AWS PROJECT

**Project Overview**

This project involves setting up a robust CI/CD pipeline using Jenkins, Docker, AWS Elastic Container Registry (ECR) and Terraform to deploy a containerized Node.js application.

The primary objectives include:

**1. Development Environment Setup:** Preparing the local development environment for building and testing the application.

**2. Infrastructure Setup:** Configuring AWS resources such as Application Load Balancers (ALB), Target Groups, and ECR repositories using Terraform.

**3. Application Containerization:** Dockerizing a sample Node.js application (Movies App) to prepare it for deployment.

**4. CI/CD Pipeline:** Implementing a Jenkins pipeline that automates the build, test, and deployment processes, including pushing Docker images to ECR and deploying them to an application server

**GITHUB PROJECT LINK - https://github.com/ping20/AWS\_Project/tree/main**

**Task 1: Setup infrastructure(infra) using Terraform (tf)**

**Objective:**

This is a foundational task to set up infrastructure using Terraform. Prepare your local development environment to develop, build, and test the Movies App. This includes installing necessary tools, cloning the repository, and ensuring the application runs correctly before containerization.

**Subtask 1: Configure AWS CLI and Create S3 Backend for Terraform**

Step 1: Install and Configure AWS CLI on Linux

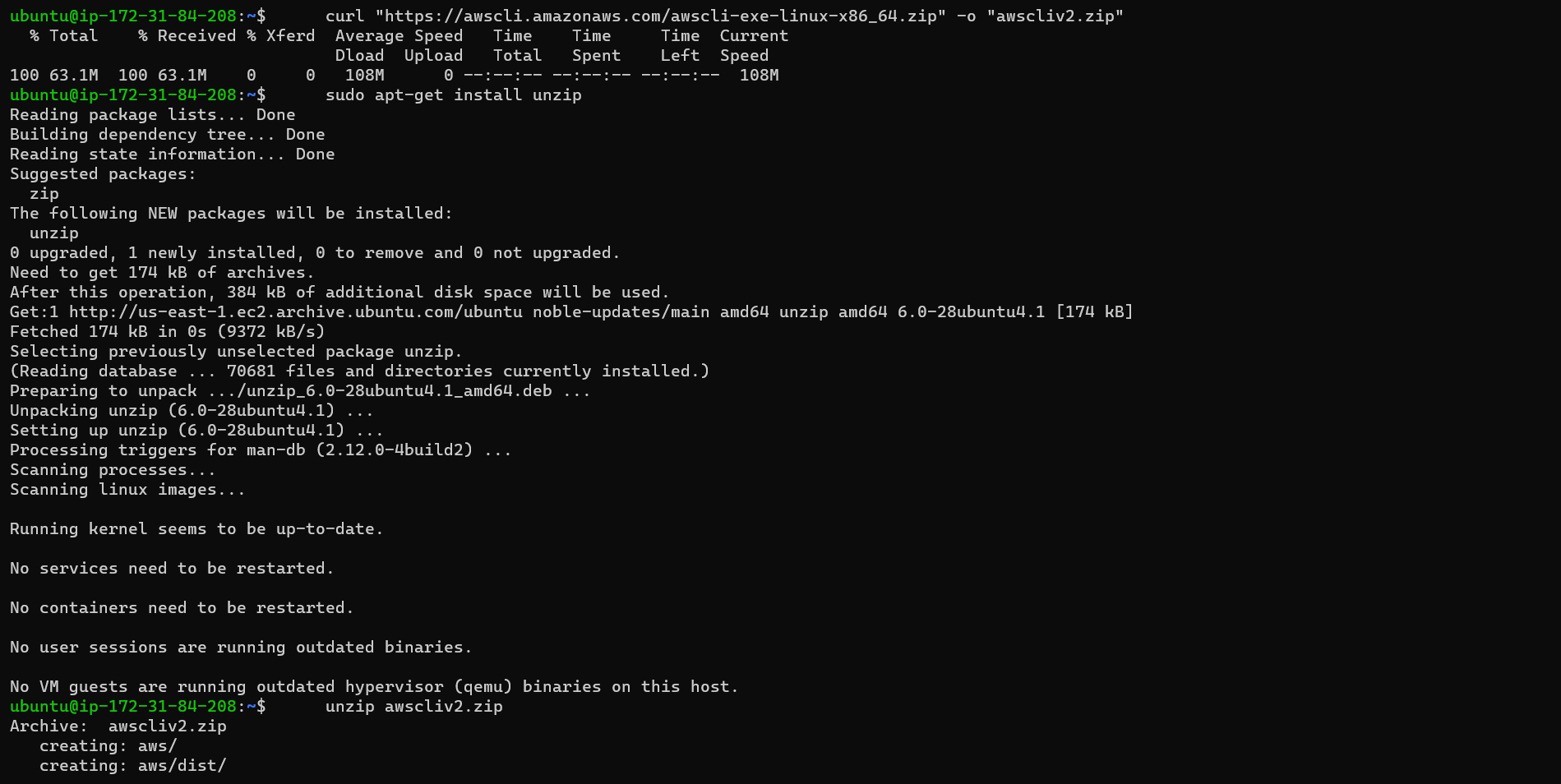
# Install AWS CLI (if not already)

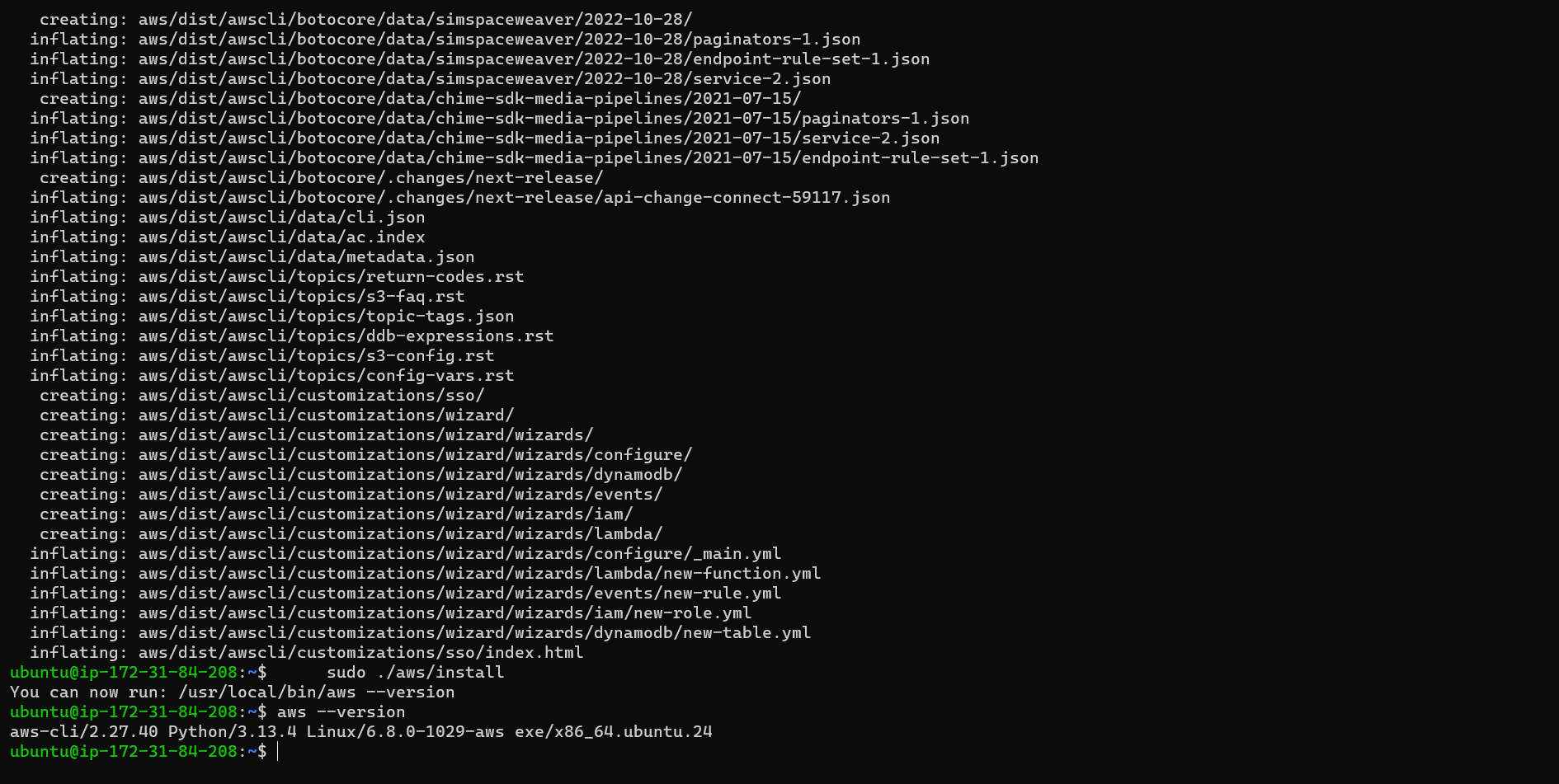
sudo apt update

sudo apt install awscli -y

# Verify installation

aws –version

* Install AWS CLI on our Linux Machine



* Create S3 Bucket using Terraform

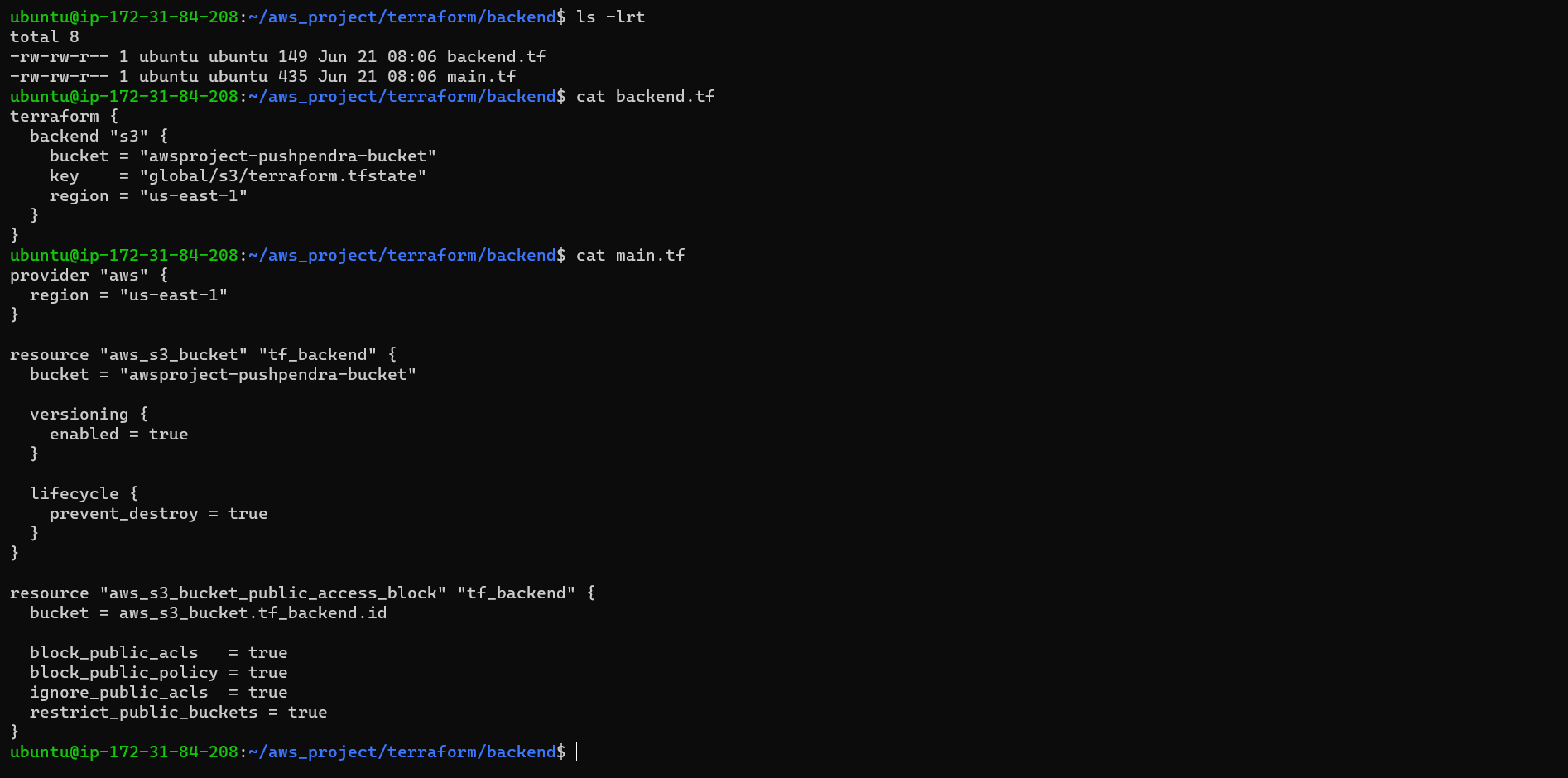
📁 Create Terraform project directory structure:

📁 Terraform Directory Structure

terraform/

└── backend/

├── main.tf

 ├── backend.tf

**📄 main.tf**

provider "aws" {

region = "us-east-1"

}

resource "aws\_s3\_bucket" "tf\_backend" {

bucket = "awsproject-pushpendra-bucket"

}

resource "aws\_s3\_bucket\_versioning" "tf\_backend\_versioning" {

bucket = aws\_s3\_bucket.tf\_backend.id

versioning\_configuration {

status = "Enabled"

}

}

resource "aws\_s3\_bucket\_public\_access\_block" "tf\_backend" {

bucket = aws\_s3\_bucket.tf\_backend.id

block\_public\_acls = true

block\_public\_policy = true

ignore\_public\_acls = true

restrict\_public\_buckets = true

}

**📄 backend.tf**

terraform {

backend "s3" {

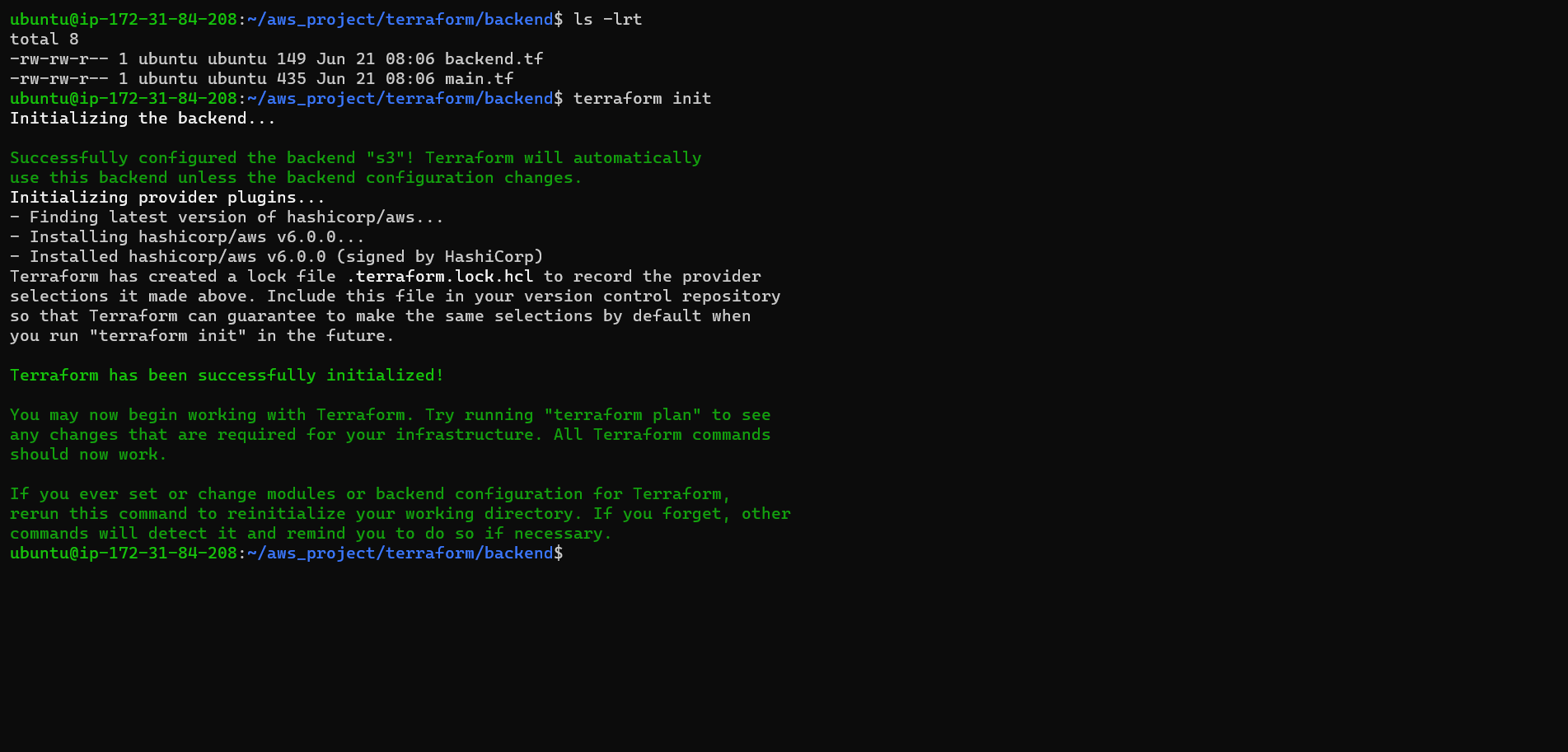
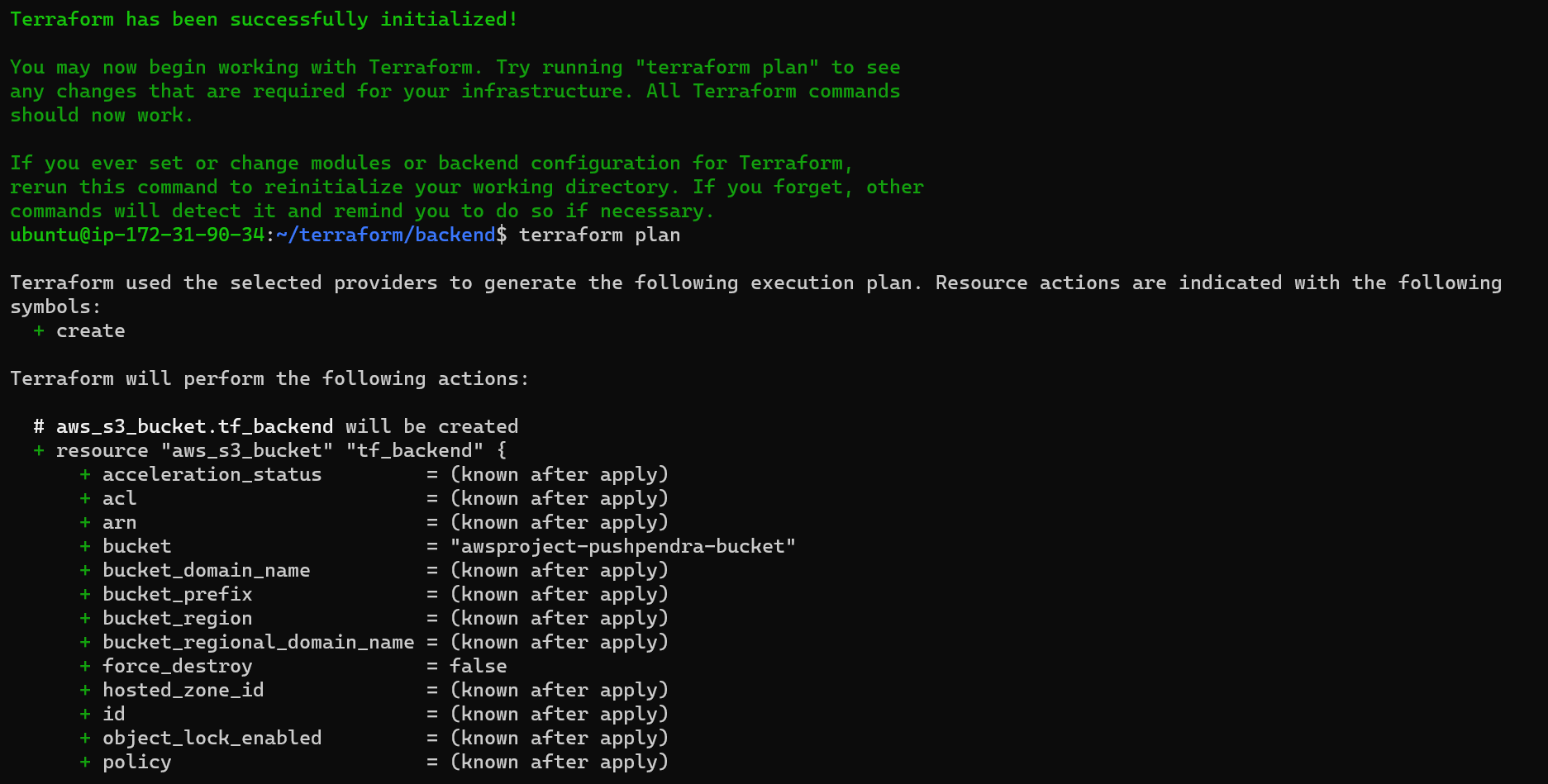
bucket = "awsproject-pushpendra-bucket"

