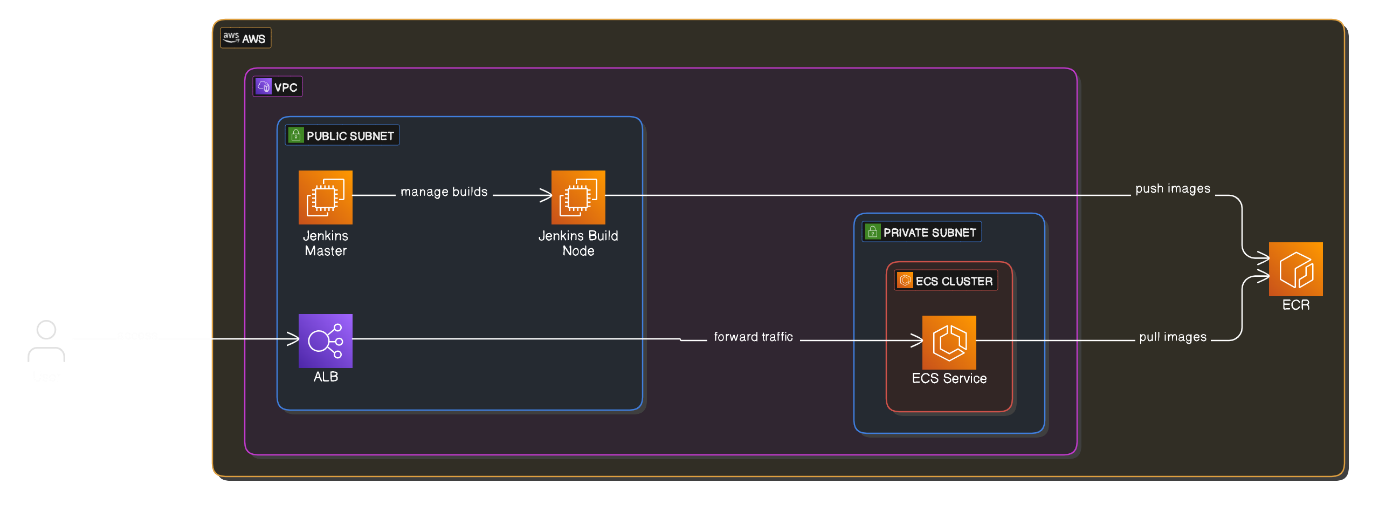
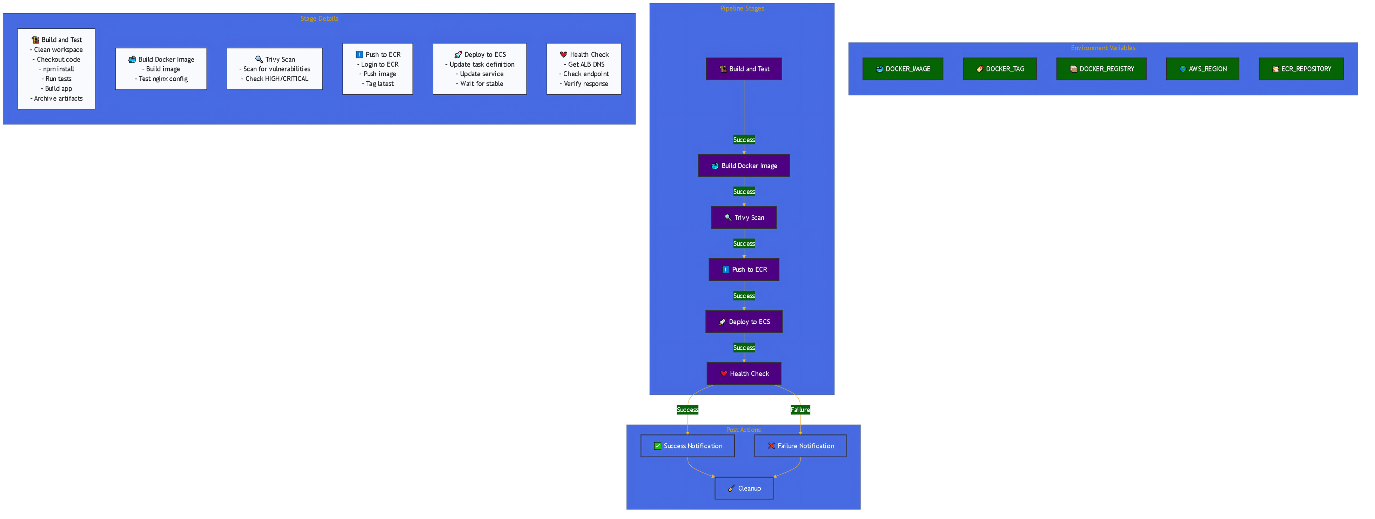
**Jenkins CI/CD Pipeline for AWS ECS Deployment**





