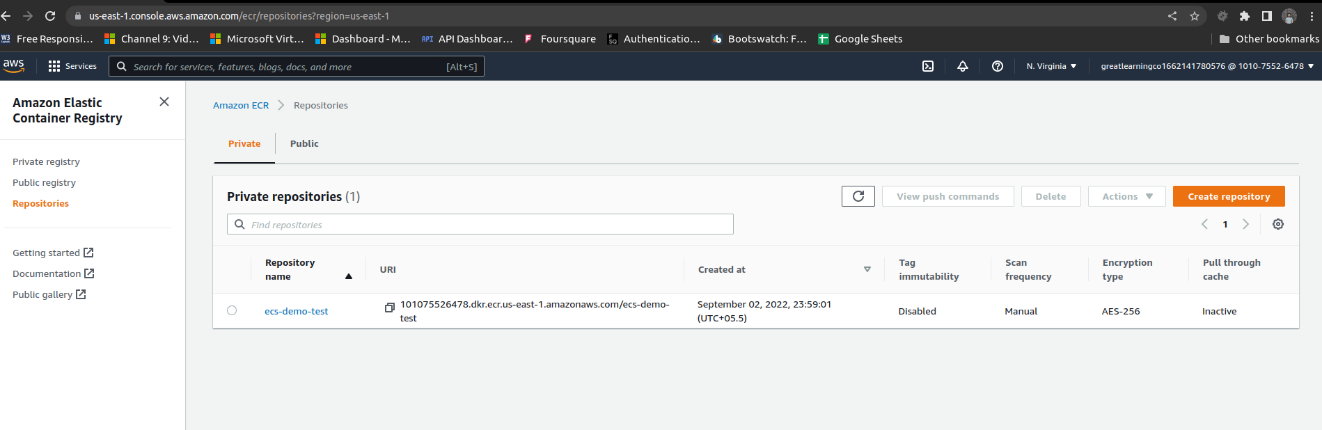
* 1. Run ‘*terraform apply’* to create ec2 instance for Jenkins-server.