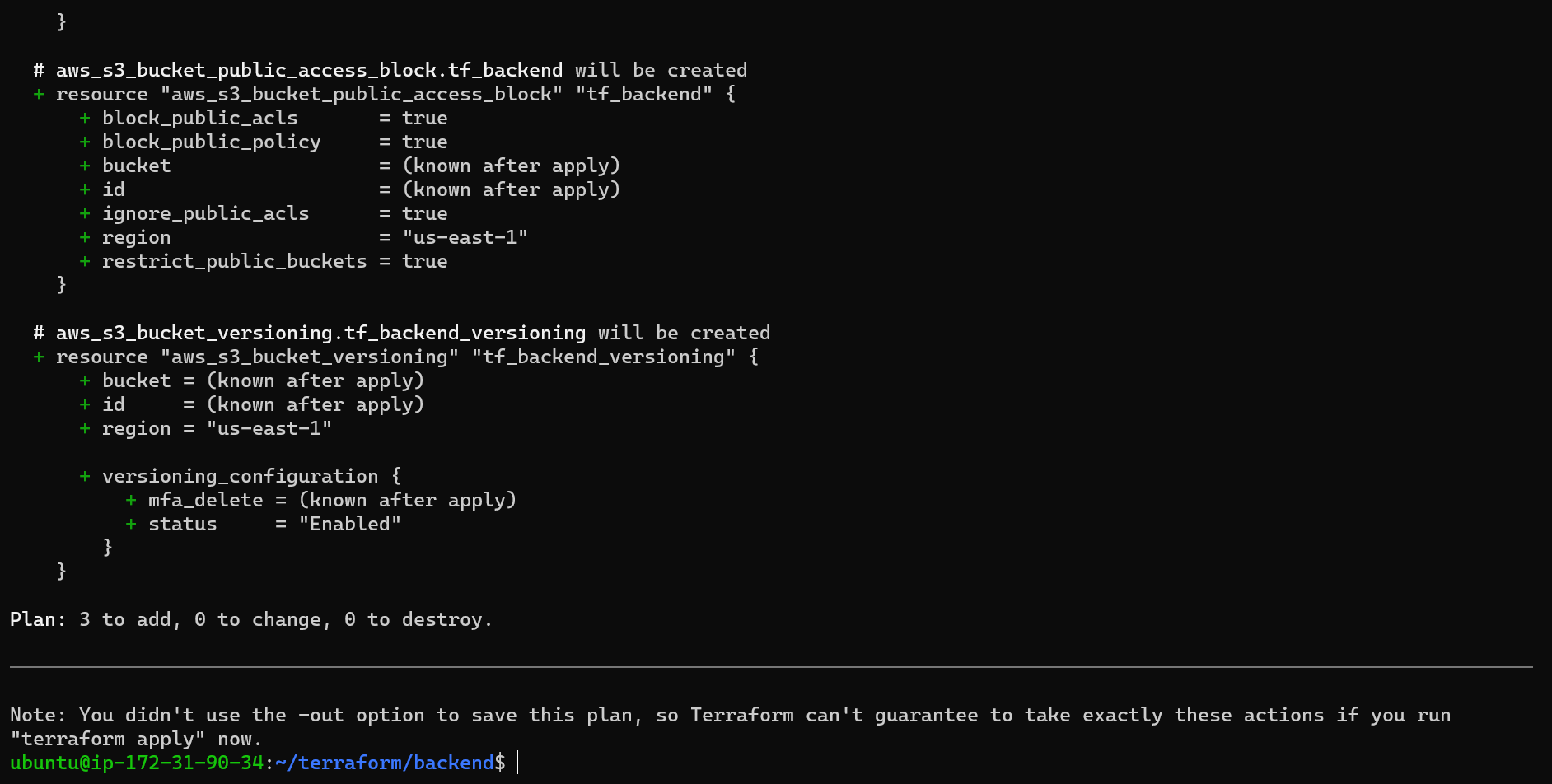
key = "global/s3/terraform.tfstate"

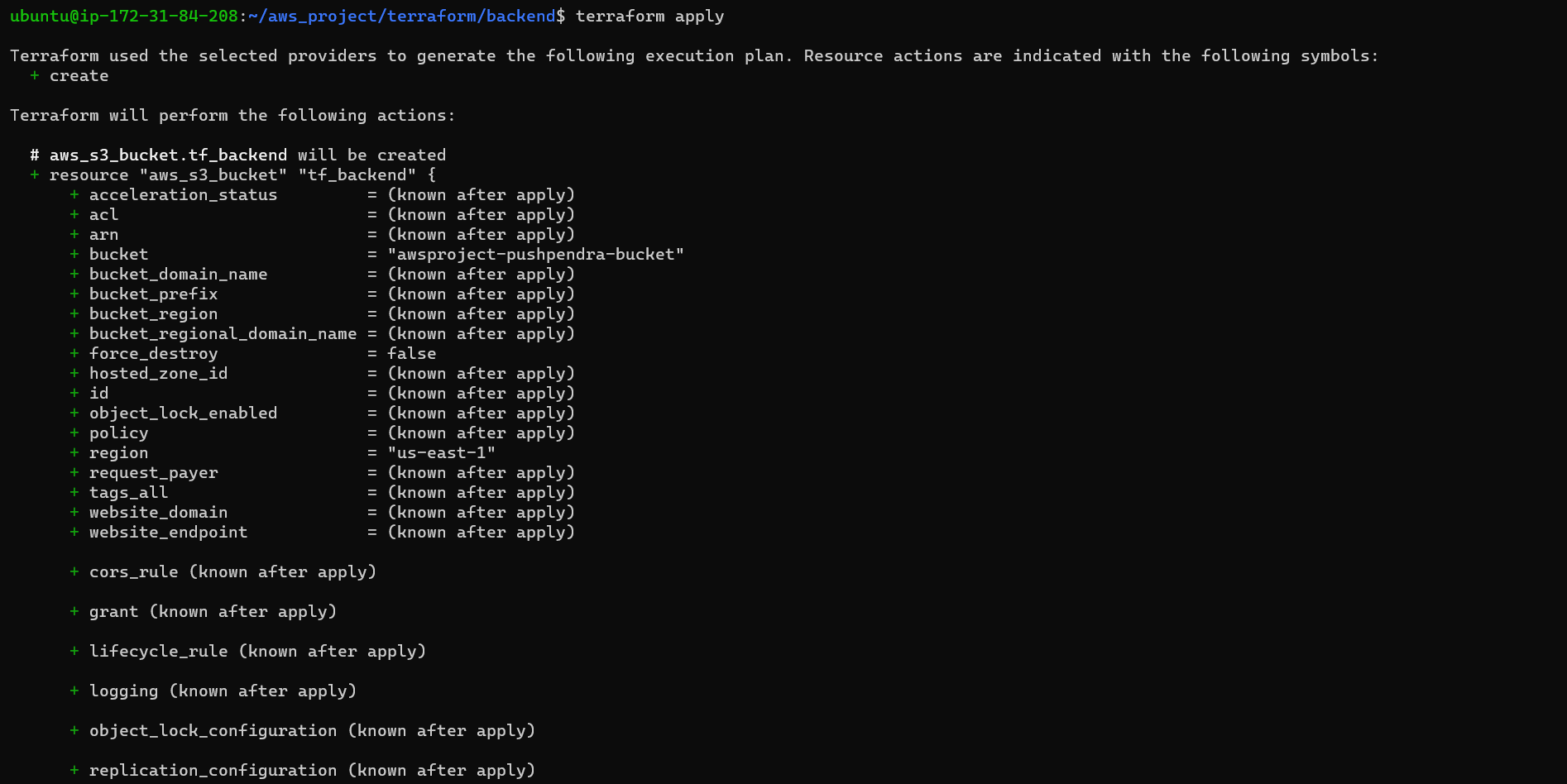
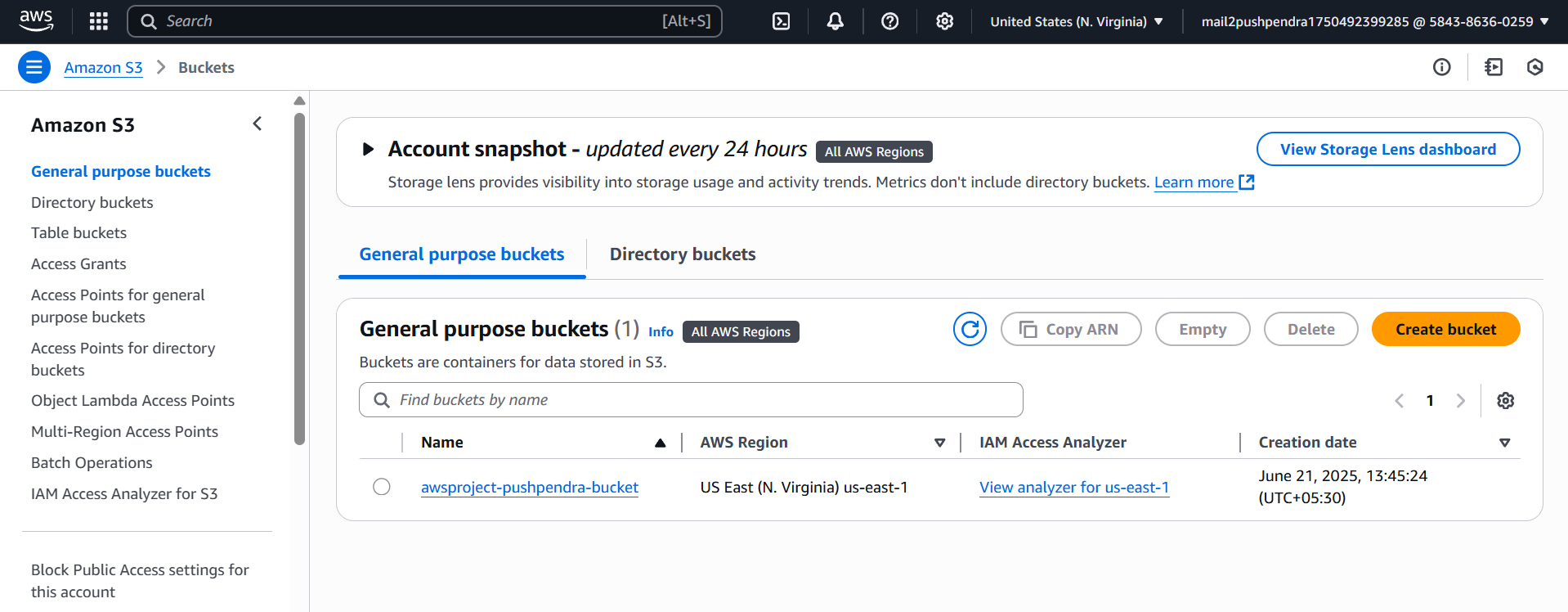
region = "us-east-1"

}

}

* 💻 Run Terraform - terraform init
* terraform plan



* terraform apply
* Created S3 Bucket using Terraform

**Subtask 2: VPC Setup, NAT Gateway, Subnets, Route Tables**

📁 Terraform Directory Structure:

terraform/

└── vpc/

├── main.tf

├── variables.tf

├── outputs.tf

**📄 main.tf**

provider "aws" {

region = var.region

}

resource "aws\_vpc" "main" {

cidr\_block = var.vpc\_cidr

tags = {

Name = "vpc-${var.project\_code}"

}

}

resource "aws\_internet\_gateway" "igw" {

vpc\_id = aws\_vpc.main.id

tags = {

Name = "igw-${var.project\_code}"

}

}

resource "aws\_subnet" "public" {

count = length(var.public\_subnet\_cidrs)

vpc\_id = aws\_vpc.main.id

cidr\_block = var.public\_subnet\_cidrs[count.index]

availability\_zone = element(["us-east-1a", "us-east-1b"], count.index)

map\_public\_ip\_on\_launch = true

tags = {

Name = "public-subnet-${count.index + 1}-${var.project\_code}"

}

}

resource "aws\_subnet" "private" {

count = length(var.private\_subnet\_cidrs)

vpc\_id = aws\_vpc.main.id

cidr\_block = var.private\_subnet\_cidrs[count.index]

availability\_zone = element(["us-east-1a", "us-east-1b"], count.index)

tags = {

Name = "private-subnet-${count.index + 1}-${var.project\_code}"

}

}

resource "aws\_eip" "nat" {

depends\_on = [aws\_internet\_gateway.igw]

}

resource "aws\_nat\_gateway" "nat\_gw" {

allocation\_id = aws\_eip.nat.id

subnet\_id = aws\_subnet.public[0].id

tags = {

Name = "nat-gateway-${var.project\_code}"

}

}

resource "aws\_route\_table" "public" {

vpc\_id = aws\_vpc.main.id

route {

cidr\_block = "0.0.0.0/0"

gateway\_id = aws\_internet\_gateway.igw.id

}

tags = {

Name = "public-rt-${var.project\_code}"

}

}

resource "aws\_route\_table\_association" "public" {

count = length(var.public\_subnet\_cidrs)

subnet\_id = aws\_subnet.public[count.index].id

route\_table\_id = aws\_route\_table.public.id

}

resource "aws\_route\_table" "private" {

vpc\_id = aws\_vpc.main.id

route {

cidr\_block = "0.0.0.0/0"

nat\_gateway\_id = aws\_nat\_gateway.nat\_gw.id

}

tags = {

Name = "private-rt-${var.project\_code}"

}

}

resource "aws\_route\_table\_association" "private" {

count = length(var.private\_subnet\_cidrs)

subnet\_id = aws\_subnet.private[count.index].id

route\_table\_id = aws\_route\_table.private.id

}

**📄 output.tf**

output "vpc\_id" {

value = aws\_vpc.main.id

}

output "public\_subnets" {

value = aws\_subnet.public[\*].id

}

output "private\_subnets" {

value = aws\_subnet.private[\*].id

}

**📄 security\_groups.tf**

data "http" "my\_ip" {

url = "http://ipv4.icanhazip.com"

}

resource "aws\_security\_group" "bastion\_sg" {

name = "bastion-sg-${var.project\_code}"

vpc\_id = aws\_vpc.main.id

ingress {

from\_port = 22

to\_port = 22

protocol = "tcp"

cidr\_blocks = ["${chomp(data.http.my\_ip.response\_body)}/32"]

}

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = {

Name = "bastion-sg-${var.project\_code}"

}

}

resource "aws\_security\_group" "private\_sg" {

name = "private-sg-${var.project\_code}"

vpc\_id = aws\_vpc.main.id

ingress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = [var.vpc\_cidr]

}

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = {

Name = "private-sg-${var.project\_code}"

}

}

resource "aws\_security\_group" "web\_sg" {

name = "web-sg-${var.project\_code}"

vpc\_id = aws\_vpc.main.id

ingress {

from\_port = 80

to\_port = 80

protocol = "tcp"

cidr\_blocks = ["${chomp(data.http.my\_ip.response\_body)}/32"]

}

egress {

from\_port = 0

to\_port = 0

protocol = "-1"

cidr\_blocks = ["0.0.0.0/0"]

}

tags = {

Name = "web-sg-${var.project\_code}"

}

}

**📄 variables.tf**

variable "project\_code" {

default = "aws-project"

}

variable "region" {

default = "us-east-1"

}

variable "vpc\_cidr" {

default = "10.0.0.0/16"

}

variable "public\_subnet\_cidrs" {

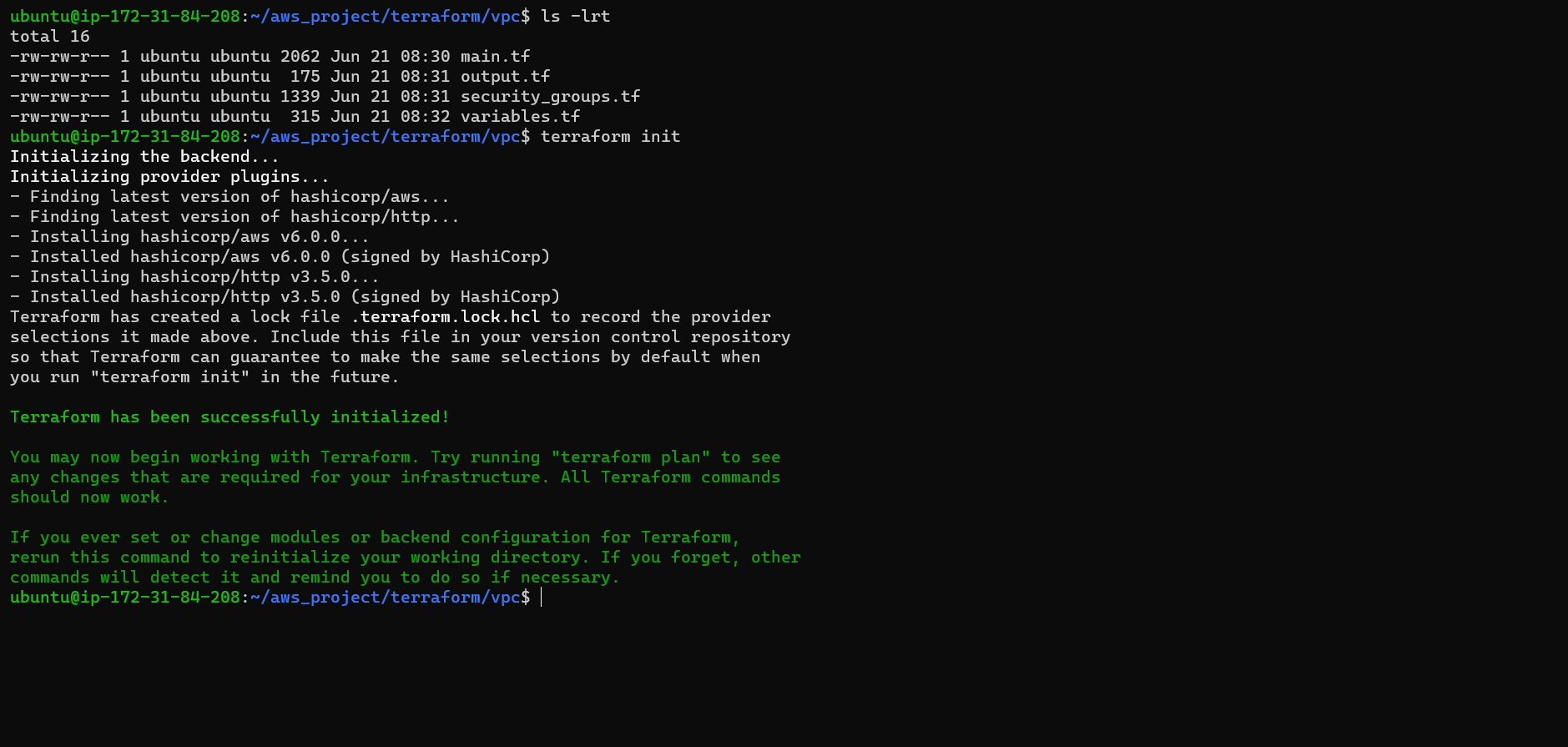
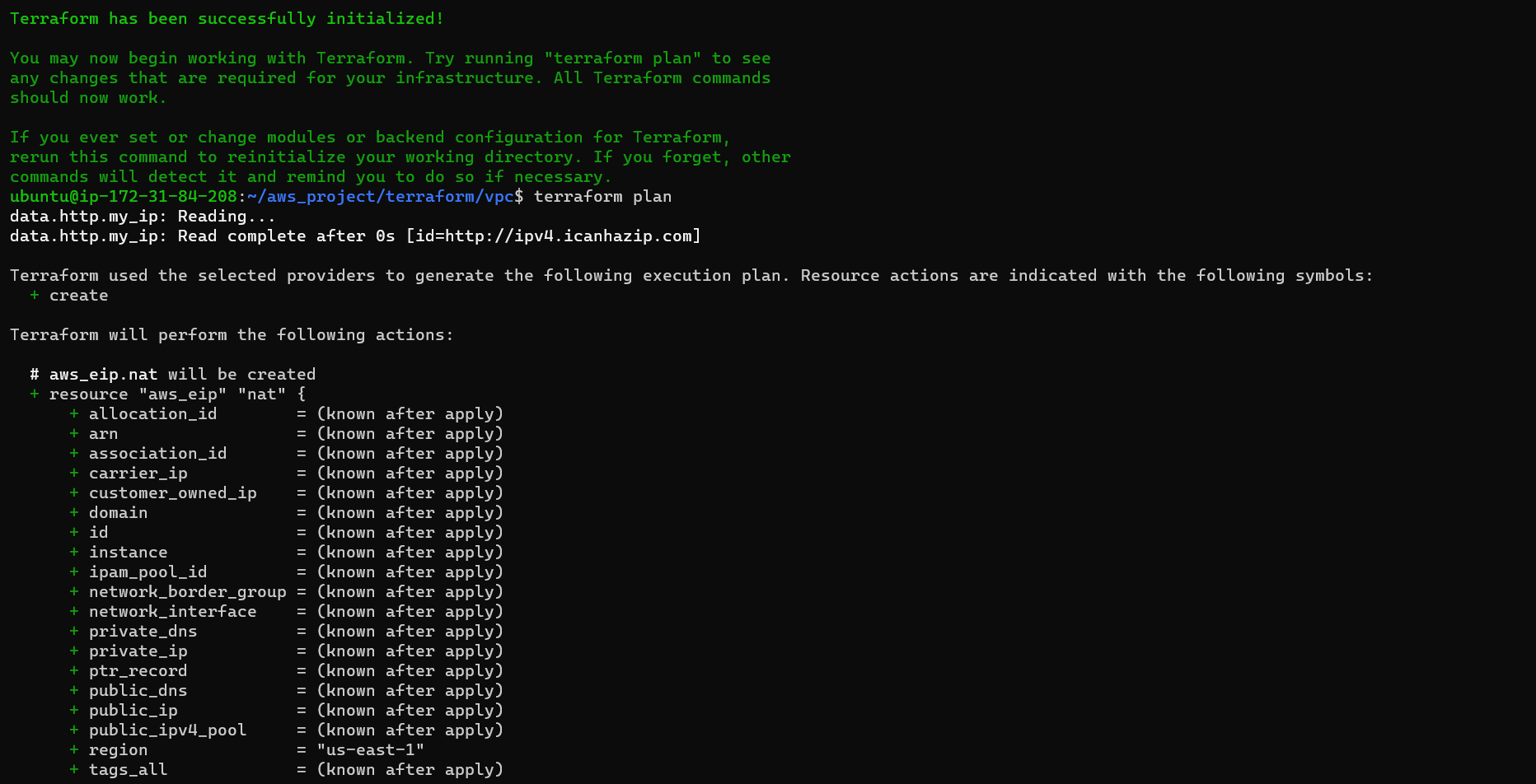
default = ["10.0.1.0/24", "10.0.2.0/24"]