**1. Create AWS Key Pair**

aws ec2 create-key-pair --key-name jenkins-node-key --query 'KeyMaterial' --output text > jenkins-node-key.pem

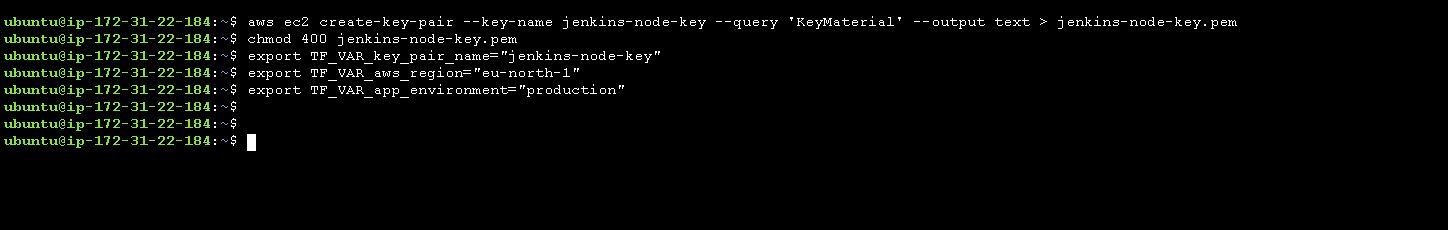
  chmod 400 jenkins-node-key.pem

**2. Set Required Environment Variables**

export TF\_VAR\_key\_pair\_name="jenkins-node-key"

export TF\_VAR\_aws\_region="eu-north-1"

export TF\_VAR\_app\_environment="production"



**3. Creating IAM Role: MySessionManagerrole**

**Step 1: Create the IAM Role**

1. Navigate to the **AWS Management Console**.
2. Open the **IAM (Identity and Access Management) service**.
3. Select **Roles** from the left-hand menu.
4. Click on **Create Role**.
5. For **Trusted entity type**, select **AWS Service**.
6. Choose **EC2** as the service that will use this role.
7. Click **Next: Permissions**.

**Step 2: Attach Managed Policies**

Attach the following managed policies to the role:

1. **AmazonEC2ContainerRegistryPowerUser**
   * ARN: arn:aws:iam::aws:policy/AmazonEC2ContainerRegistryPowerUser
2. **AmazonSSMManagedInstanceCore**
   * ARN: arn:aws:iam::aws:policy/AmazonSSMManagedInstanceCore
3. **AmazonEC2FullAccess**
   * ARN: arn:aws:iam::aws:policy/AmazonEC2FullAccess
4. **AmazonECS\_FullAccess**
   * ARN: arn:aws:iam::aws:policy/AmazonECS\_FullAccess

**Step 3: Configure Trust Relationship**

1. On the **Review** page, enter the **Role Name** as MySessionManagerrole.
2. Add a **Description**: "Allows EC2 instances to call AWS services on your behalf."
3. Click **Create Role**.

**Step 4: Verify the Role**

1. Go to the IAM **Roles** section.
2. Search for MySessionManagerrole.
3. Confirm that all the managed policies are attached and the trust relationship is set to allow ec2.amazonaws.com to assume the role.