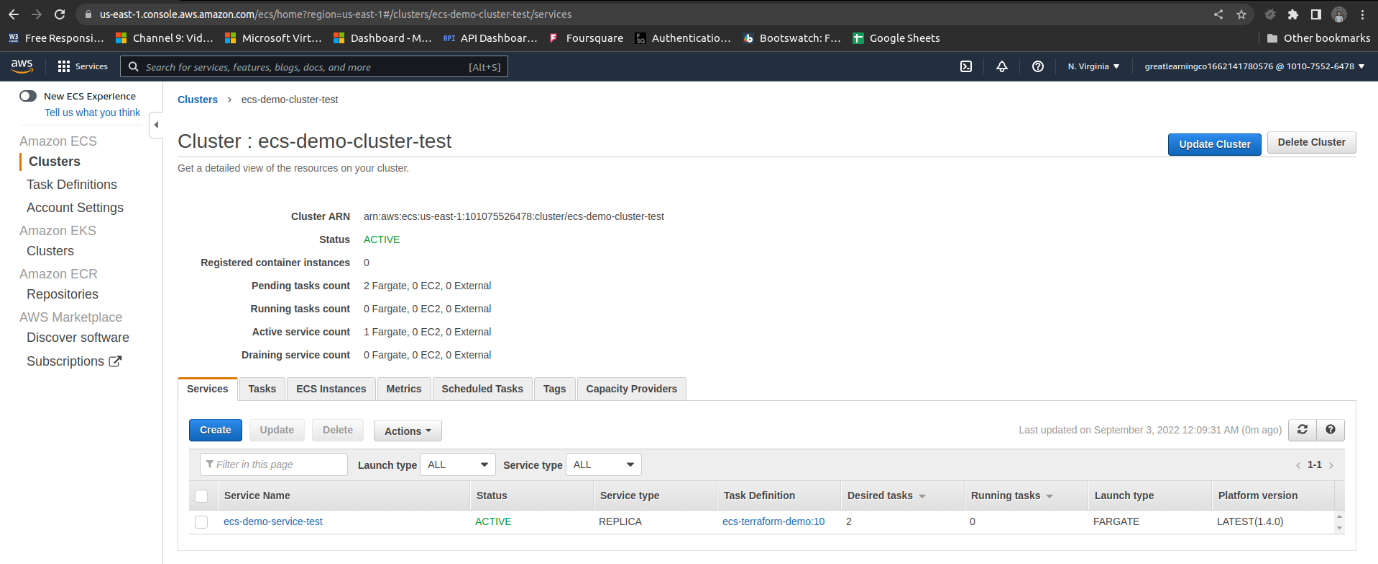


1. ECS-ECR infrastructure
   1. On your system, go inside folder ecs-ecr.
   2. Run command ‘*terraform init’* to initialize terraform.
   3. Run ‘*terraform apply’* to create ECS-ECR infrastructure with 1 VPC, 6 subnets, load balancer, Nat-gateway, internet-gateway, security groups, ECR, ECS cluster, ECS service and ECS task definition.