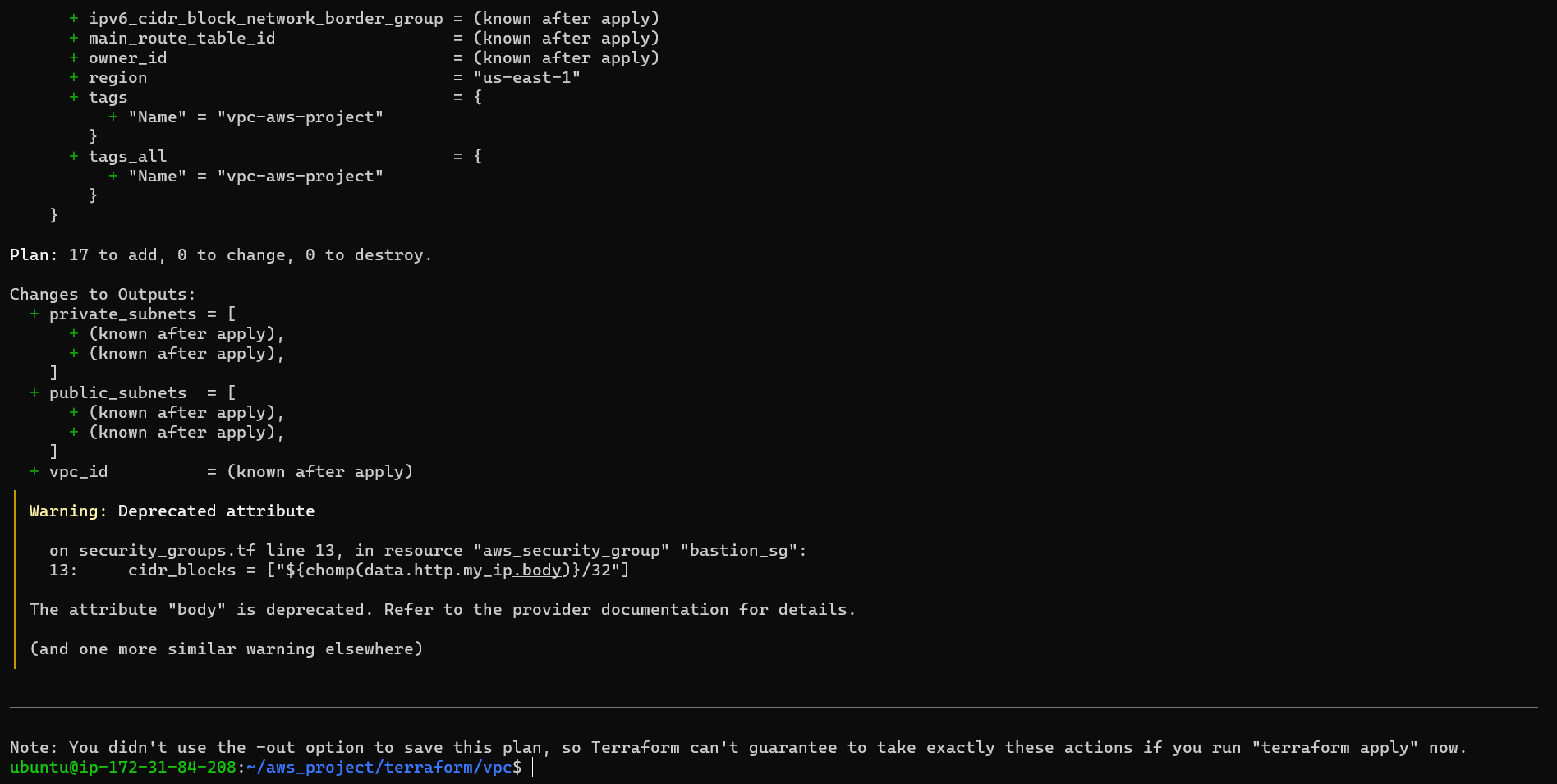
}

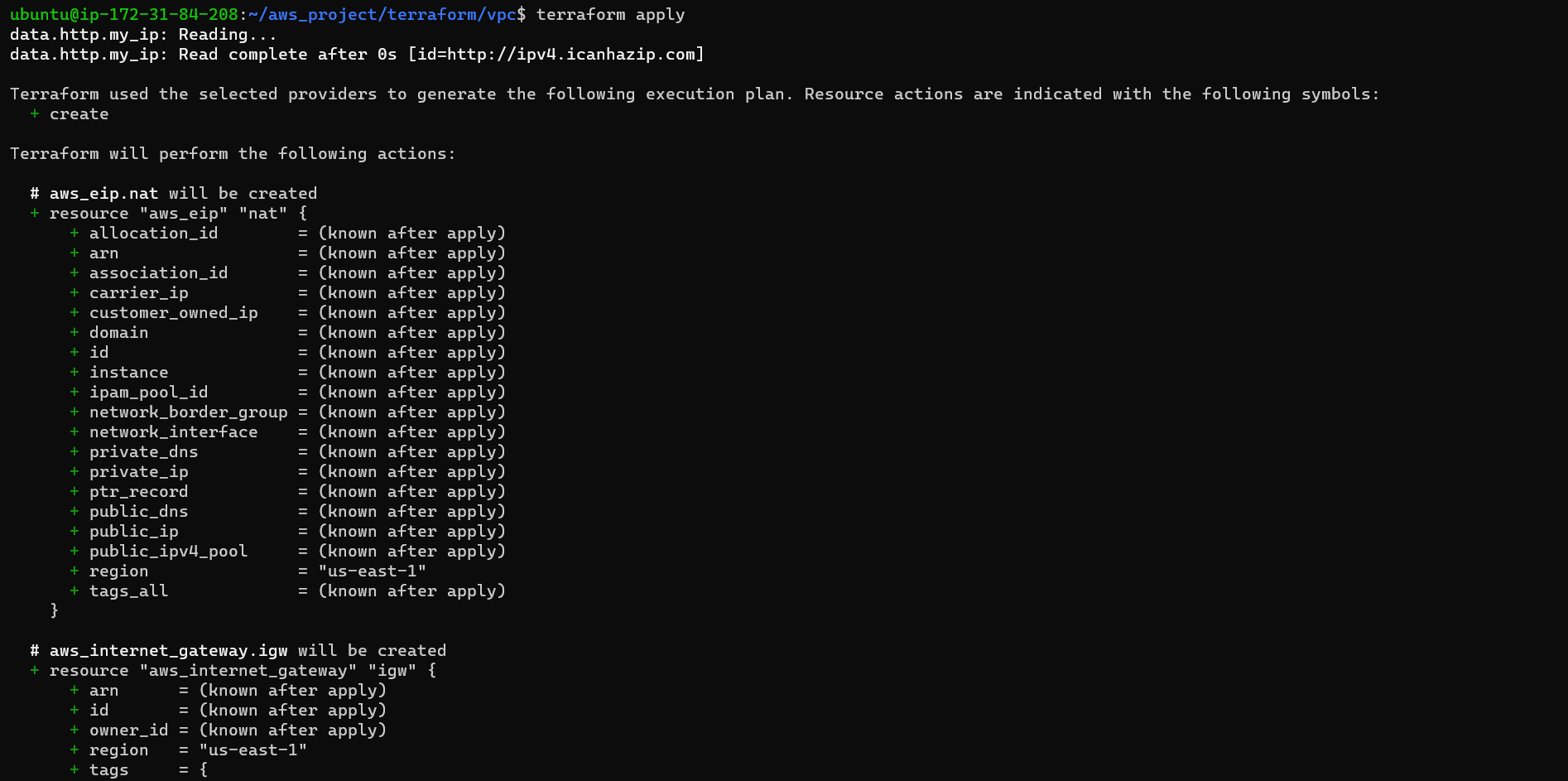
variable "private\_subnet\_cidrs" {

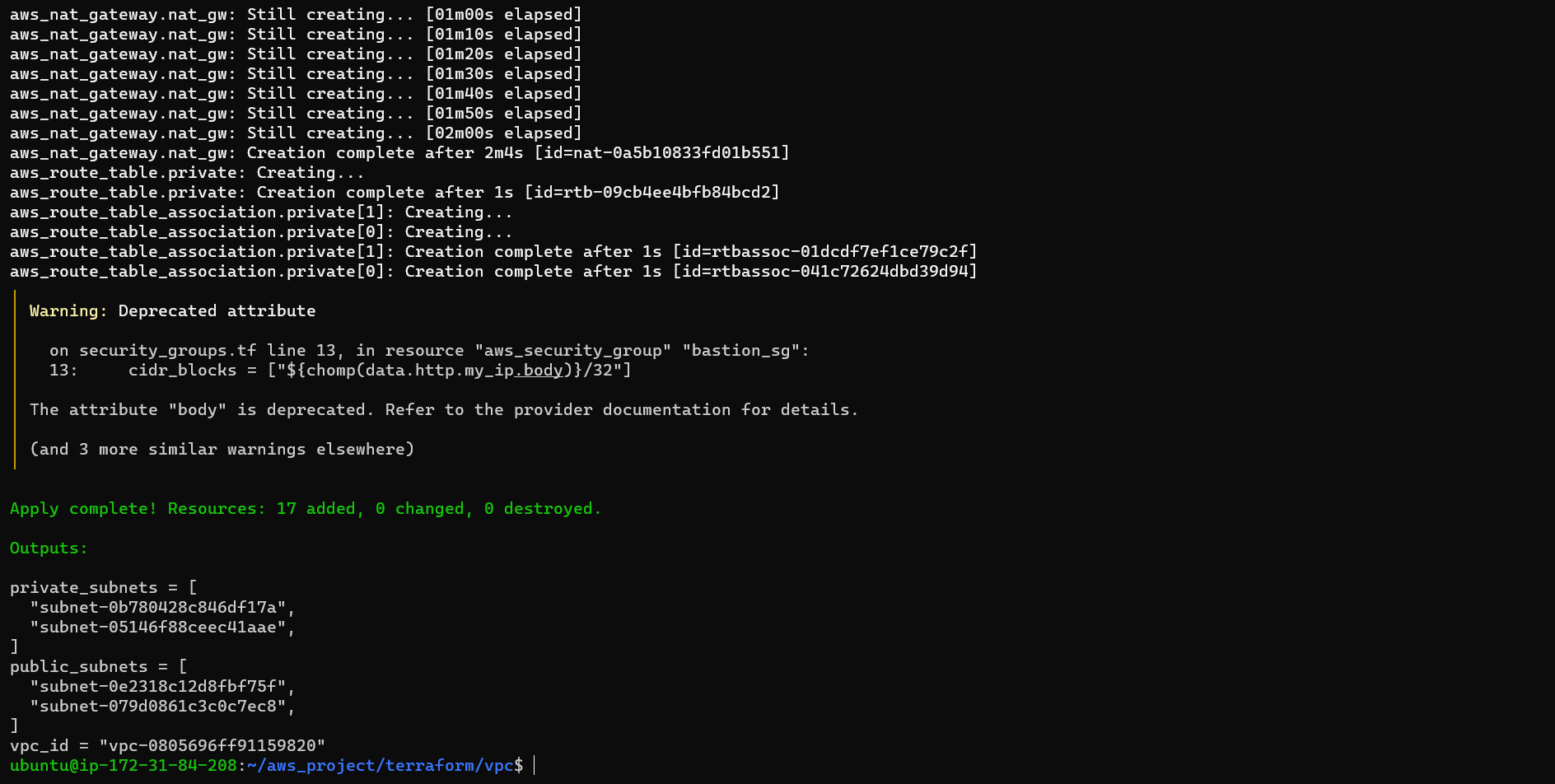
default = ["10.0.101.0/24", "10.0.102.0/24"]

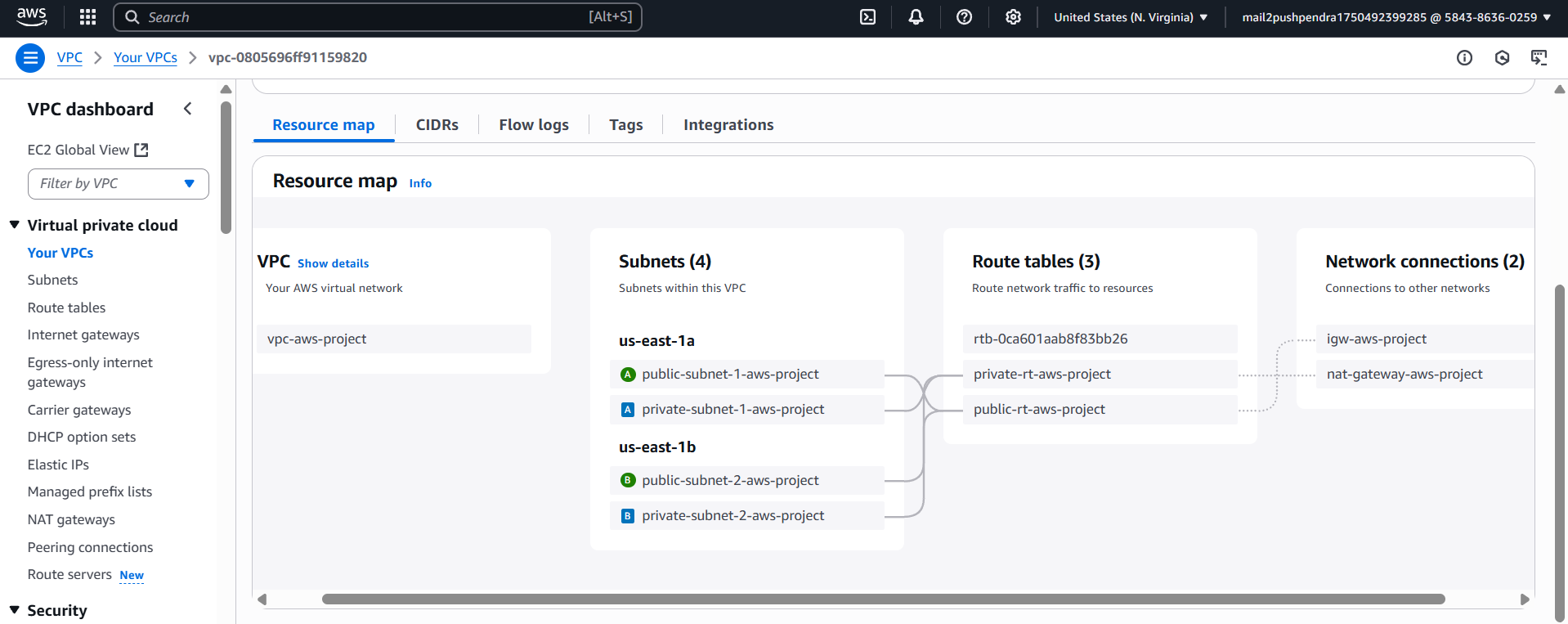
}

* **💻** Deploy: Run Terraform - terraform init
* terraform plan

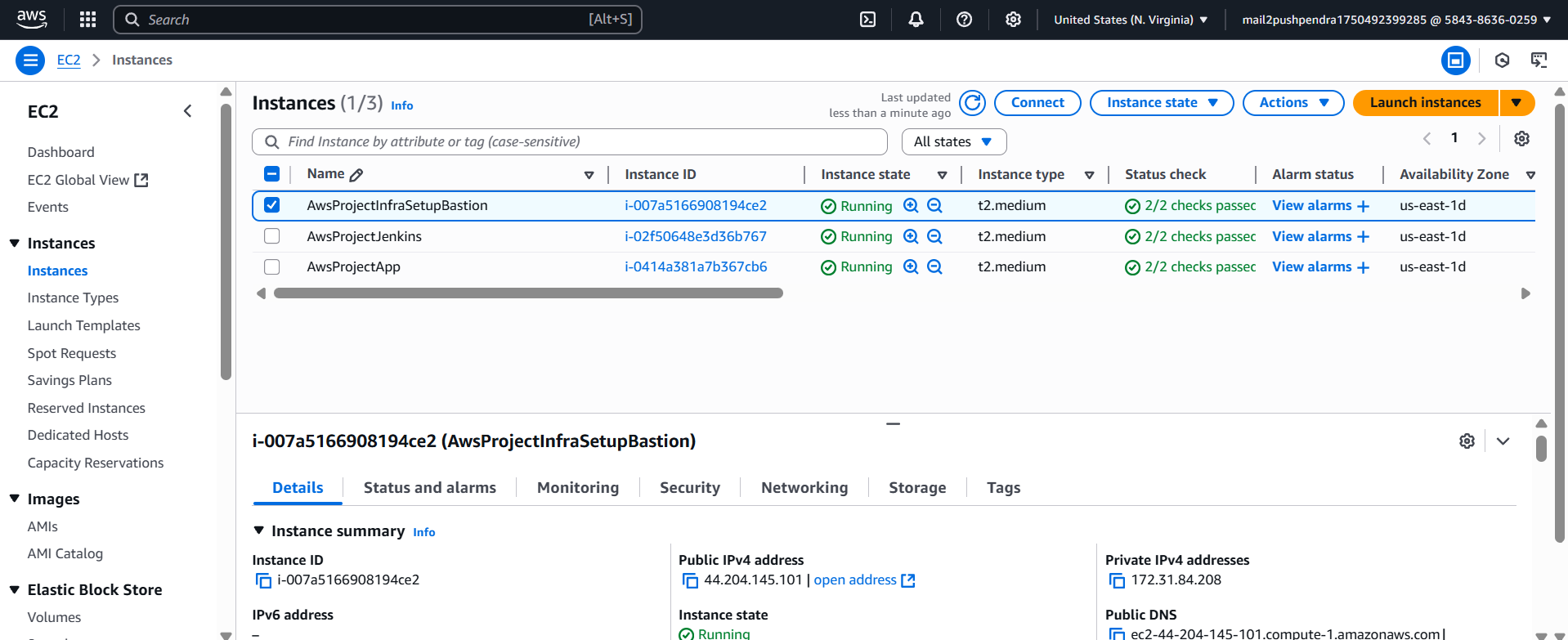
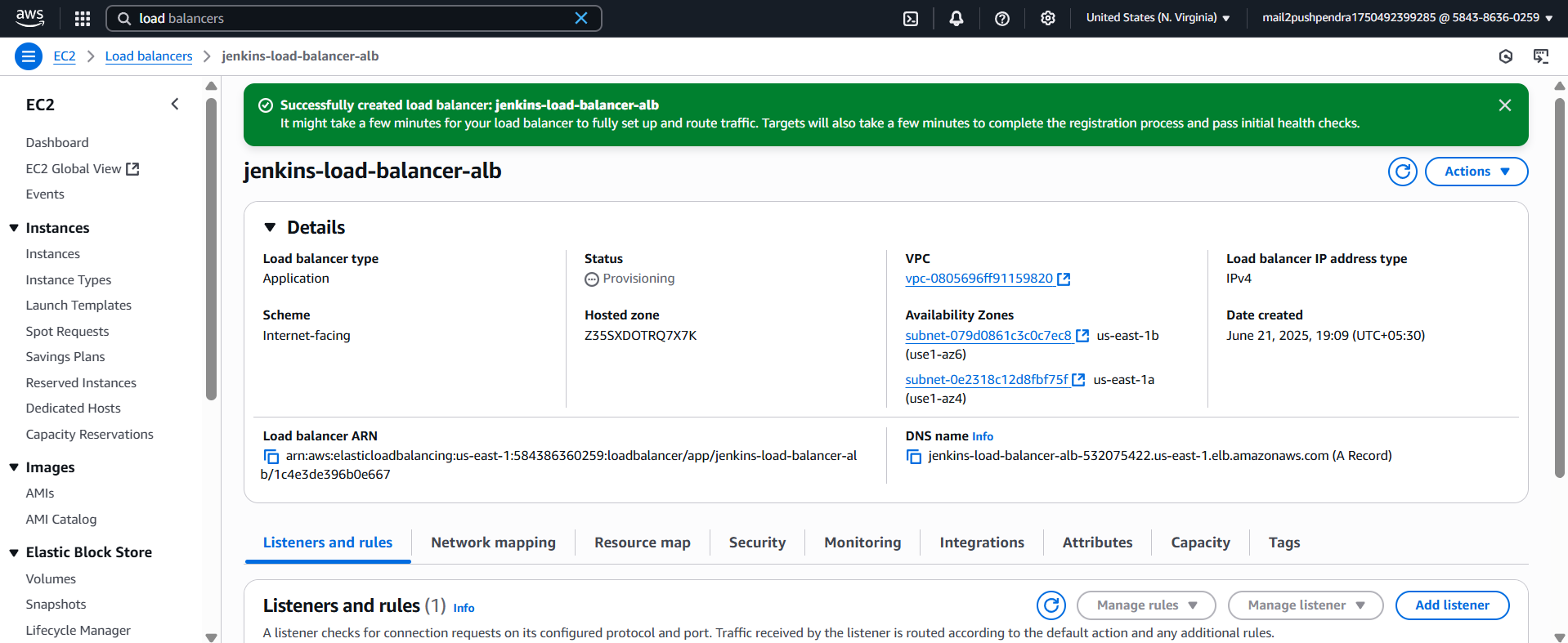