**4. Clone the project**

git clone https://github.com/sandeepkalathil/Jenkins-ECS-Project.git

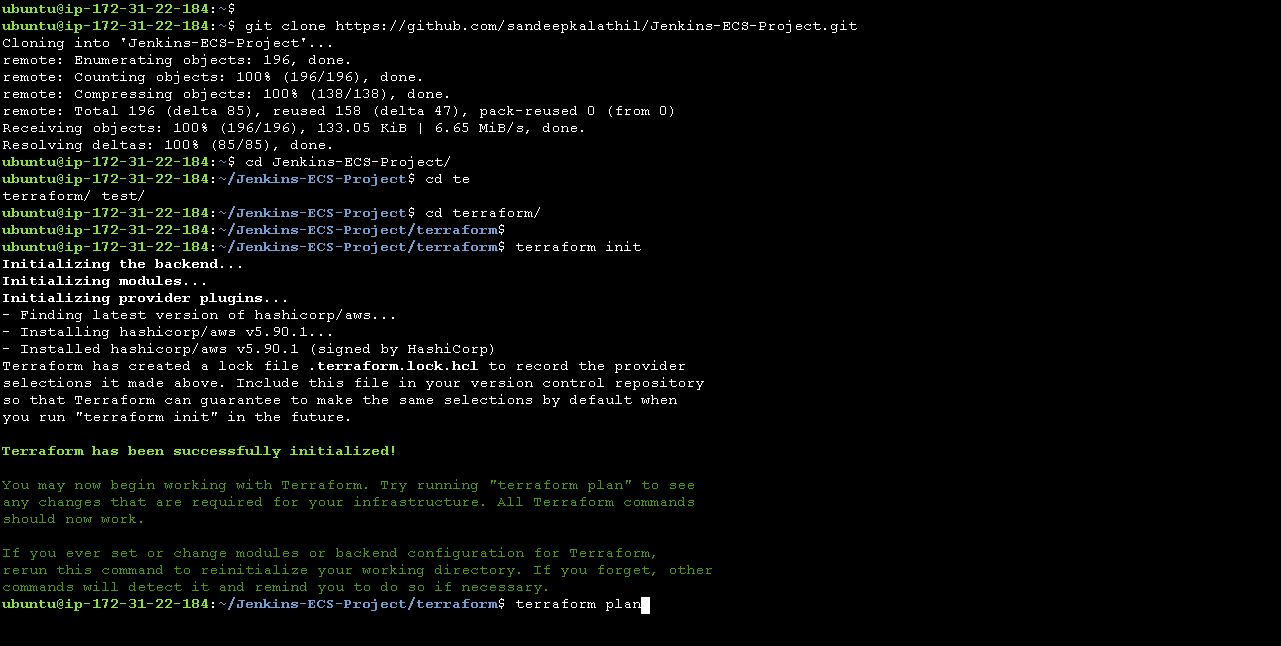
cd Jenkins-ECS-Project/

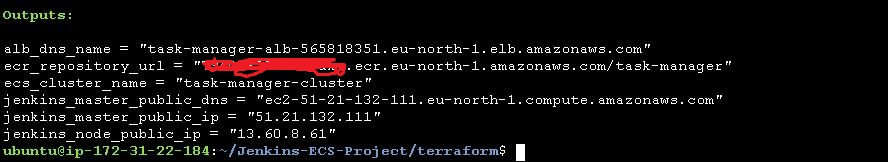
cd terraform

terraform init

terraform plan

terraform apply

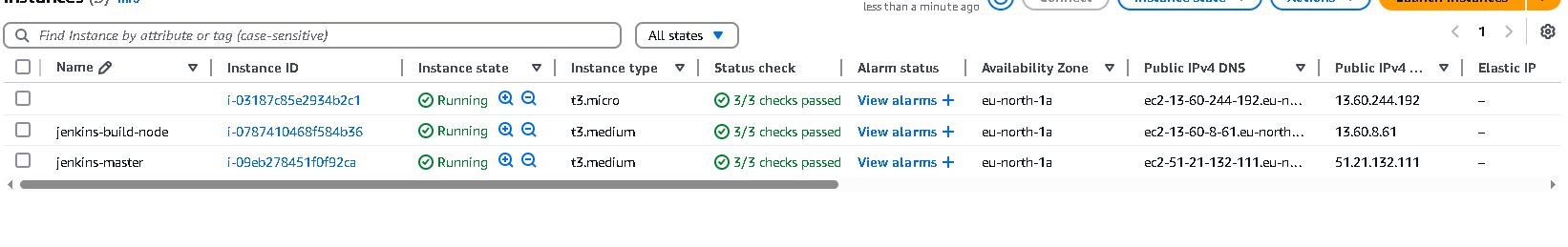


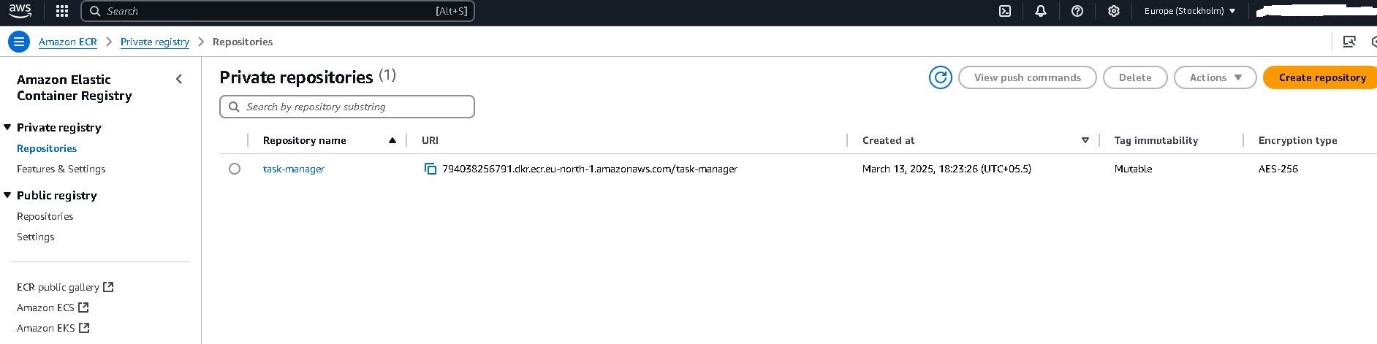


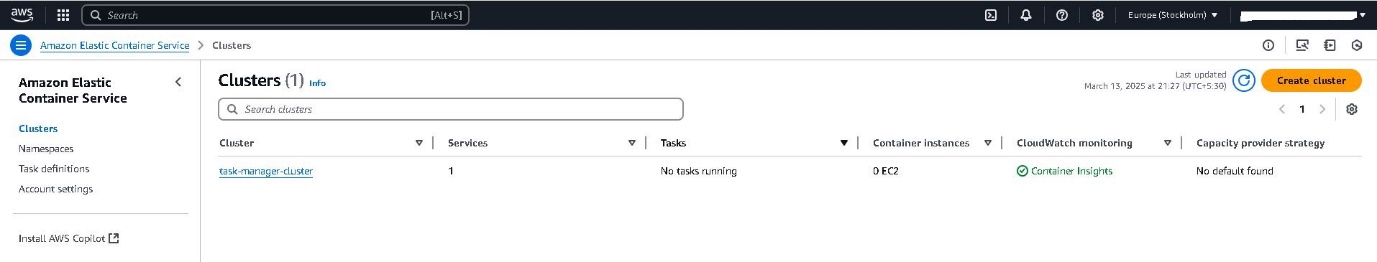
**5.Key Components of the Infrastructure**

This Terraform script deploys a Jenkins master and build node on AWS EC2, along with an ECS-based web application. Below are the key components:

1. **Networking (VPC Module)**:
   * Creates a VPC with public and private subnets.
   * Configures a single NAT gateway for outbound internet access from private subnets.
2. **Jenkins Master and Node EC2 Instances**:
   * Installs Jenkins, Docker, and AWS CLI on the master node.
   * The build node is configured with Docker and Java for running builds.
   * Security groups allow SSH and Jenkins access (though the wide-open ingress rules should be restricted for production).
3. **ECR for Container Storage**:
   * Creates an Elastic Container Registry (ECR) for storing Docker images.
4. **ECS Cluster and Fargate Task Definition**:
   * Defines an ECS cluster and Fargate-based service.
   * Deploys the containerized application with an ALB for load balancing.
   * Manages IAM roles and policies for task execution and logging.
5. **Security and IAM Roles**:
   * Separate security groups for ALB and ECS tasks.
   * IAM role for ECS task execution with access to ECR and CloudWatch logs.