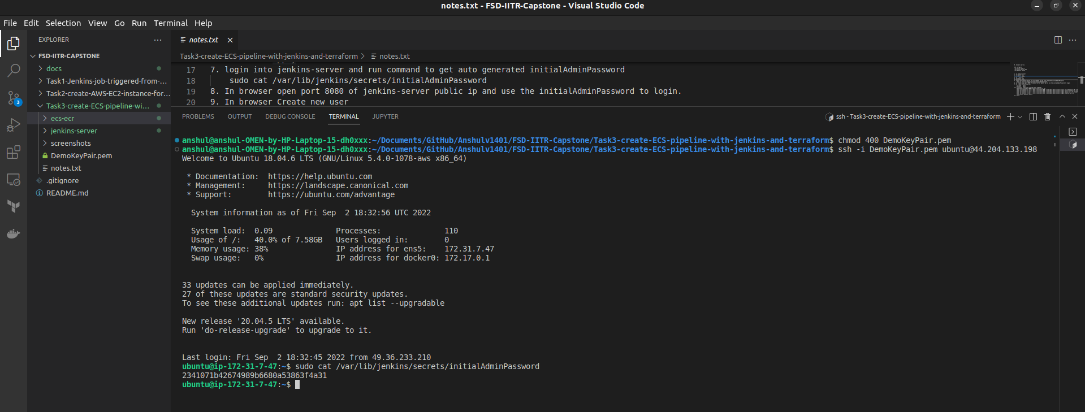
ECR repository created



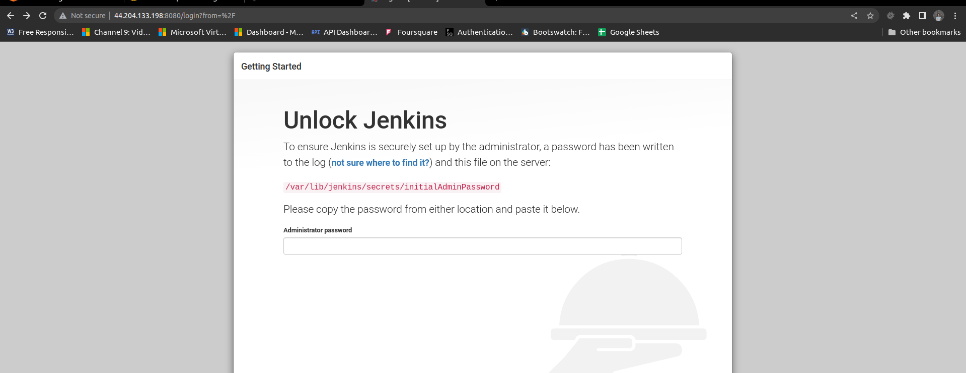
ECS Cluster created



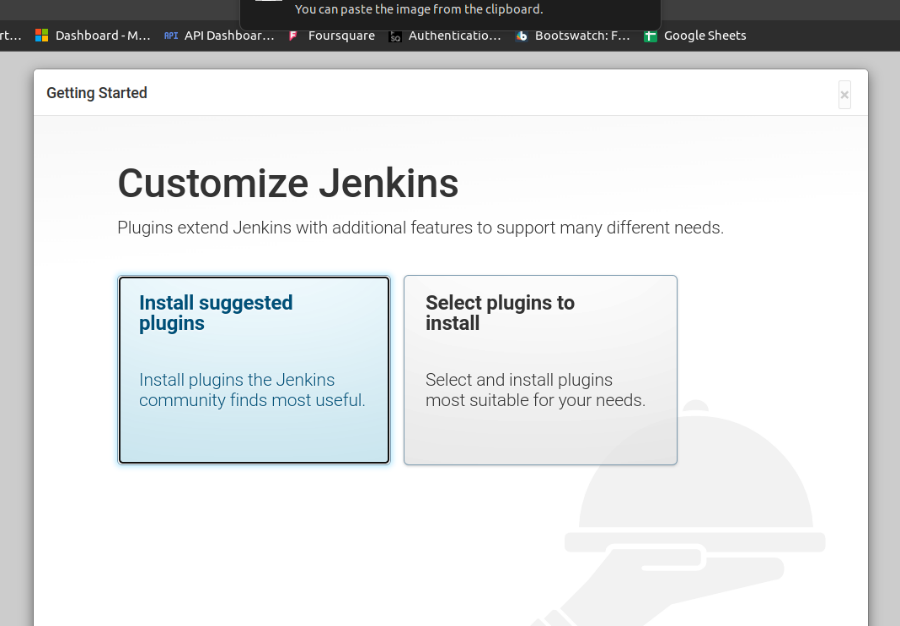
1. Jenkins-server configuration.
   1. Login into jenkins-server with the given ‘DemoKeyPair.pem’ file and run command ‘*sudo cat /var/lib/jenkins/secrets/initialAdminPassword*’ to get auto generated initialAdminPassword.



* 1. In browser open port 8080 of jenkins-server public IP and use the initialAdminPassword to login.



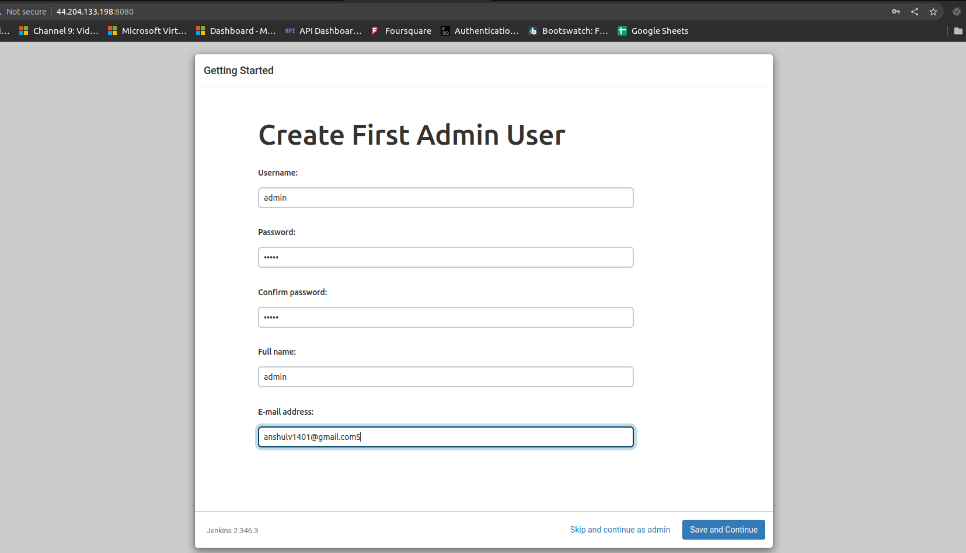
* 1. Install suggested plugins.