* terraform apply

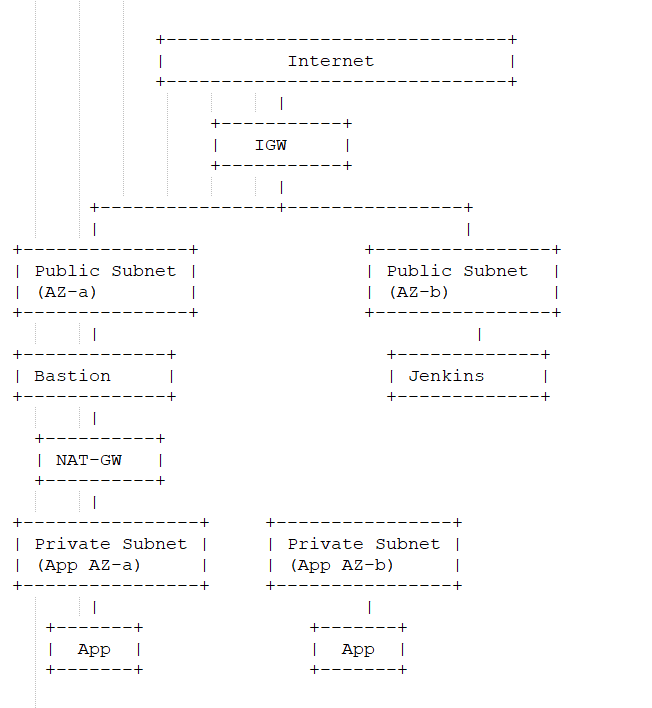


* Created all required resources using Terraform

**Subtask 3: Created EC2 instances (bastion, Jenkins, app) & ALB**

* ****Created EC2 instances (bastion, Jenkins, app)
* Created Application Load Balancer (ALB)
* Security Groups (SGs) for the following resources:  Created in Sub-Task 2 itself in security\_groups.tf file as done above. (see subtask 2 security\_groups.tf file for reference)

**Bonus: VPC Architecture Diagram**

****

**Task 2: Setup Config Management for hosts using Ansible & CI pipeline using Jenkins**

**Objective:**

This is Task 2 in our DevOps project, which focuses on Configuration Management (Ansible) and CI/CD Setup (Jenkins, ALB, ECR).

**Subtask 1: Install Ansible and Bootstrap Configuration**

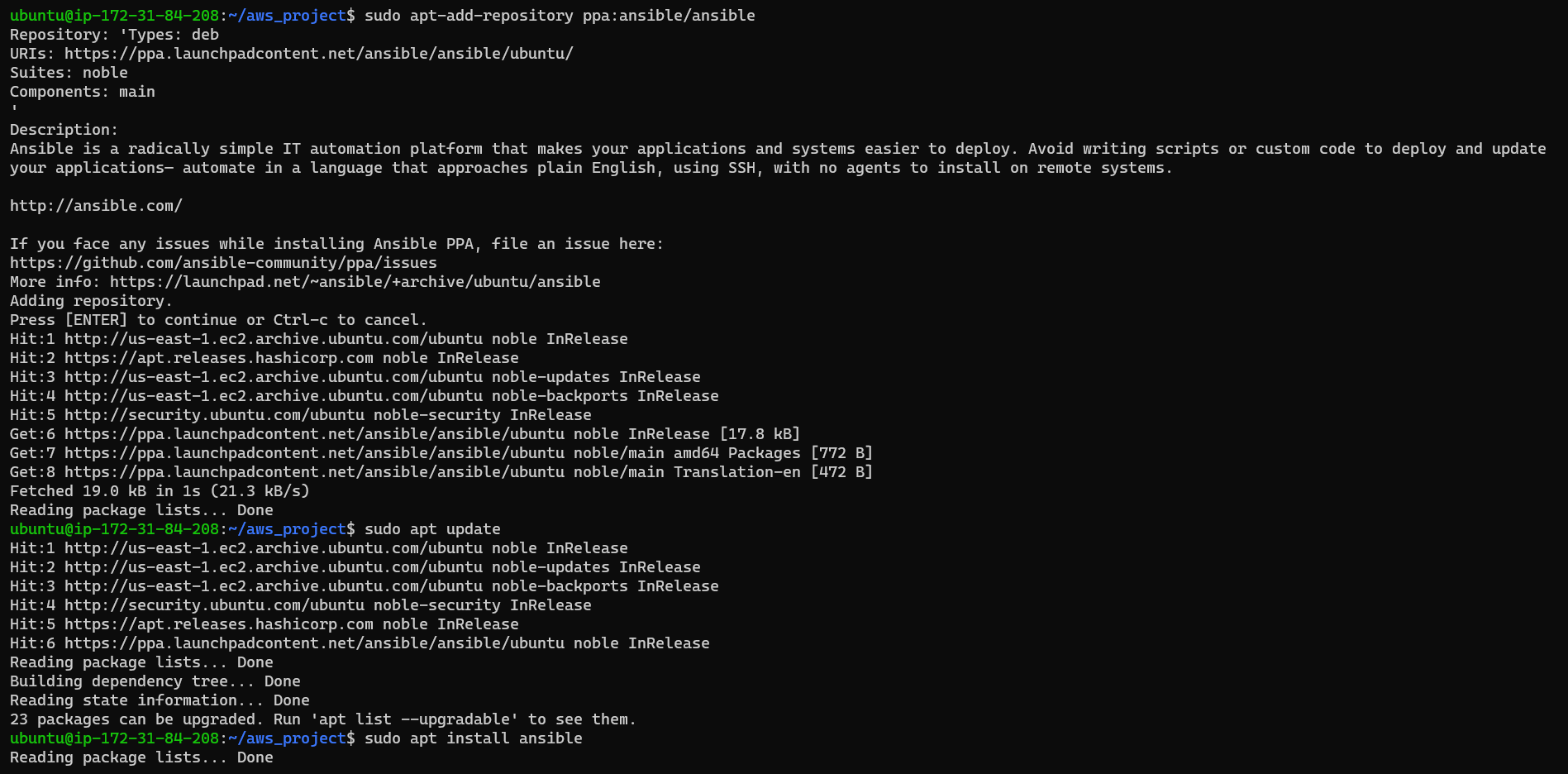
Install Ansible on our Bastion Machine –

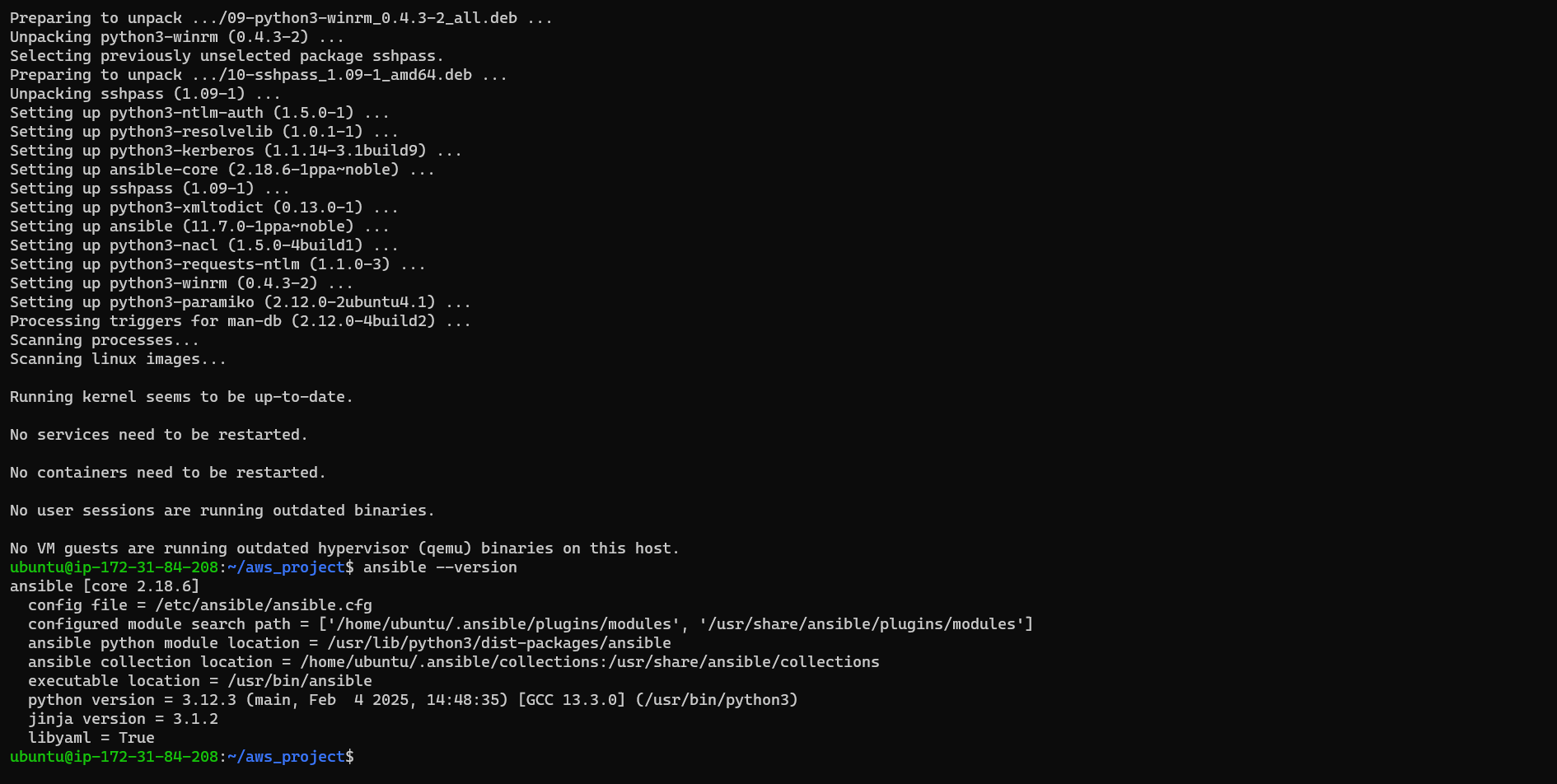
1) sudo apt-add-repository ppa:ansible/ansible

2) sudo apt update

3) sudo apt install ansible

4) ansible –version

*  Install Ansible



📁 Ansible Directory Structure:

└── ansible/

├── inventory

├── install\_docker.yaml

**📄 inventory**

[jenkins]

172.31.89.9 ansible\_user=ubuntu

[app]

172.31.94.106 ansible\_user=ubuntu

[all:vars]

ansible\_ssh\_private\_key\_file=/home/ubuntu/aws\_project/ansible/awsprojectkey.pem

**📄 install\_docker.yaml**

---

- name: Install Docker on Jenkins and App hosts

  hosts: jenkins, app

  become: true

  tasks:

    - name: Install Docker

      apt:

        name: docker.io

        state: present

        update\_cache: true

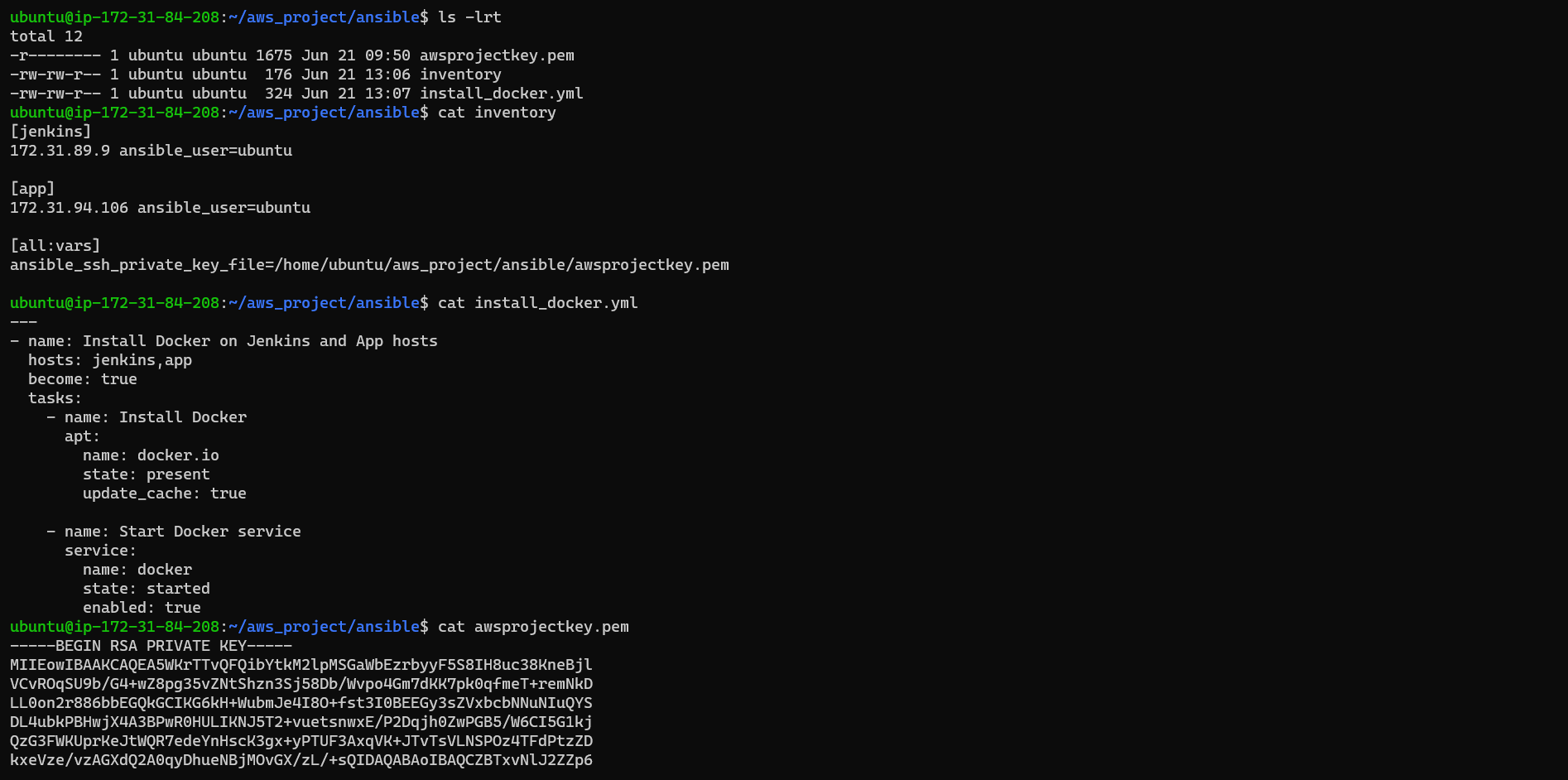
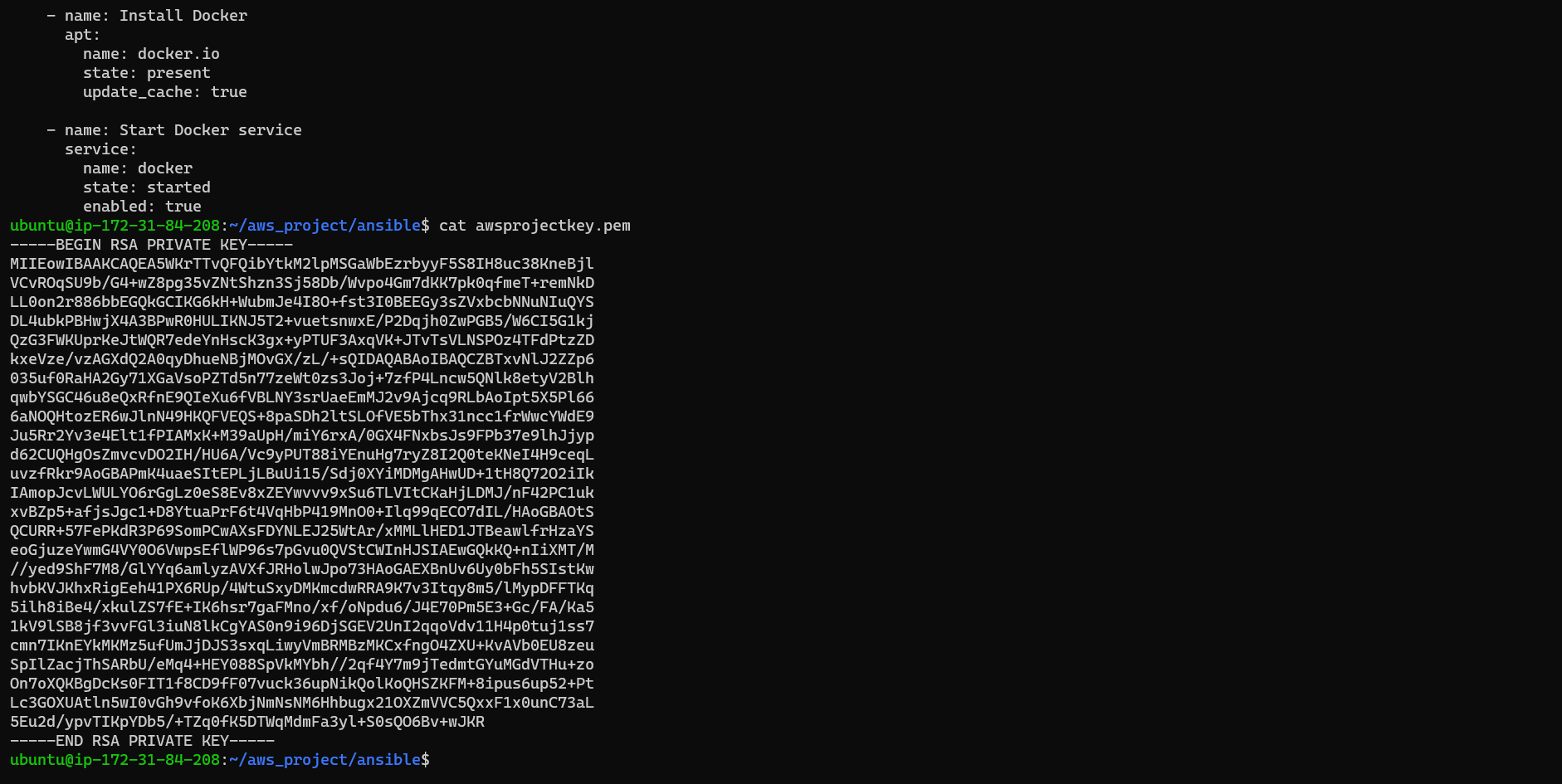
    - name: Start Docker service