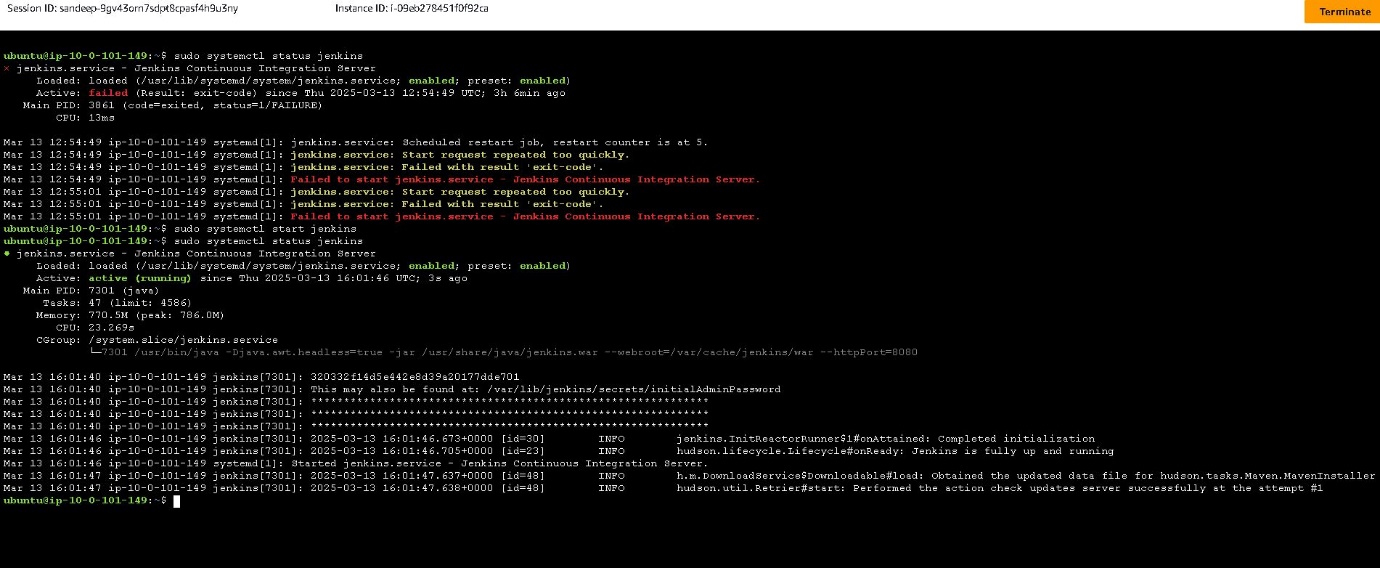






1. **Jenkins Setup**

On the Jenkins master server make sure that the Jenkins service is running, else start the service.