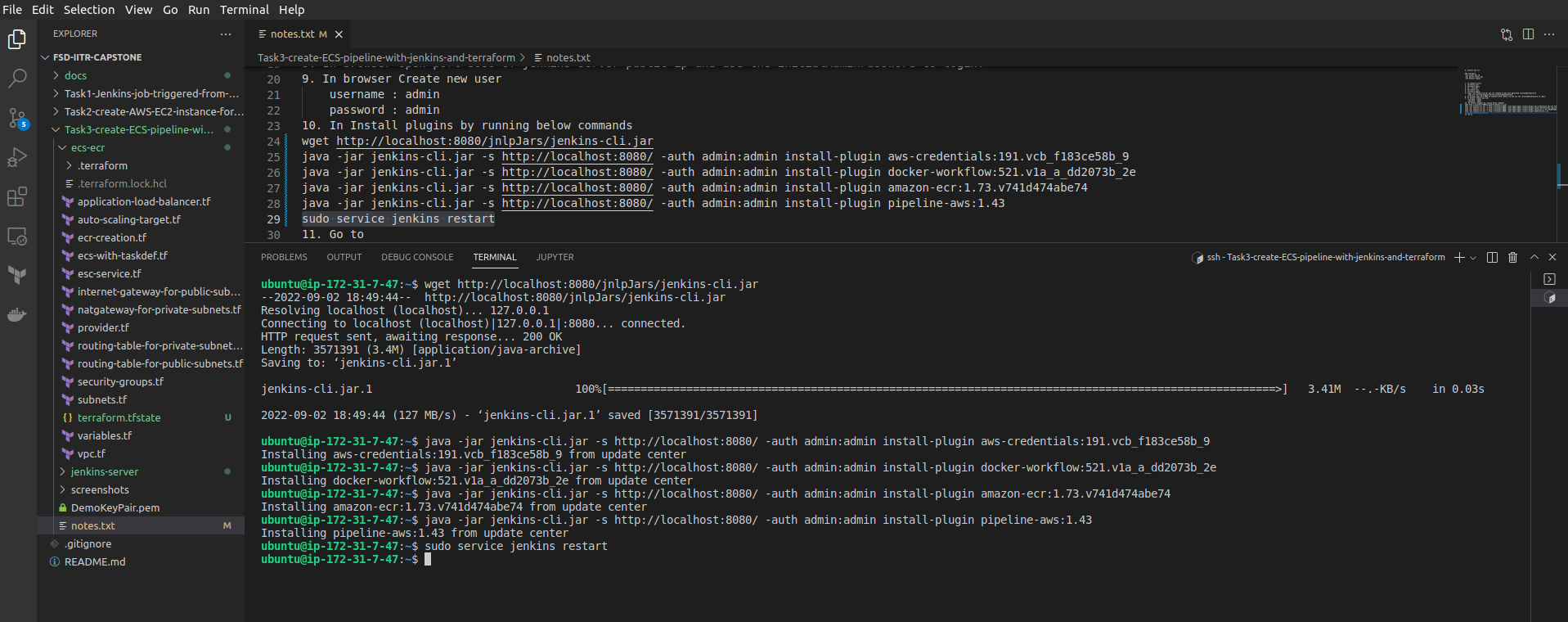
* 1. In browser, create new user

username: admin

password: admin



* 1. In Jenkins-server Install plugins by running below commands.
     1. wget http://localhost:8080/jnlpJars/jenkins-cli.jar
     2. java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:admin install-plugin aws-credentials:191.vcb\_f183ce58b\_9
     3. java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:admin install-plugin docker-workflow:521.v1a\_a\_dd2073b\_2e
     4. java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:admin install-plugin amazon-ecr:1.73.v741d474abe74
     5. java -jar jenkins-cli.jar -s http://localhost:8080/ -auth admin:admin install-plugin pipeline-aws:1.43
     6. sudo service jenkins restart



1. Credentials configuration on Jenkins.
   1. In browser, to go Manage Jenkins>Manage credentials, click “(global)” “Add credentials”.

Graphical user interface

Description automatically generated

* 1. Create AWS new credentials with below values.
     1. Kind: AWS Credentials
     2. Scope: Global
     3. ID: demo-admin-user
     4. AccessKeyId: <AccessKeyId>
     5. Secret Access Key: <Secret Access Key>