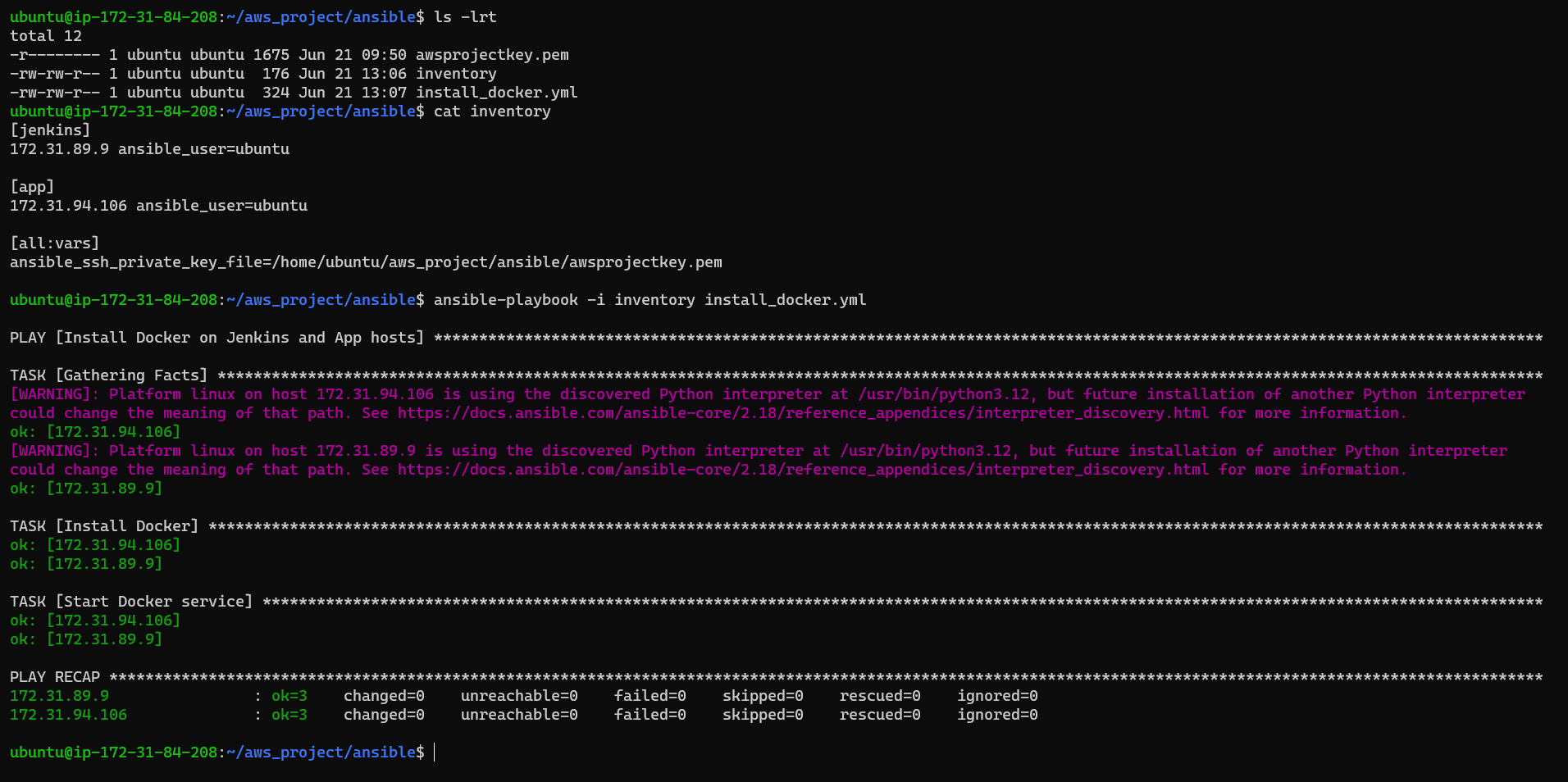
      service:

        name: docker

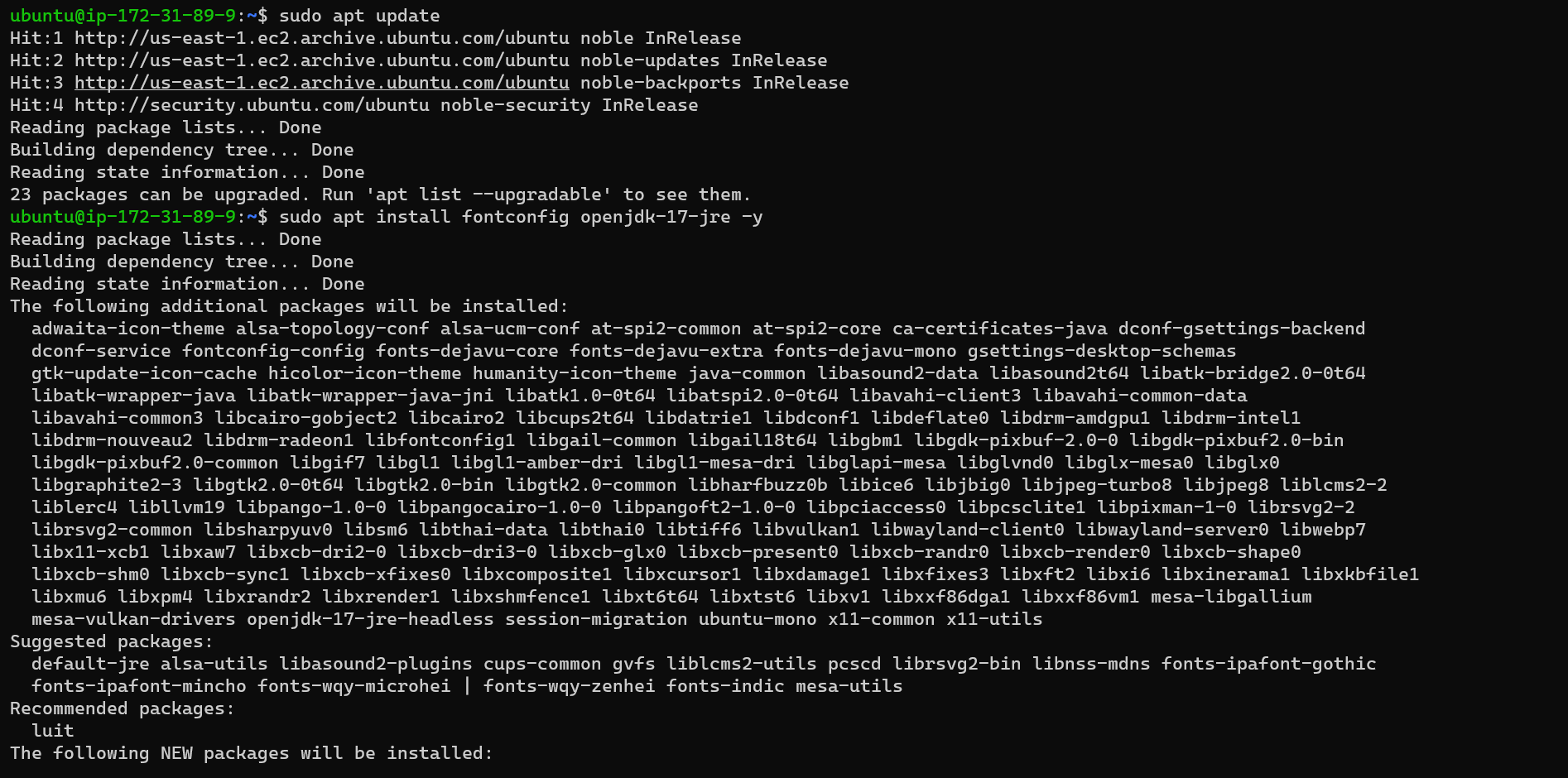
        state: started

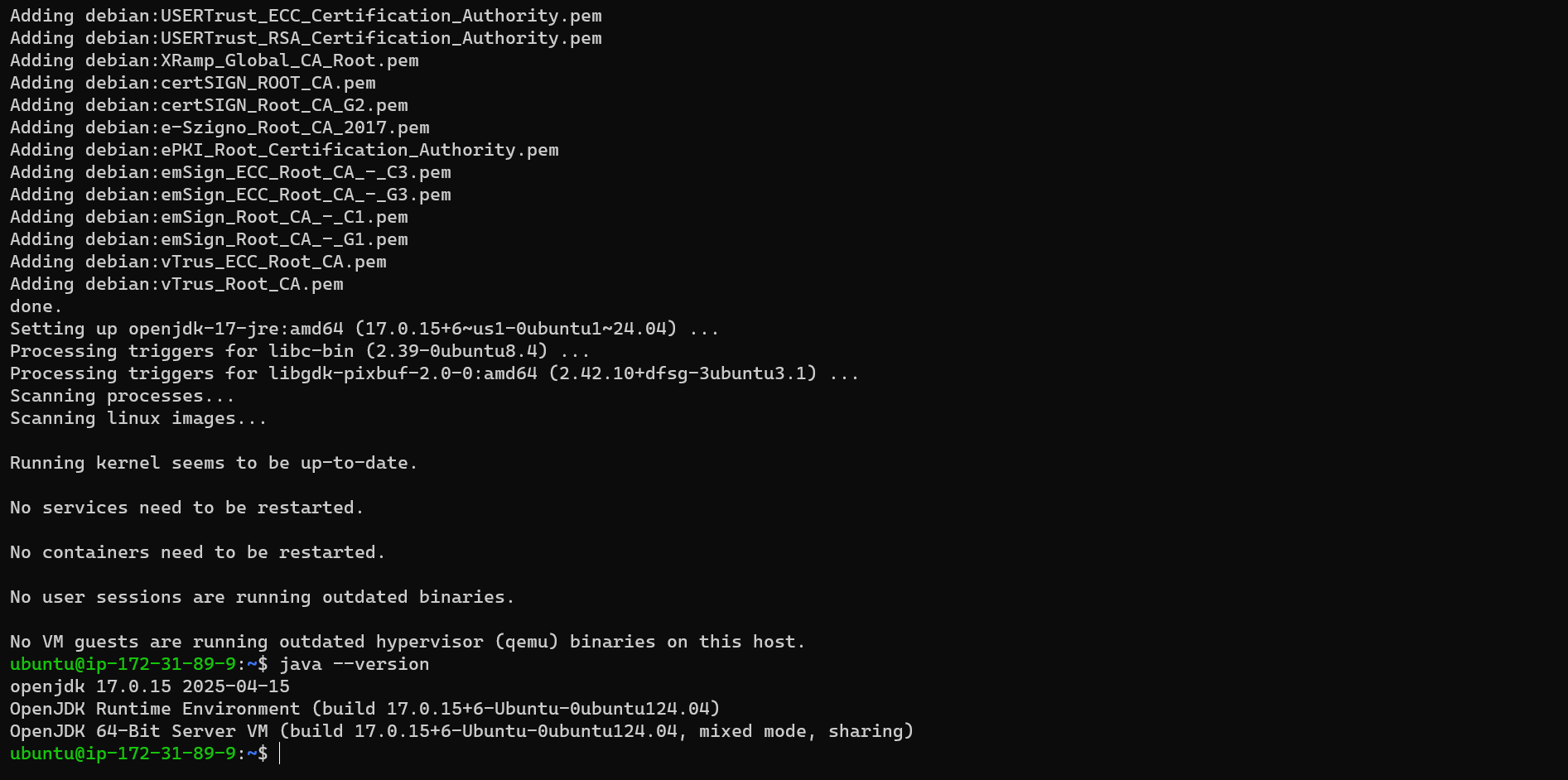
        enabled: true

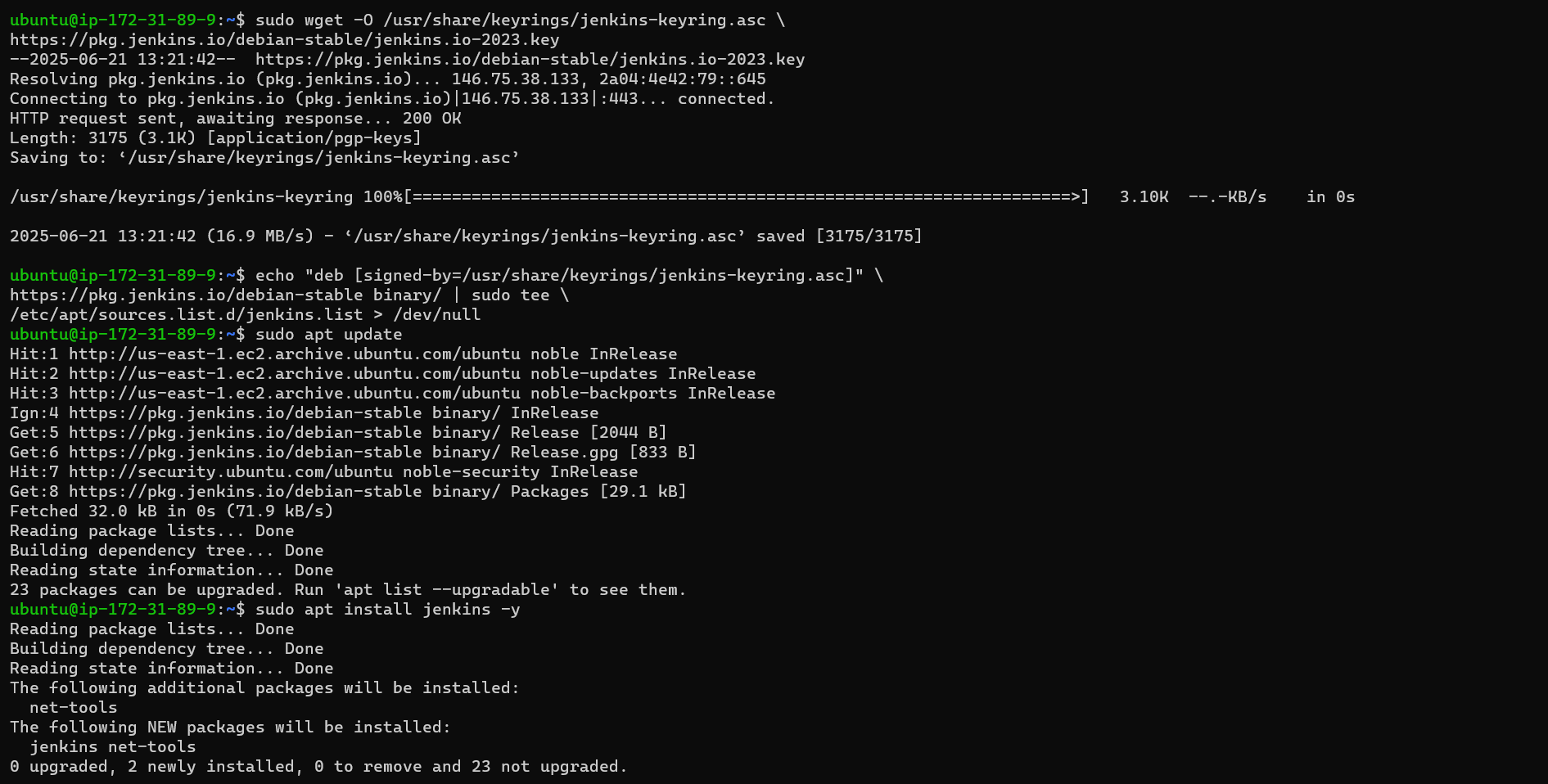
* Ansible Files
* 📦 Run Playbook :

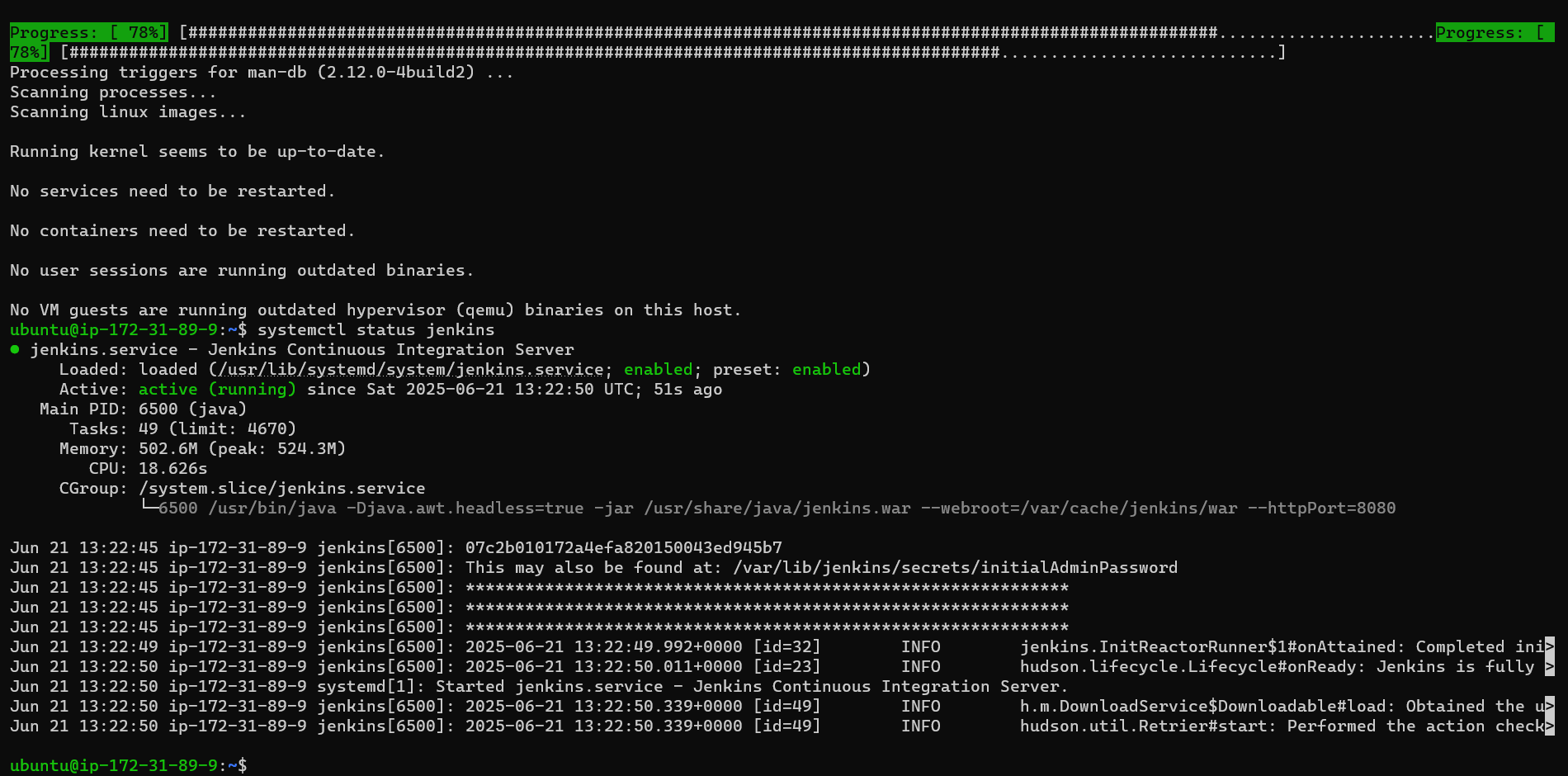
ansible-playbook -i inventory install\_docker.yml