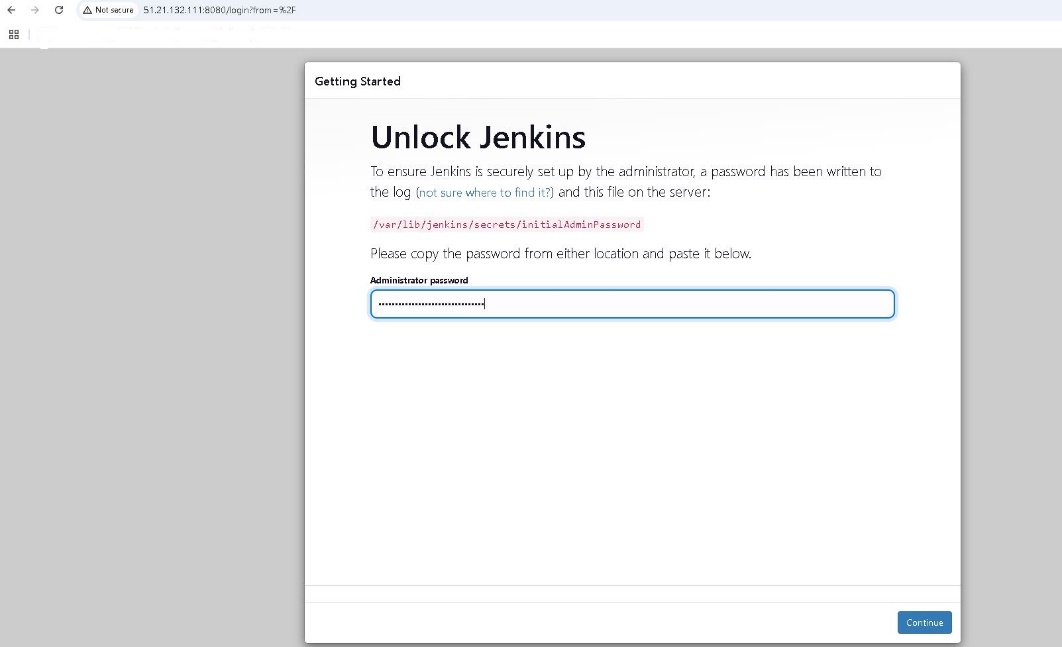


**1. Access Jenkins Master**

* Get Jenkins master public IP from Terraform output
* Access Jenkins UI: http://<jenkins\_master\_public\_ip>:8080
* Get initial admin password:

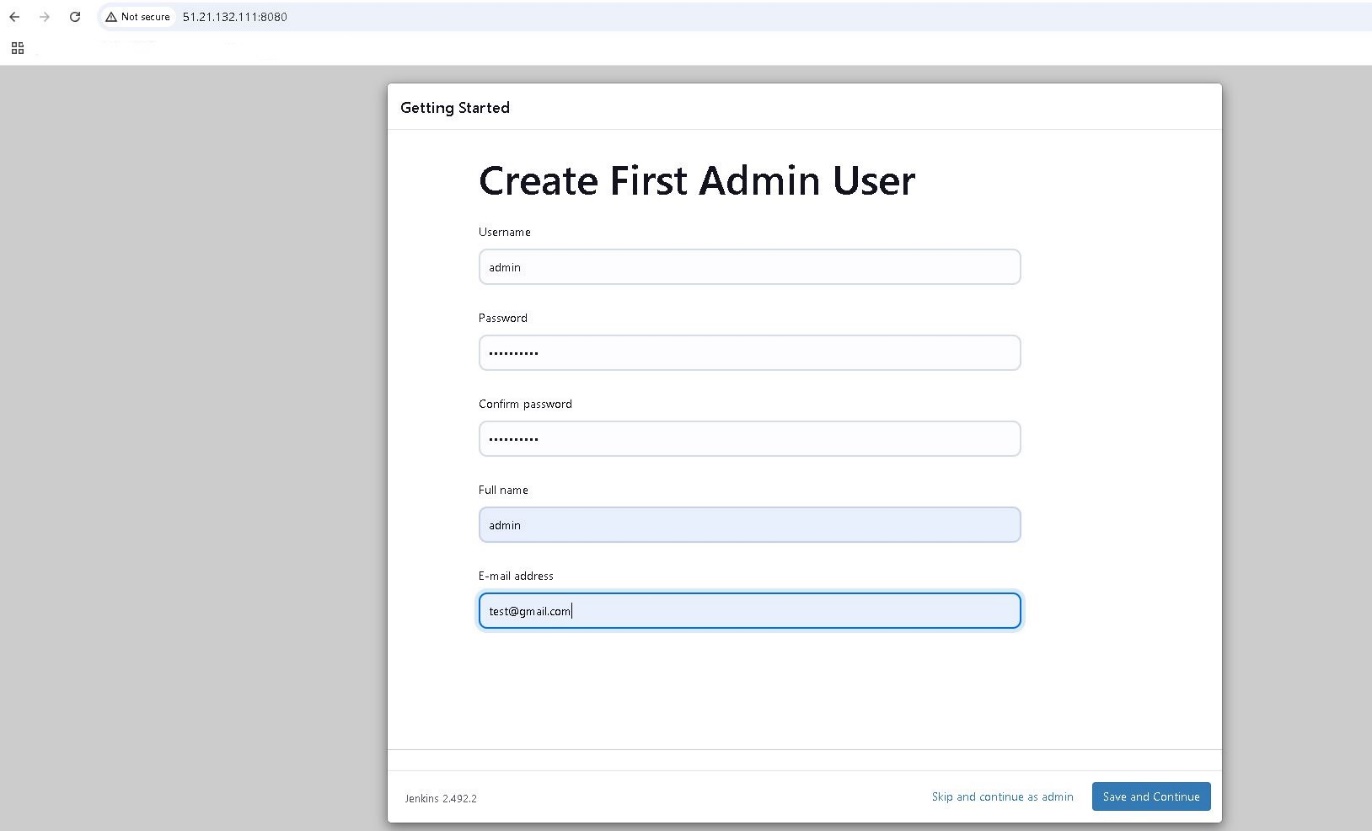
ssh -i jenkins-node-key.pem ubuntu@<jenkins\_master\_public\_ip>