Graphical user interface, text, application, email

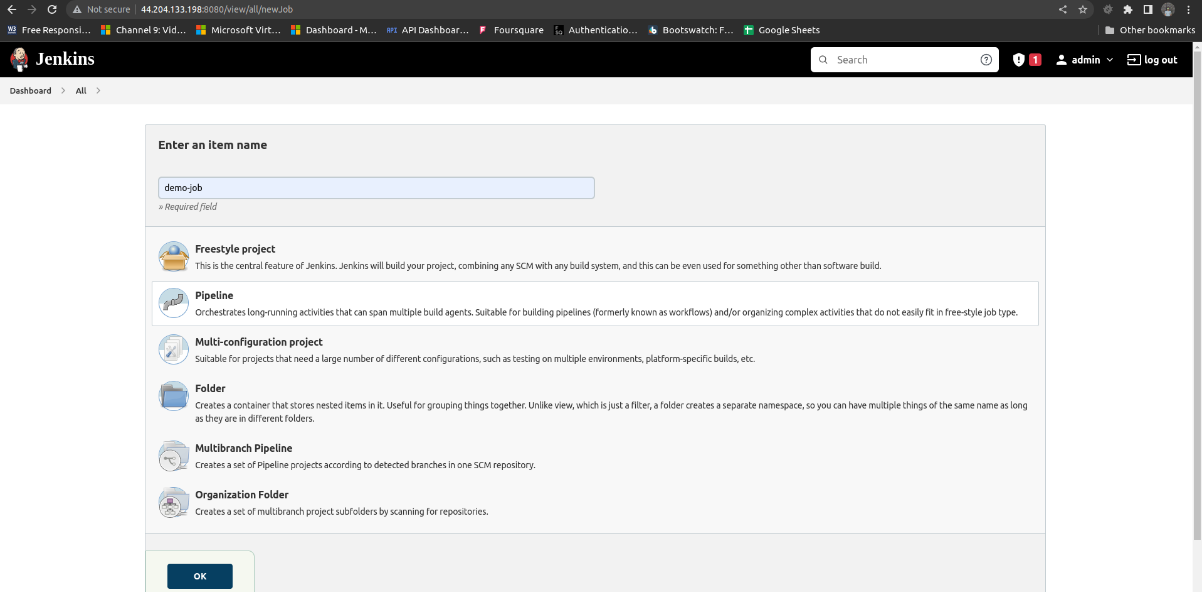
Description automatically generated

* 1. Create GitHub credentials with below values, Click on Add credentials
     1. Kind: username with password
     2. Scope: Global
     3. Username: <GitHub account username>
     4. Password: <GitHub account password>
     5. ID: <any value>

Graphical user interface, text, application, email

Description automatically generated

1. Create Jenkins Job
   1. Click on new Item
      1. name: demo-job
      2. item: pipeline



* 1. Job configuration
     1. Check “Discard old builds”, max = 4
     2. Check “GitHub project”, project URL = https://github.com/anshulv1401/emo-nodejs-app

Graphical user interface, text, application, email

Description automatically generated

* + 1. Under build trigger, check “GitHub hook trigger for GitScm polling”

Graphical user interface, text, application, email

Description automatically generated

* + 1. Under pipeline, select definition as “Pipeline script for SCM”
    2. Under scm select “Git”
    3. Under repository, give URL for git project. Example: “https://github.com/anshulv1401/emo-nodejs-app.
    4. For credentials selected already stored git credentials

Graphical user interface, text, application, email

Description automatically generated

* + 1. Save

1. Integrate GitHub and Jenkins
   1. In GitHub repo, under settings, select webhooks
   2. Click on Add webhook
   3. Give payload URL as “<jenkin-server-url>:8080/github-webhook/”
   4. example <http://44.204.129.138:8080//github-webhook/>

A screenshot of a computer

Description automatically generated with medium confidence

1. Jenkins file configuration.
   1. Update the jenkins file in github repo.
      1. AWS\_ACCOUNT\_ID="101075526478"
      2. AWS\_DEFAULT\_REGION="us-east-1"
      3. CLUSTER\_NAME="ecs-demo-cluster-test"
      4. SERVICE\_NAME="ecs-demo-service-test"
      5. TASK\_DEFINITION\_NAME="ecs-terraform-demo"
      6. DESIRED\_COUNT="4"
      7. IMAGE\_REPO\_NAME="ecs-demo-test"
      8. registryCredential = "demo-admin-user"

A screenshot of a computer

Description automatically generated

* 1. Push the changes.

1. Check the pipeline.
   1. In browser go to jenkins-job.
      1. On git push build should have automatically triggered

Graphical user interface, table

Description automatically generated

* 1. On pipeline completion, go to load balancers, and copy DNSName, and paste it in the browser.

Graphical user interface, text

Description automatically generated

1. Clean up
   1. In AWS console, goto Amazon ECR > Repositories > ecs-demo-test, and delete all the images

Graphical user interface, text, application, website

Description automatically generated

* 1. Navigate back to your local terminal, under folder esc-ecr and jenkins-server, run command “terraform destroy”.

A computer screen capture

Description automatically generated with medium confidence

# References

<https://www.clickittech.com/devops/ci-cd-docker/>

<https://www.architect.io/blog/2021-03-30/create-and-manage-an-aws-ecs-cluster-with-terraform/>

<https://engineering.finleap.com/posts/2020-02-20-ecs-fargate-terraform/>

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

# Revision Sheet

|  |  |  |
| --- | --- | --- |
| **Revision** | **Date** | **Description** |
| Version 1.0 | Sept 3, 2022 | Author Anshul Vanawat |
|  |  |  |
|  |  |  |