**Subtask 2: Install and Configure Jenkins**

* ****Install Java



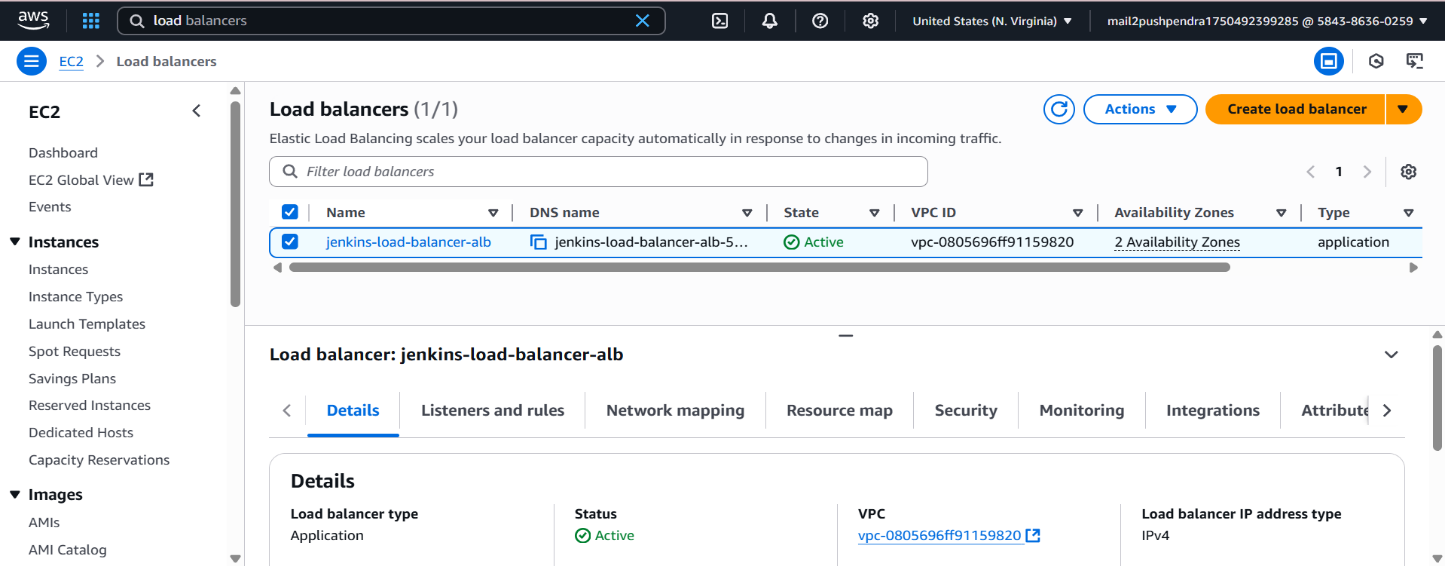
* Install Jenkins

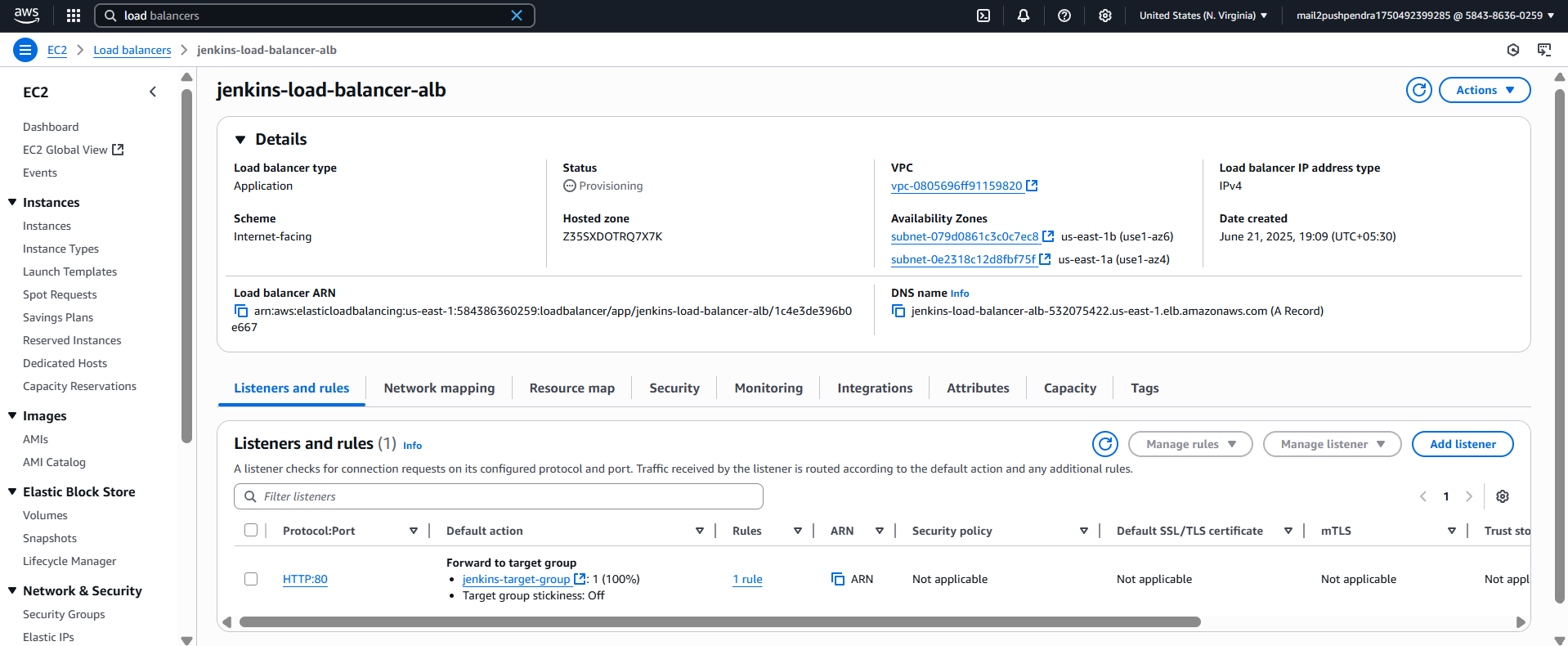


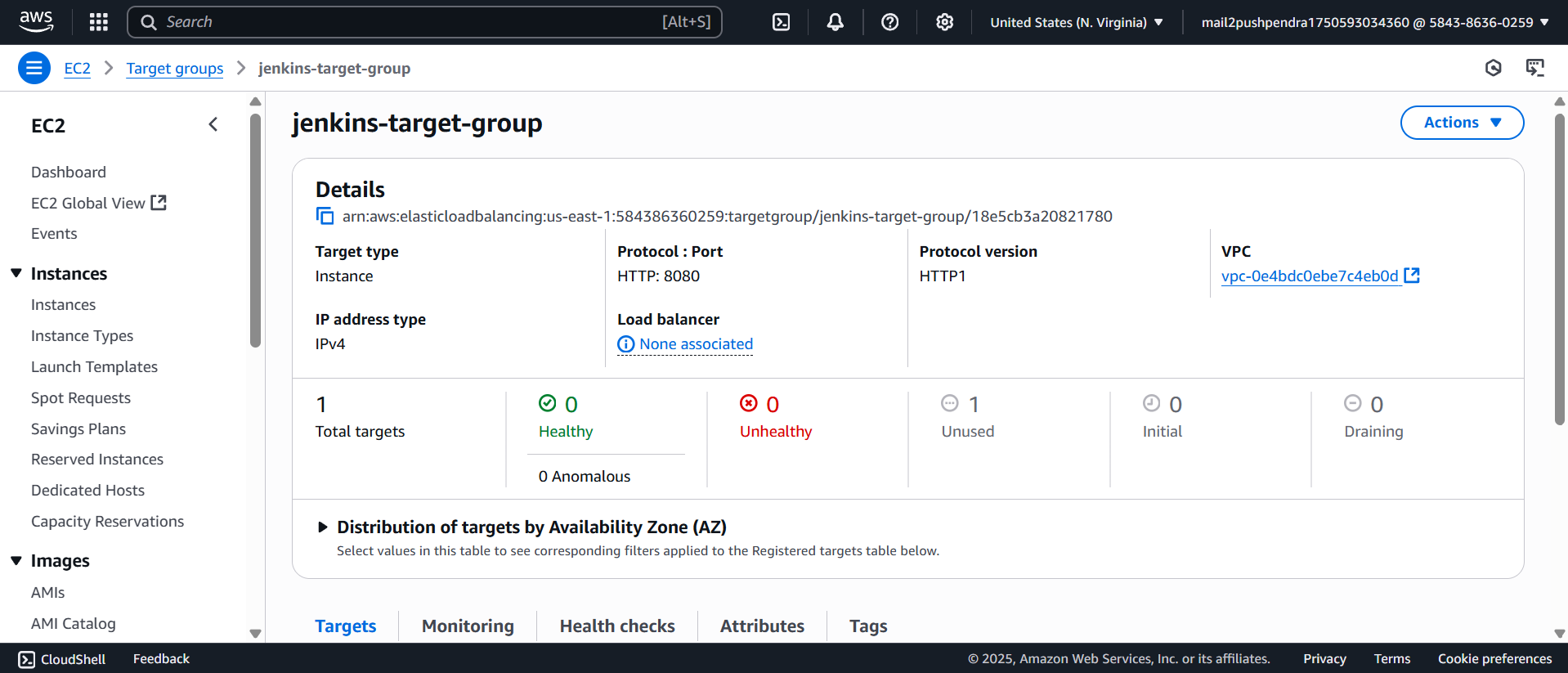
**Subtask 3: Create ALB for Jenkins**

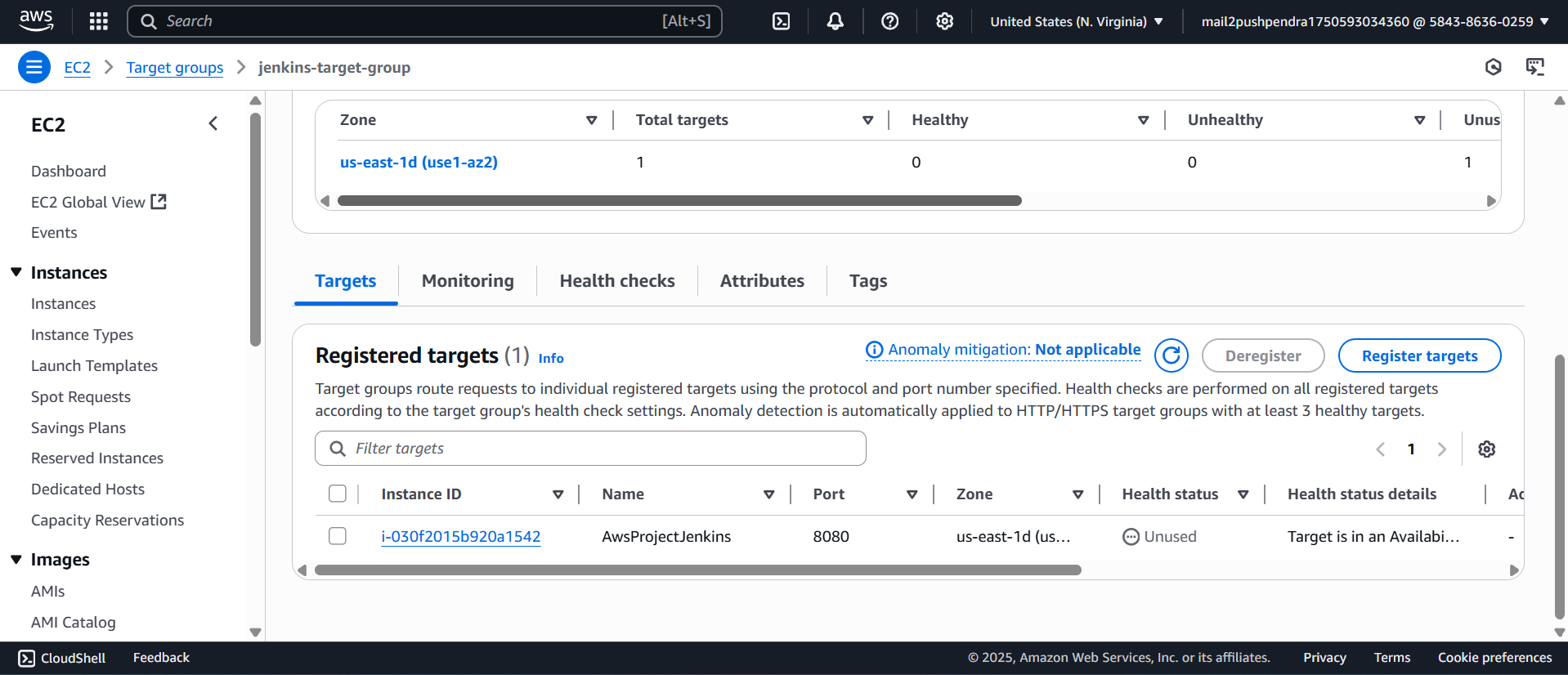
**Objective:**

Set up an internet-facing Application Load Balancer (ALB) to route traffic to the Jenkins and Application servers securely.

* Created Load Balancer ALB

****

* ****Targets Group for ALB

****

**Subtask 4: ECR, IAM Role, and Jenkins SSH to App Host**

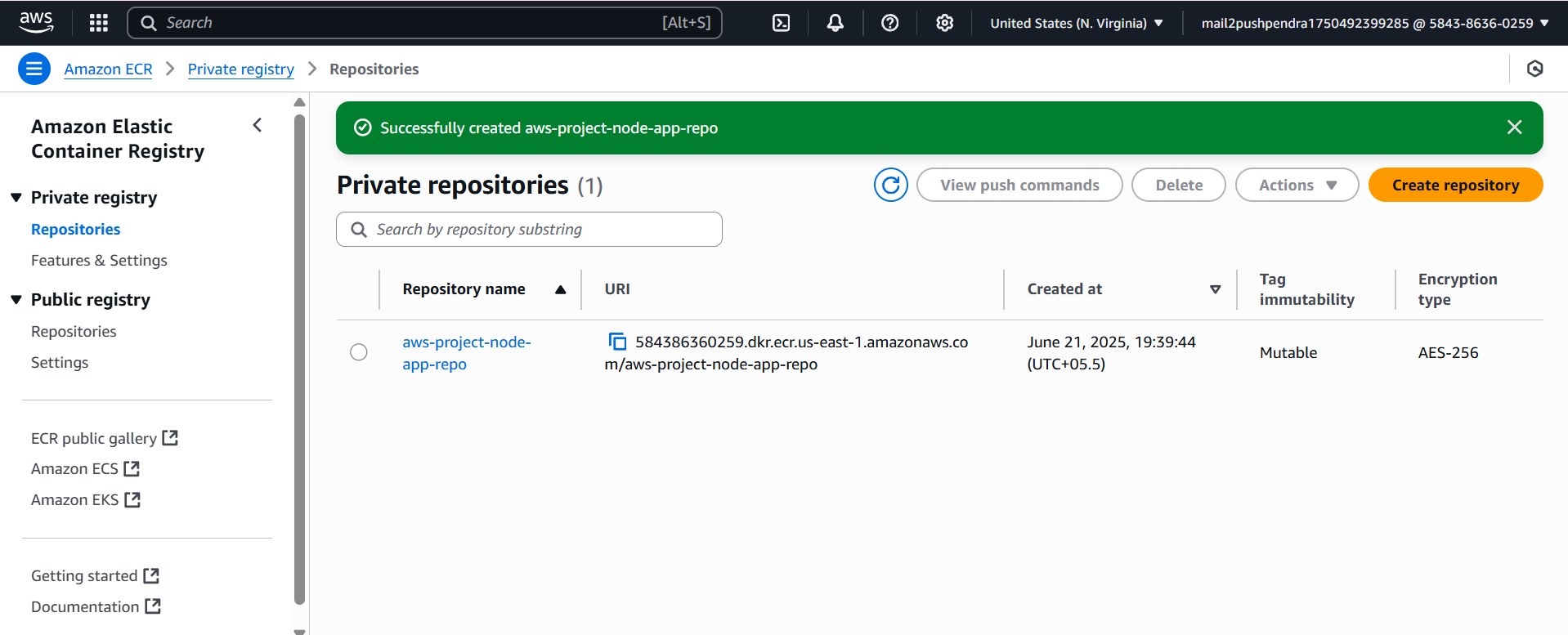
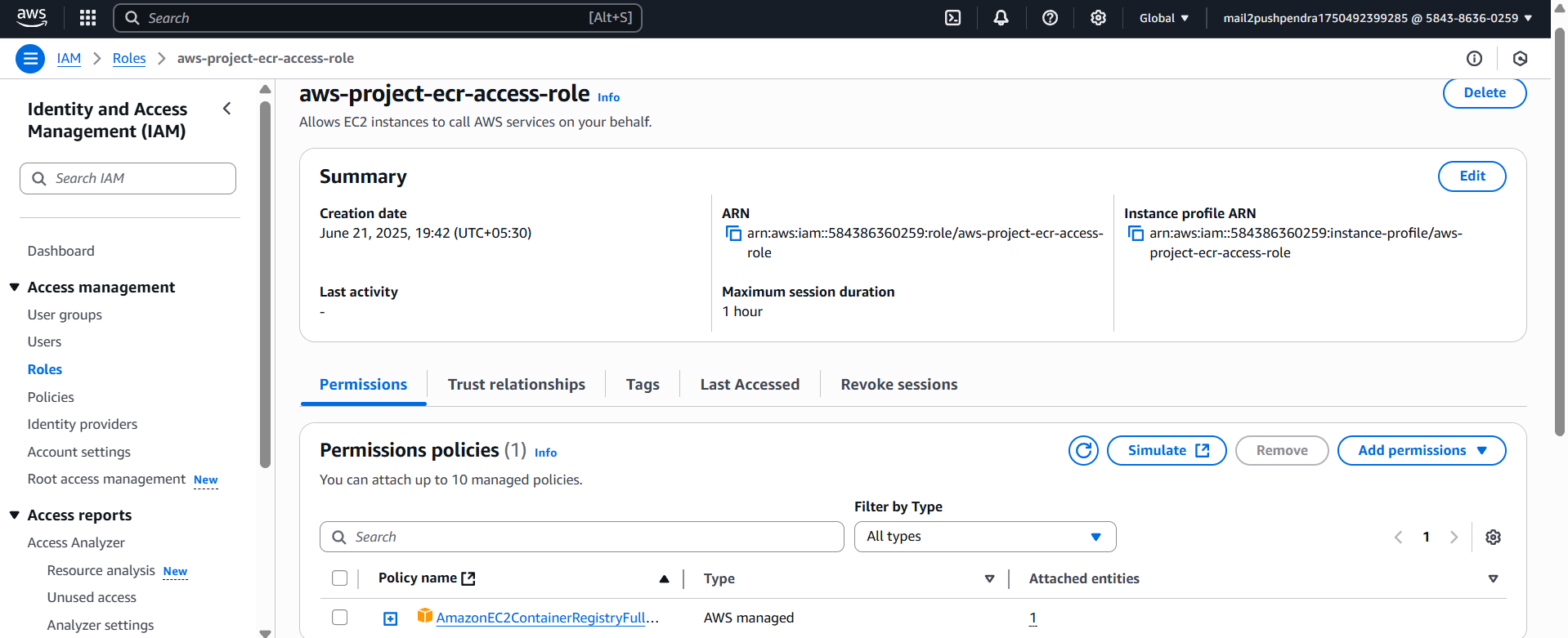
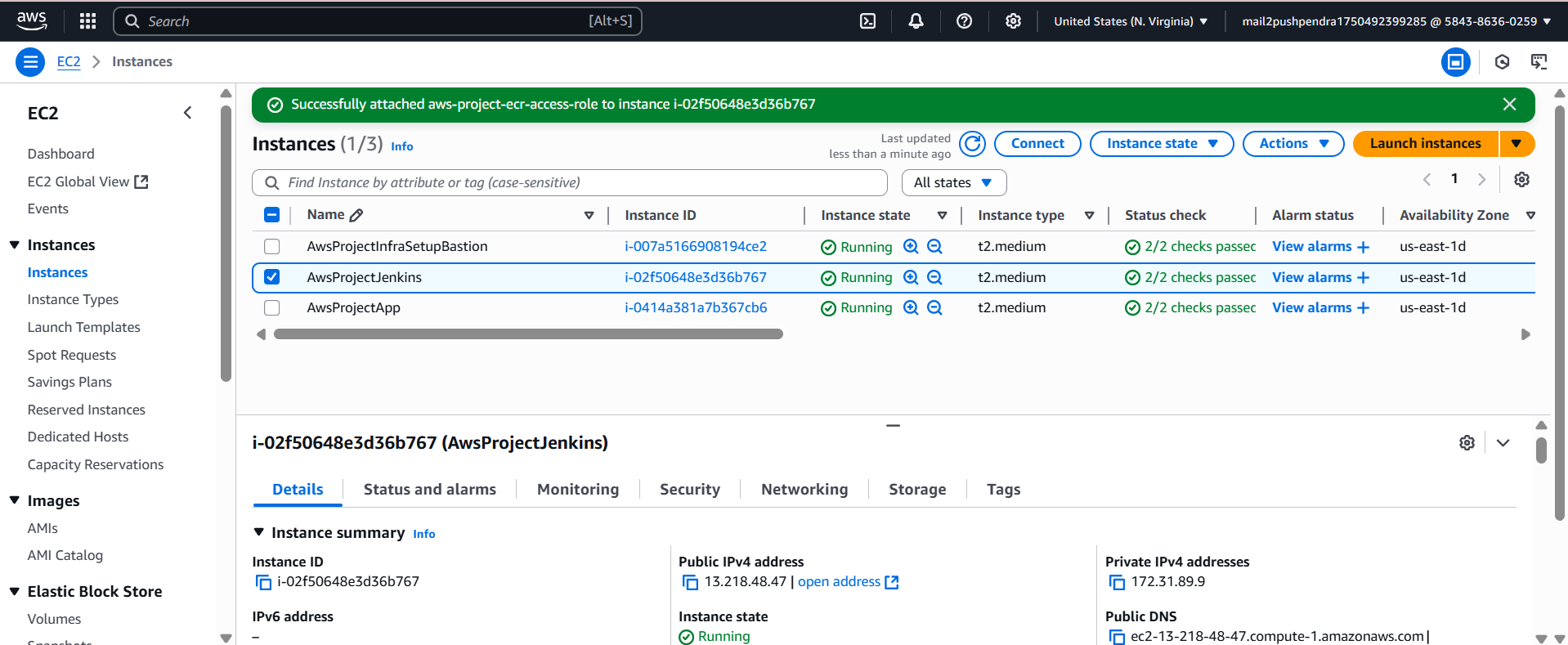
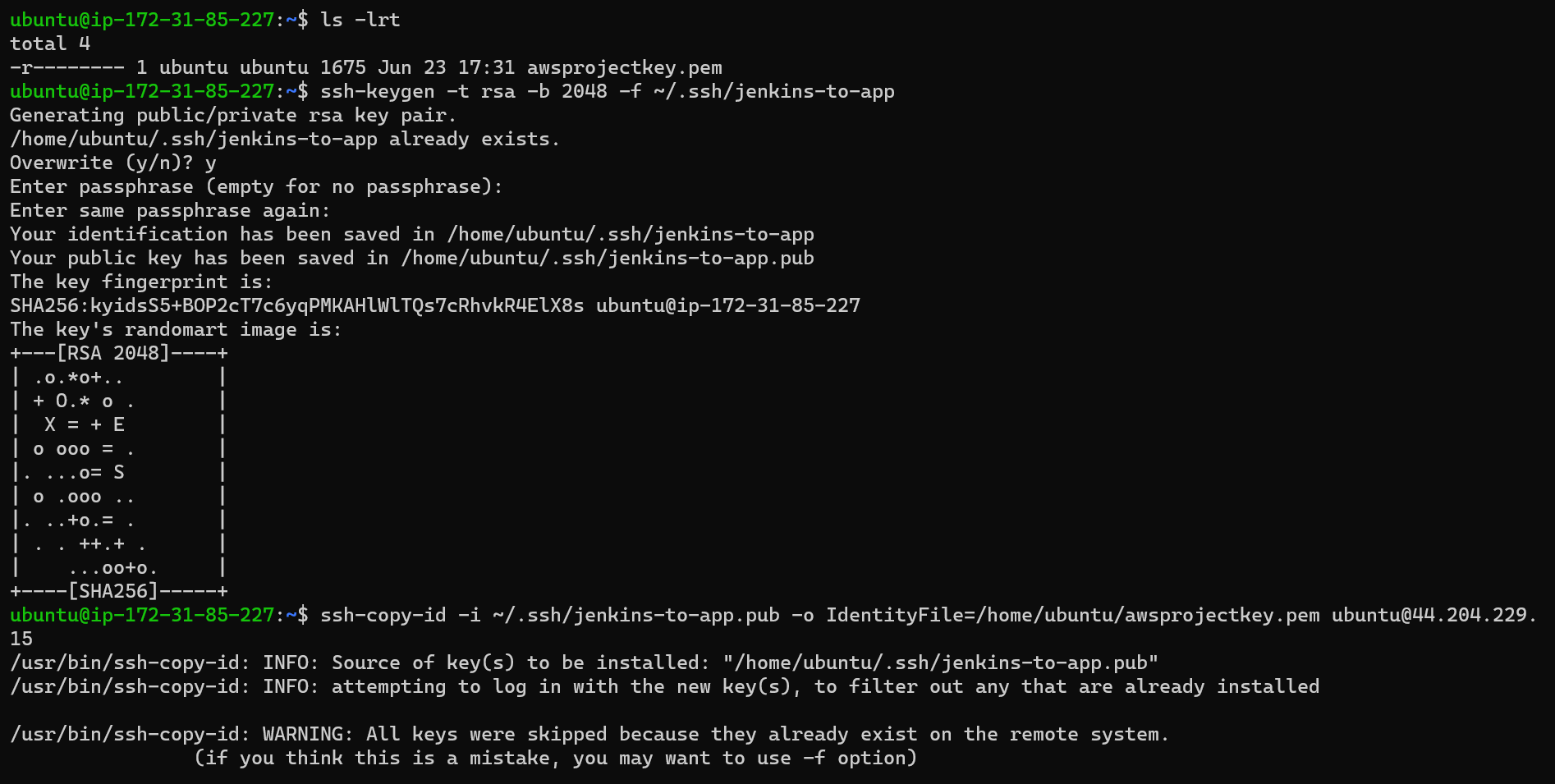
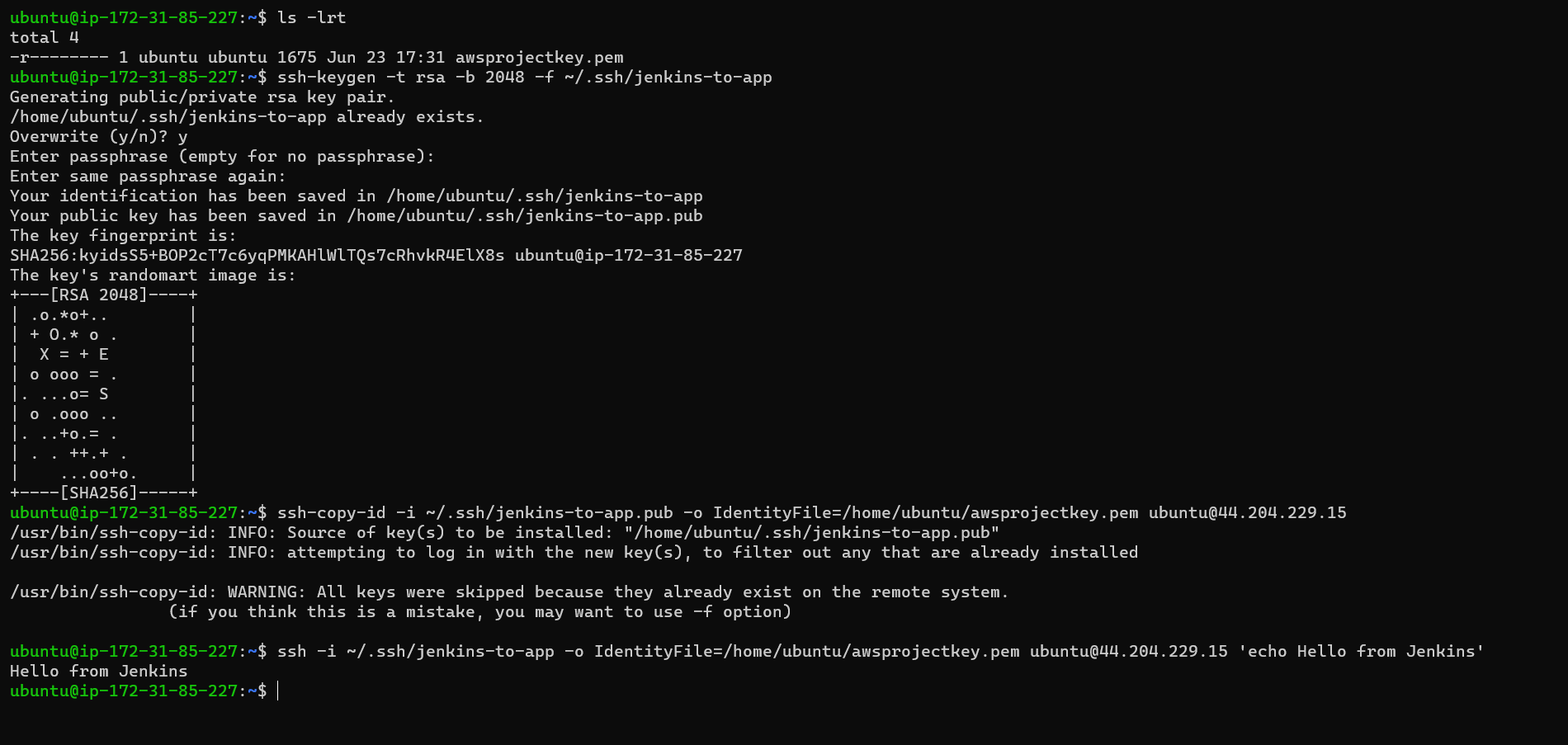
**Objective:**