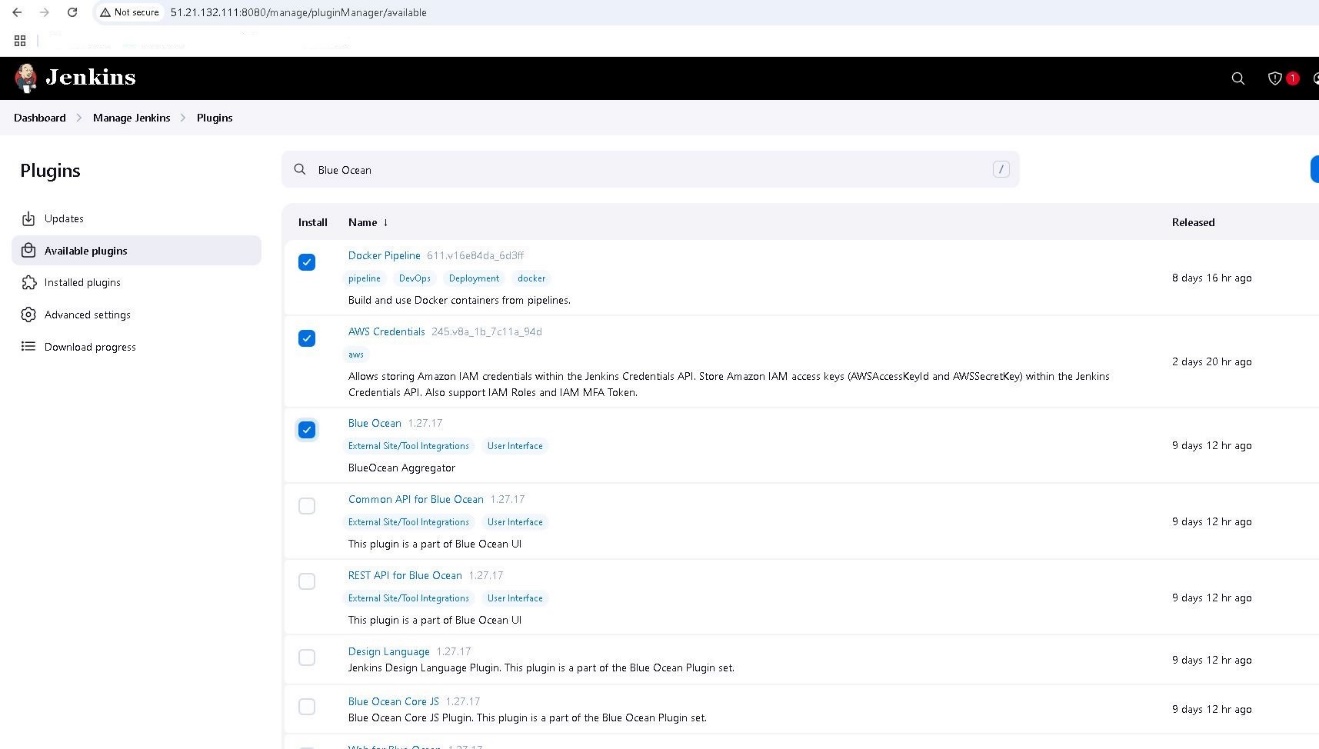
sudo cat /var/lib/jenkins/secrets/initialAdminPassword



**2. Configure Jenkins Master**

1. Install suggested plugins
2. Create admin user
3. Install additional plugins:
   * Docker Pipeline
   * AWS Credentials
   * Blue Ocean





**Docker Pipeline Plugin**

* **Purpose**: Allows Jenkins to interact with Docker containers directly within the pipeline.
* **Usage**: It enables building, running, and managing Docker containers and images as part of the CI/CD pipeline. This is especially useful for creating isolated build environments, running tests in containers, and deploying containerized applications to platforms like AWS ECS or Kubernetes.
* **Example Scenario**: Building a Docker image from a Jenkins pipeline and pushing it to AWS ECR.