Create an Amazon ECR repository for Docker images, configure IAM roles for Jenkins and Application servers, and establish SSH connectivity between Jenkins and the Application server.

**🔧 Steps: Create ECR Repo**

* Create an Amazon ECR Repository: Use the AWS Management Console or AWS CLI to create a private ECR repository named - “aws-project-node-app-repo”.
* Note the repository URI for future reference.

1. Go to **Amazon ECR > Create Repository**
2. Name: aws-project-node-app-repo
3. Visibility: **Private**
4. Leave defaults and **Create**

* Create ECR Repo
* Create IAM Role for ECR Access
* Create IAM Role for ECR Access
* Attach this role to Jenkins and App EC2 instances
* Generate SSH Key for Jenkins ↔ App
* Verify Jenkins Can SSH to App : ssh -i ~/.ssh/jenkins-to-app ubuntu@<APP-HOST-IP> 'echo Hello from Jenkins'

**Subtask Bonus: Create ALB and TGs via Terraform**

**📄 alb\_target\_groups.tf**

resource "aws\_lb" "jenkins\_alb" {

name = "jenkins-alb"

internal = false

load\_balancer\_type = "application"

subnets = [aws\_subnet.public[0].id, aws\_subnet.public[1].id]

security\_groups = [aws\_security\_group.web\_sg.id]

}

resource "aws\_lb\_target\_group" "jenkins\_tg" {

name = "jenkins-tg"

port = 8080

protocol = "HTTP"

vpc\_id = aws\_vpc.main.id

}

resource "aws\_lb\_listener" "http" {

load\_balancer\_arn = aws\_lb.jenkins\_alb.arn

port = 80

protocol = "HTTP"

default\_action {

type = "forward"

target\_group\_arn = aws\_lb\_target\_group.jenkins\_tg.arn

}

}

resource "aws\_lb\_listener\_rule" "jenkins\_rule" {

listener\_arn = aws\_lb\_listener.http.arn

action {

type = "forward"

target\_group\_arn = aws\_lb\_target\_group.jenkins\_tg.arn

}

condition {

path\_pattern {

values = ["/jenkins\*", "/jenkins"]

}

}

}

**Task 3: Dockerize sample Node.js app & write Jenkinsfile**

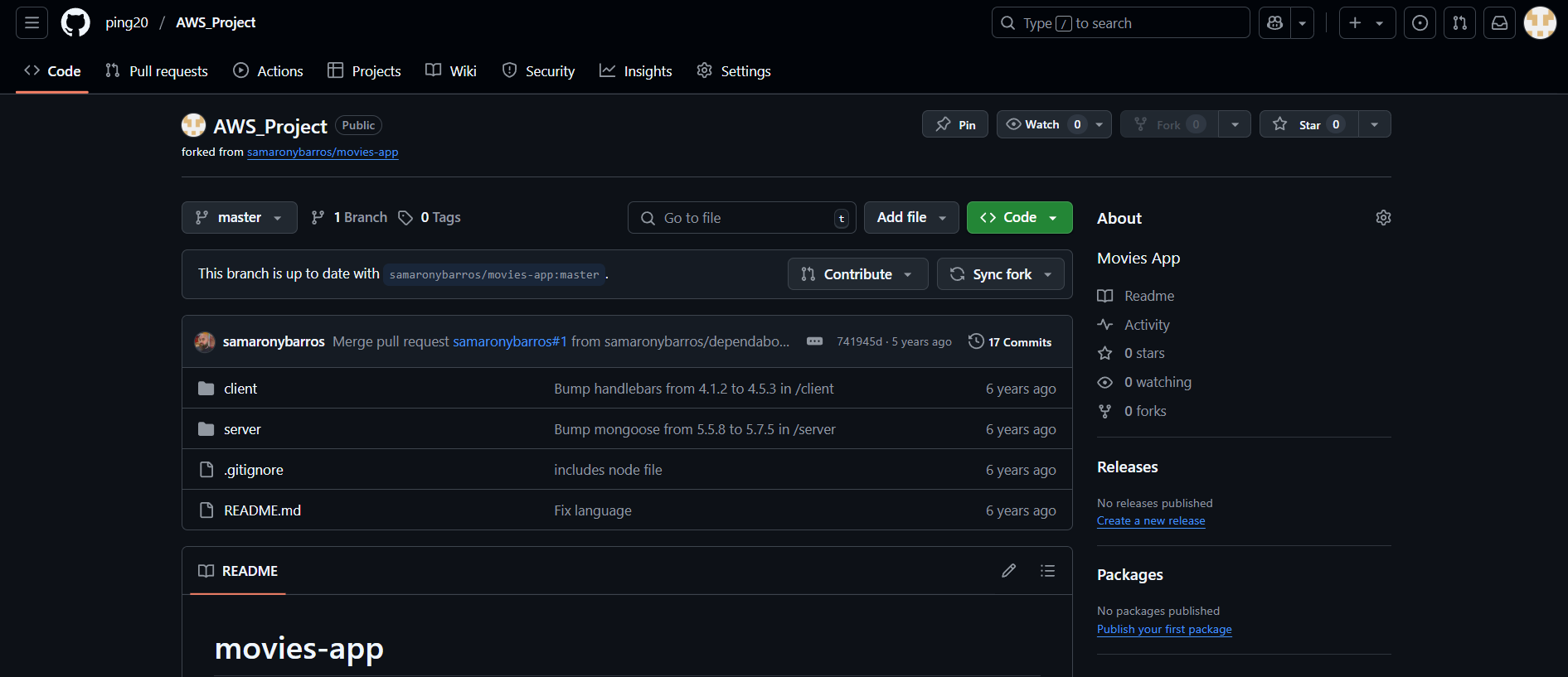
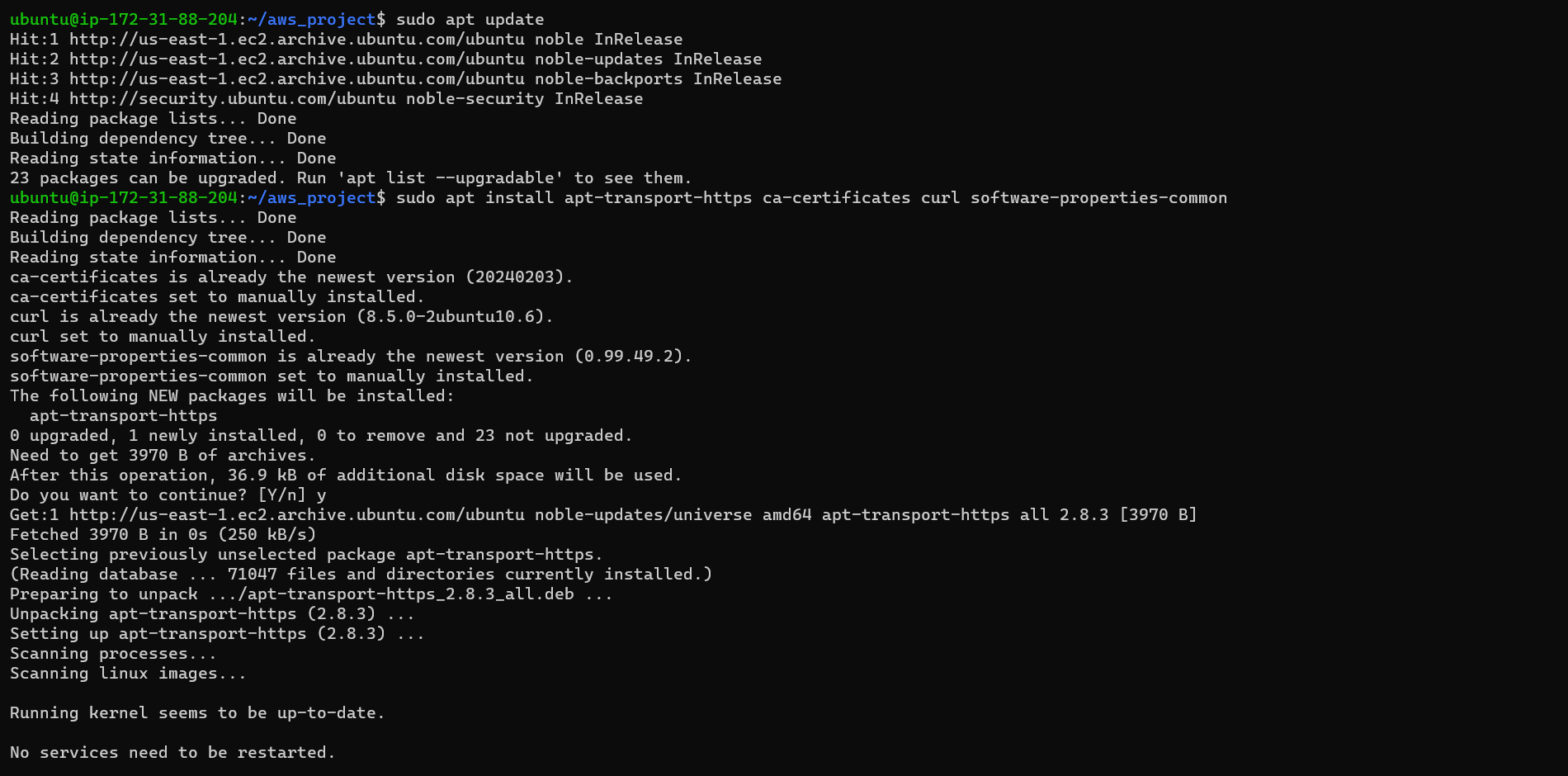
**Objective:**

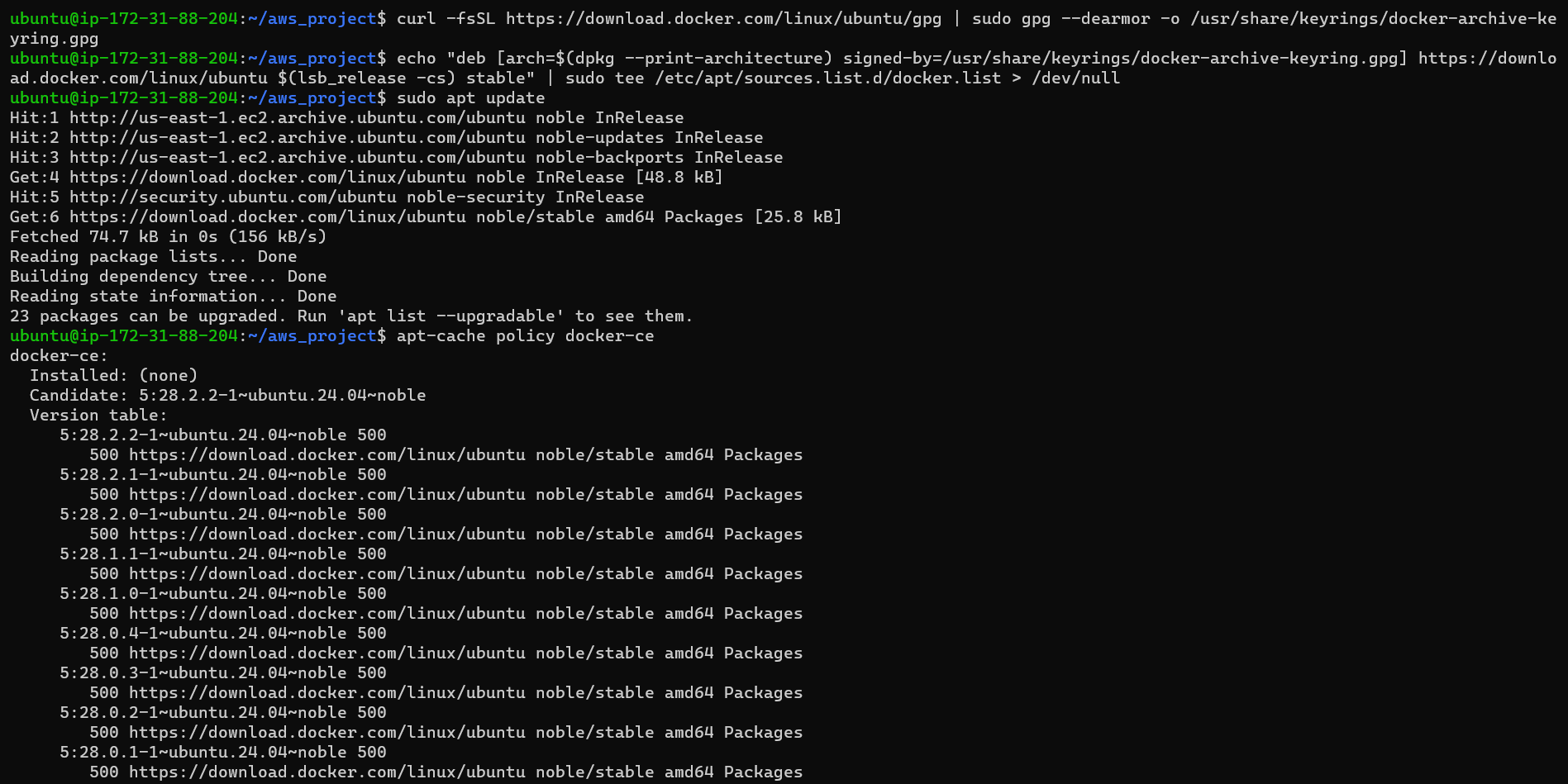
Dockerize the movies-app Node.js application, test it locally, and push the code along with the Dockerfile to a private GitHub repository.

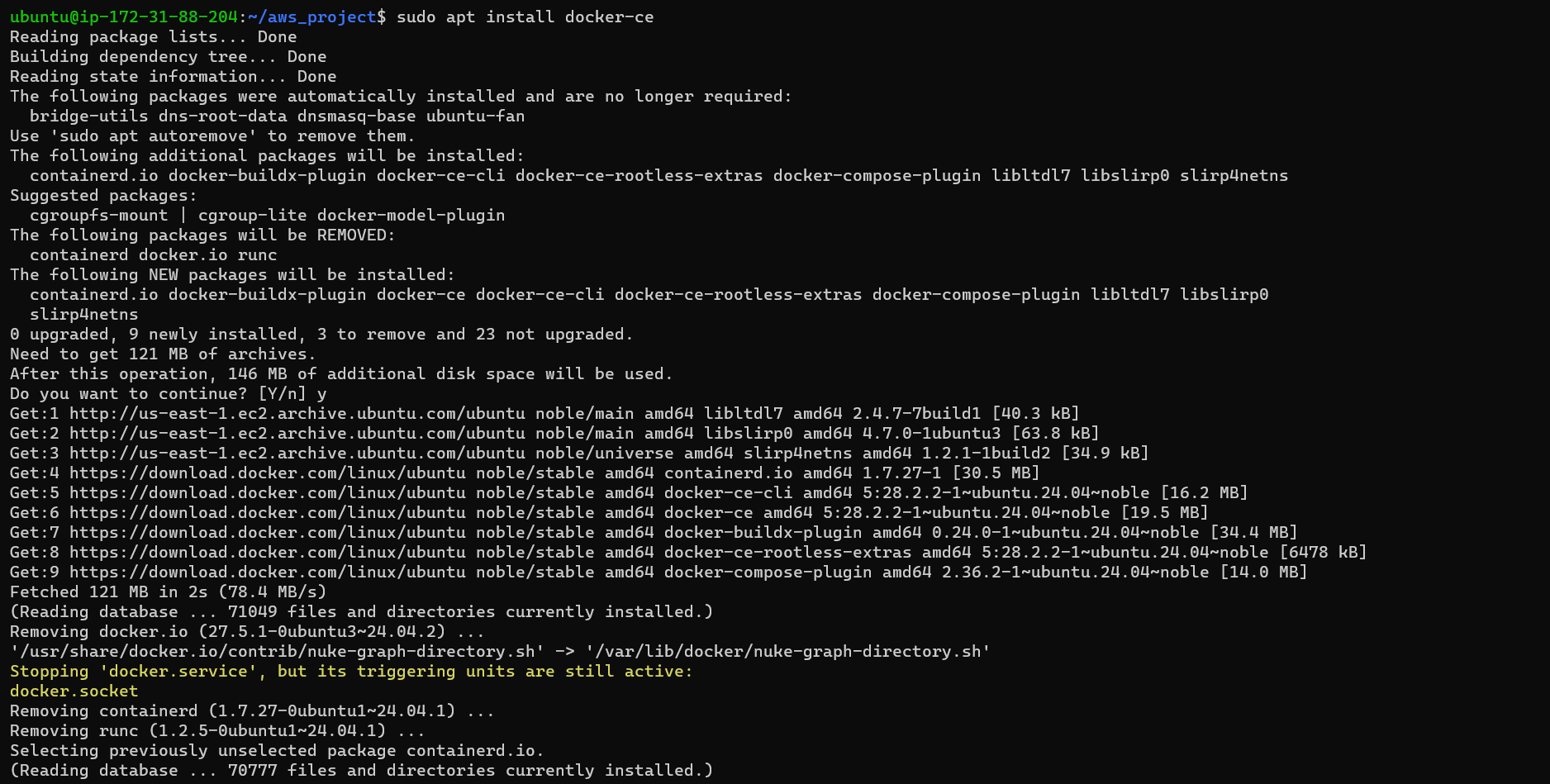
**Subtask 1: Create Repo & Dockerize the App**

Clone and Upload Sample App to GitHub

1. Visit: <https://github.com/samaronybarros/movies-app>
2. Click Code > Download ZIP or git clone -: Extract or clone to your local system.
3. Create a new private GitHub repository named: AWS\_Project
4. Copy the backend code into your new repo (ignore frontend):

* Clone and Upload Sample App to GitHub
* Install Docker

****



**🐳 Create Dockerfile**

**📄 Dockerfile**

# Use alpine node image

FROM node:18-alpine

# Set environment variables

ENV HOST=0.0.0.0

ENV PORT=3000

ENV NODE\_OPTIONS=--openssl-legacy-provider

# Set work directory

WORKDIR /app

# Install bash (good for scripts)

RUN apk add --no-cache bash

# Copy package.json and install dependencies

COPY package\*.json ./

RUN npm install

# Copy app source code

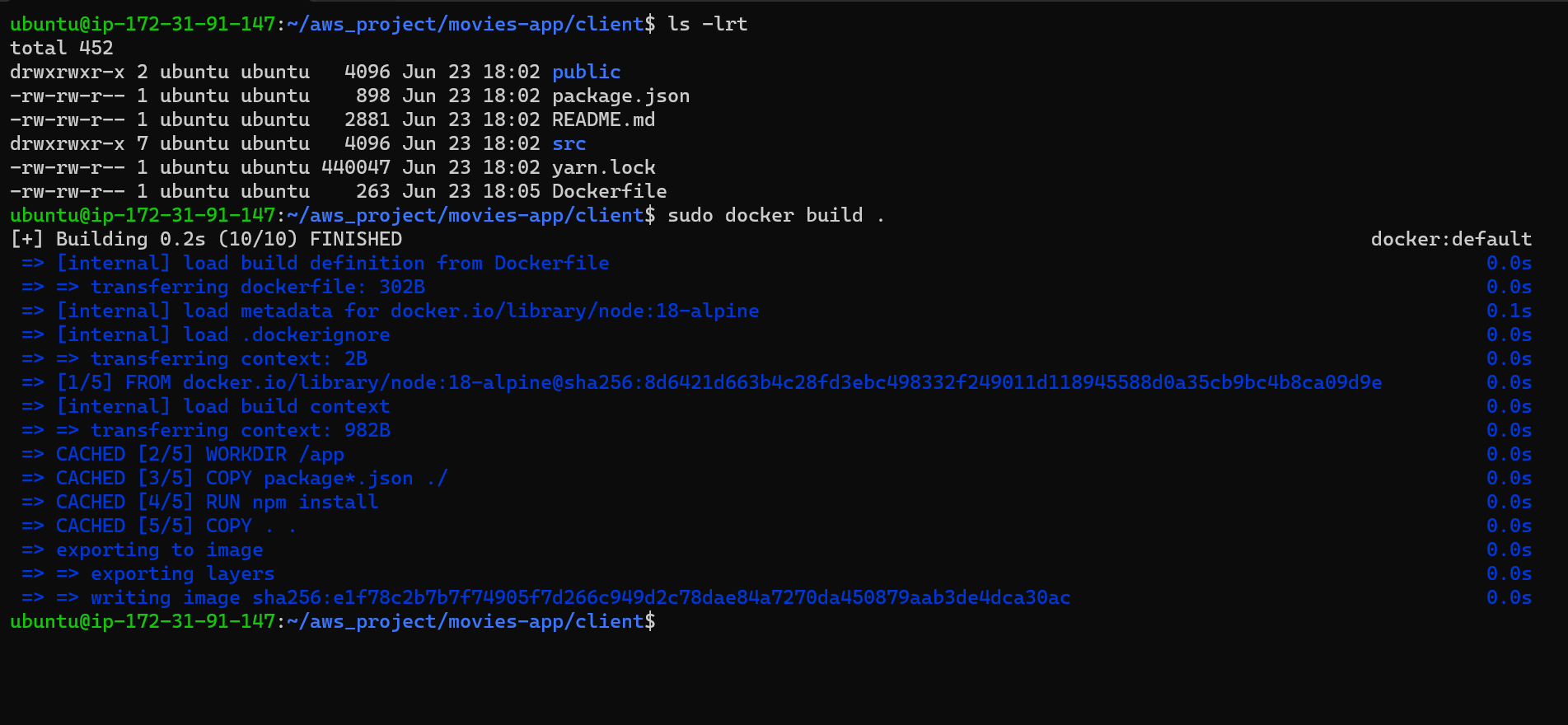
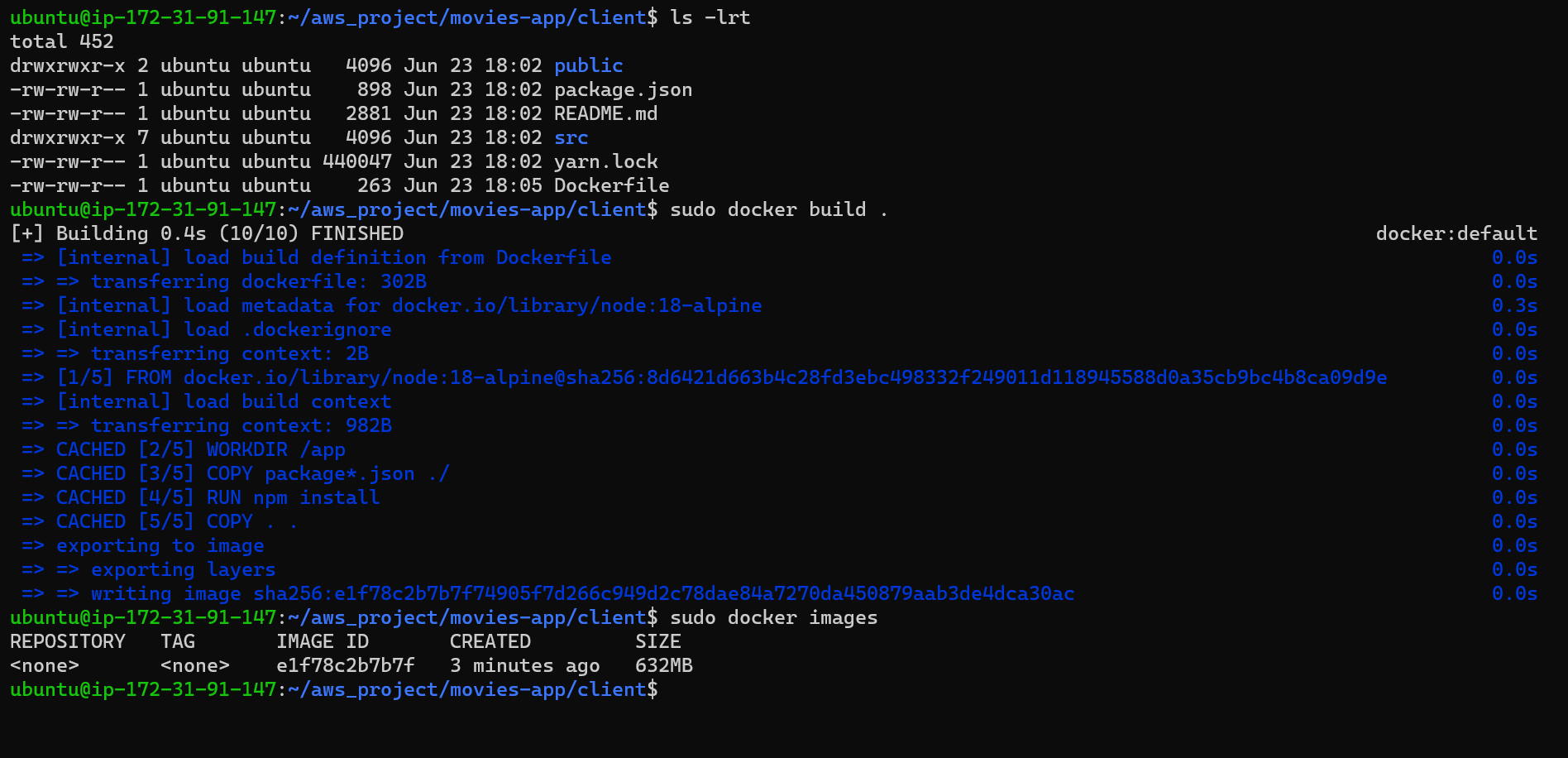
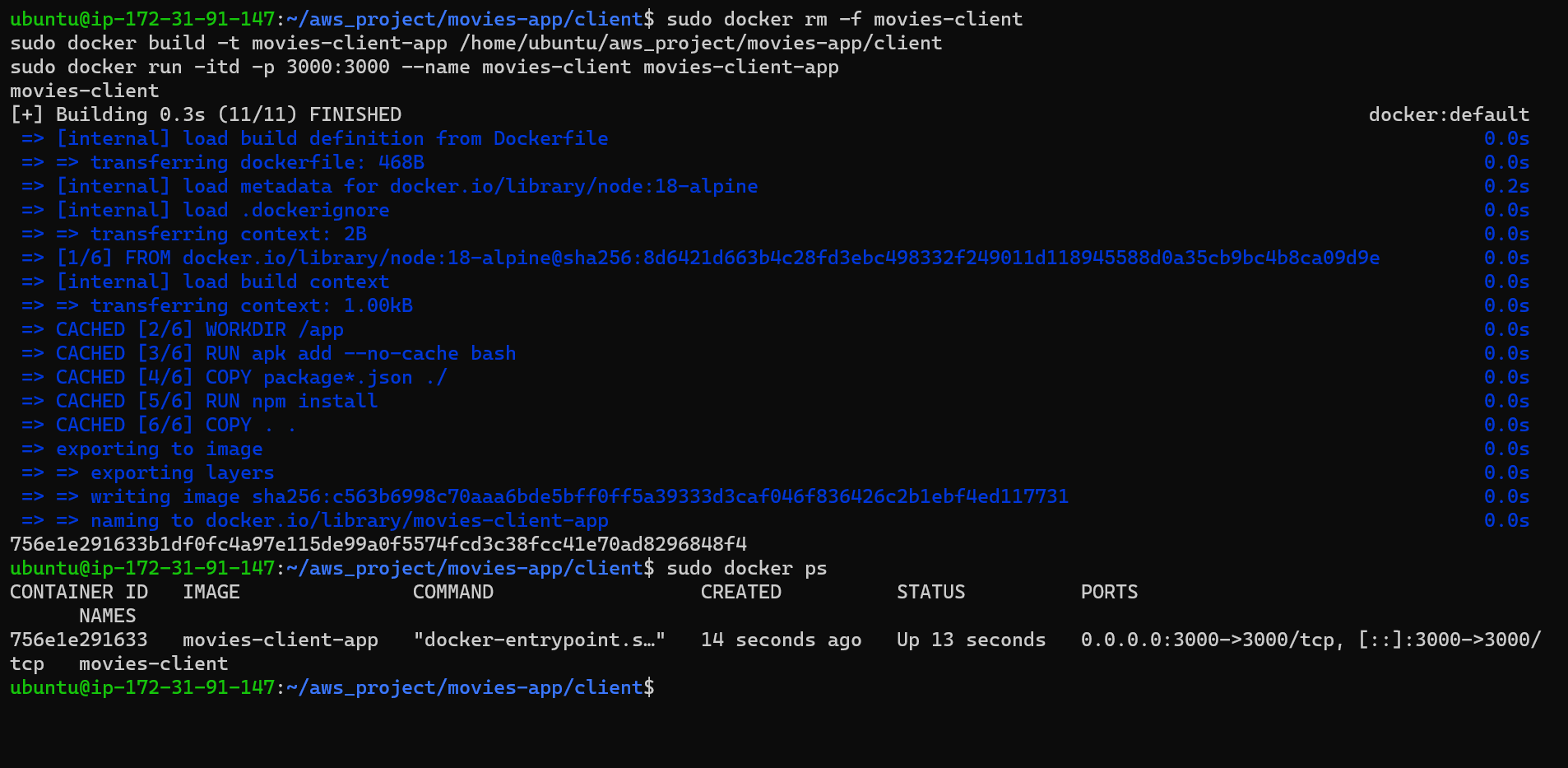
COPY . .

# Expose the port

EXPOSE 3000

# Run the app

CMD ["npm", "start"]

* Test Docker Build Locally
* ****Docker Image
* Running Docker Container
* Once working, push Dockerfile to GitHub:

#bash

* Add the remote repository
  + git add Dockerfile
* Stage and commit the changes
  + git commit -m "Added Dockerfile"
* Push to GitHub
  + git push

**Subtask 2: Create Jenkinsfile and Setup CI Pipeline**

**Objective:**

Develop a Jenkins pipeline (Jenkinsfile) that automates the process of checking out the code from GitHub, building the Docker image, and pushing it to Amazon ECR.

**📄 Jenkinsfile**

pipeline {

  agent any

  environment {

    REPO\_NAME = 'node-app-repo'

    IMAGE\_TAG = "${BUILD\_NUMBER}"

    AWS\_REGION = 'us-east-1'

    ECR\_URI = "<your\_aws\_account\_id>.dkr.ecr.${AWS\_REGION}.amazonaws.com/${REPO\_NAME}"

  }

  stages {

    stage('Checkout Code') {

      steps {

        git branch: 'master', url: 'git@github.com:<your-username>/node-docker-app.git'

      }}

    stage('Build & Push Docker Image') {

      steps {

        sh '''

        aws ecr get-login-password --region $AWS\_REGION | docker login --username AWS --password-stdin $ECR\_URI

        docker build -t $ECR\_URI:$IMAGE\_TAG .

        docker push $ECR\_URI:$IMAGE\_TAG

        '''

      }}

    stage('Deploy to App Host') {

      steps {

        sh '''

        ssh -o StrictHostKeyChecking=no -i ~/.ssh/jenkins-to-app ubuntu@<APP-PRIVATE-IP> << EOF

          docker ps -q --filter ancestor=$ECR\_URI:$IMAGE\_TAG | grep . && docker stop \$(docker ps -q --filter ancestor=$ECR\_URI:$IMAGE\_TAG)

          docker pull $ECR\_URI:$IMAGE\_TAG

          docker run -d -p 3000:3000 $ECR\_URI:$IMAGE\_TAG

        EOF

        '''

      }

}

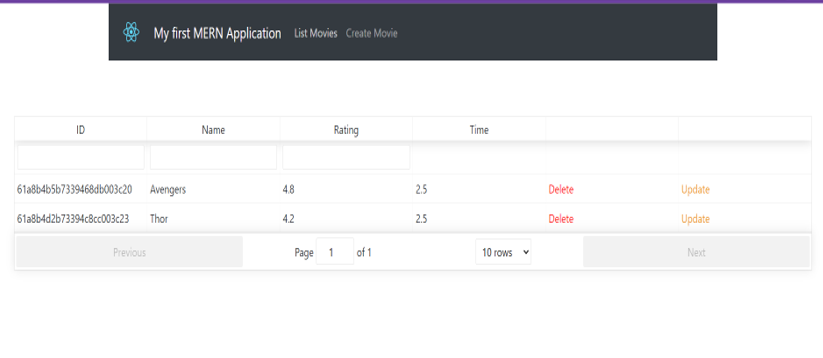
}

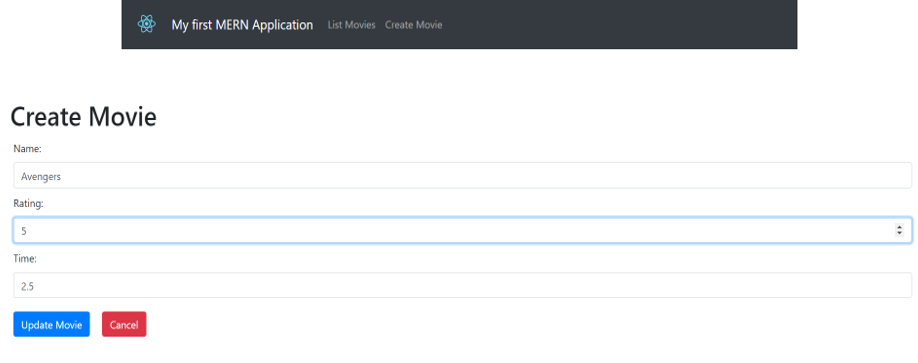
}

**🛠️ Create Jenkins Job**

1. Go to **Jenkins Dashboard**
2. Click **New Item > Pipeline**
3. Name it: docker-ci-pipeline
4. Select **Pipeline > OK**
5. Under **Pipeline script from SCM**:
   * SCM: Git
   * Repo: git@github.com:ping20/AWS\_Project.git
   * Credentials: Add SSH key (or use HTTPS + GitHub token)
6. Script Path: Jenkinsfile
7. Click **Save**
8. Click **Build Now**

**Subtask 3: Deploy to App Host from Jenkins**

* Deploy to App Host



**Subtask Bonus: Estimate Cost**

Use AWS Pricing Calculator: <https://calculator.aws.amazon.com/>

| **Resource** | **Type** | **Qty** | **Approx Cost (USD/month)** |
| --- | --- | --- | --- |
| EC2 (t2.micro) | Jenkins | 1 | ~$8 |
| EC2 (t2.micro) | App | 1 | ~$8 |
| EC2 (t2.micro) | Bastion | 1 | ~$8 |
| ALB | 1 ALB + 1 TG | 1 | ~$20 |
| NAT Gateway | 1 (AZ-a only) | 1 | ~$30–35 |
| S3 (Terraform State) | 1 bucket | 1 | ~$0.50 |
| ECR | Storage only | ~1 GB | ~$0.10 |

**🔢 Total (monthly): ~$75–80/month**

🔻 Reduce cost by:

* Stopping EC2 instances when idle
* Deleting NAT GW, ALB during idle periods
* Using Terraform destroy and recreate when needed

**Conclusion**

By completing the above tasks and subtasks, I have established a solid foundation for deploying a containerized Node.js application using Jenkins and AWS services.

The key achievements include:

1. Development Environment Setup Development Environment Setup:

- Installed necessary tools and dependencies.

- Verified the application runs correctly in the local environment.

2. AWS Infrastructure Setup AWS Infrastructure Setup:

- Configured ALB and Target Groups to manage application traffic.

- Set up ECR repositories and IAM roles for secure Docker image management.

3. Application Containerization Application Containerization:

- Successfully Dockerized the Movies App.

- Tested Docker builds locally to ensure application integrity.

4. CI/CD Pipeline Implementation CI/CD Pipeline Implementation:

- Developed a Jenkins pipeline that automates the build and push processes.

- Addressed and resolved common pipeline errors to ensure smooth operations.