**2. AWS Credentials Plugin**

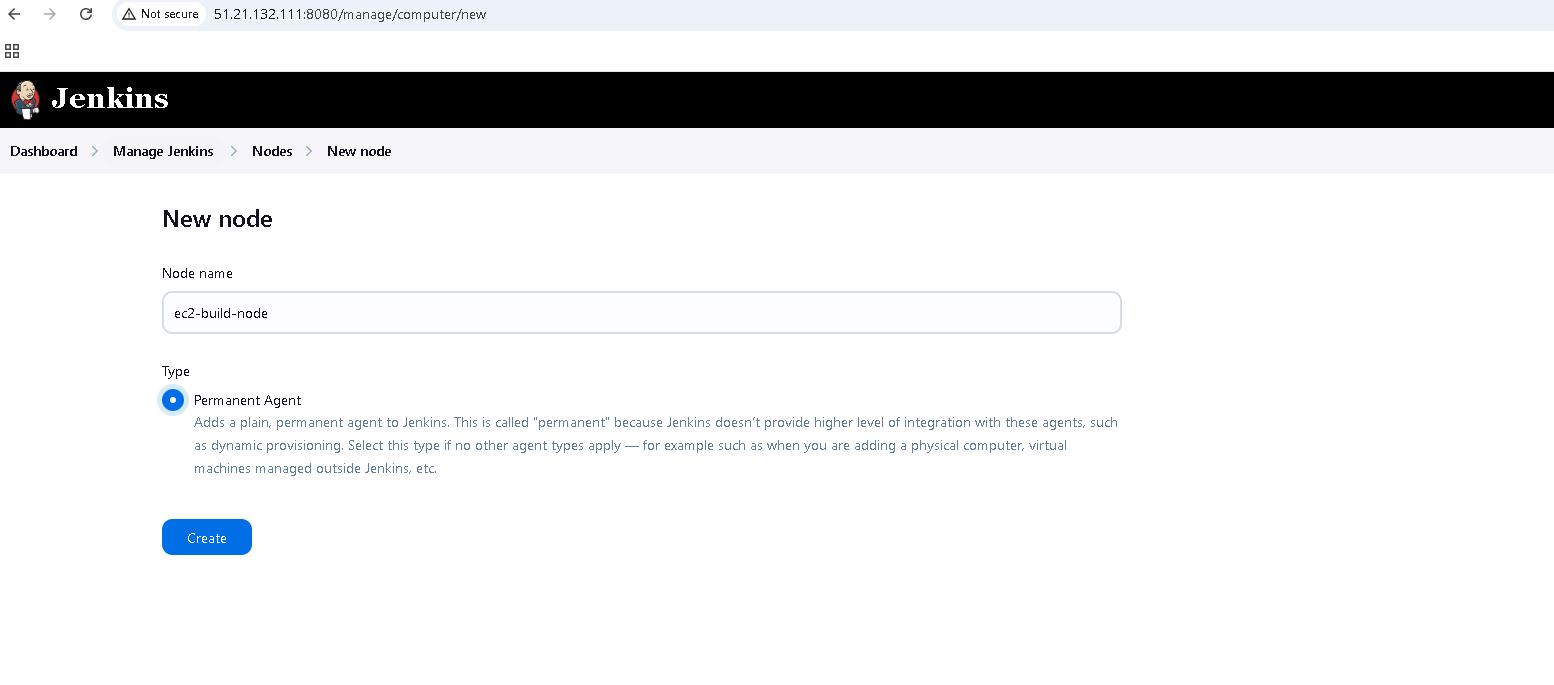
* **Purpose**: Provides a secure way to manage AWS credentials within Jenkins.
* **Usage**: Allows Jenkins to access AWS resources such as S3, EC2, and ECS by securely storing and managing IAM user credentials or access tokens. It integrates with AWS CLI and SDKs, enabling actions like uploading artifacts to S3 or deploying infrastructure with Terraform.
* **Example Scenario**: Deploying a Docker container to AWS ECS or updating CloudFormation stacks from a Jenkins pipeline.

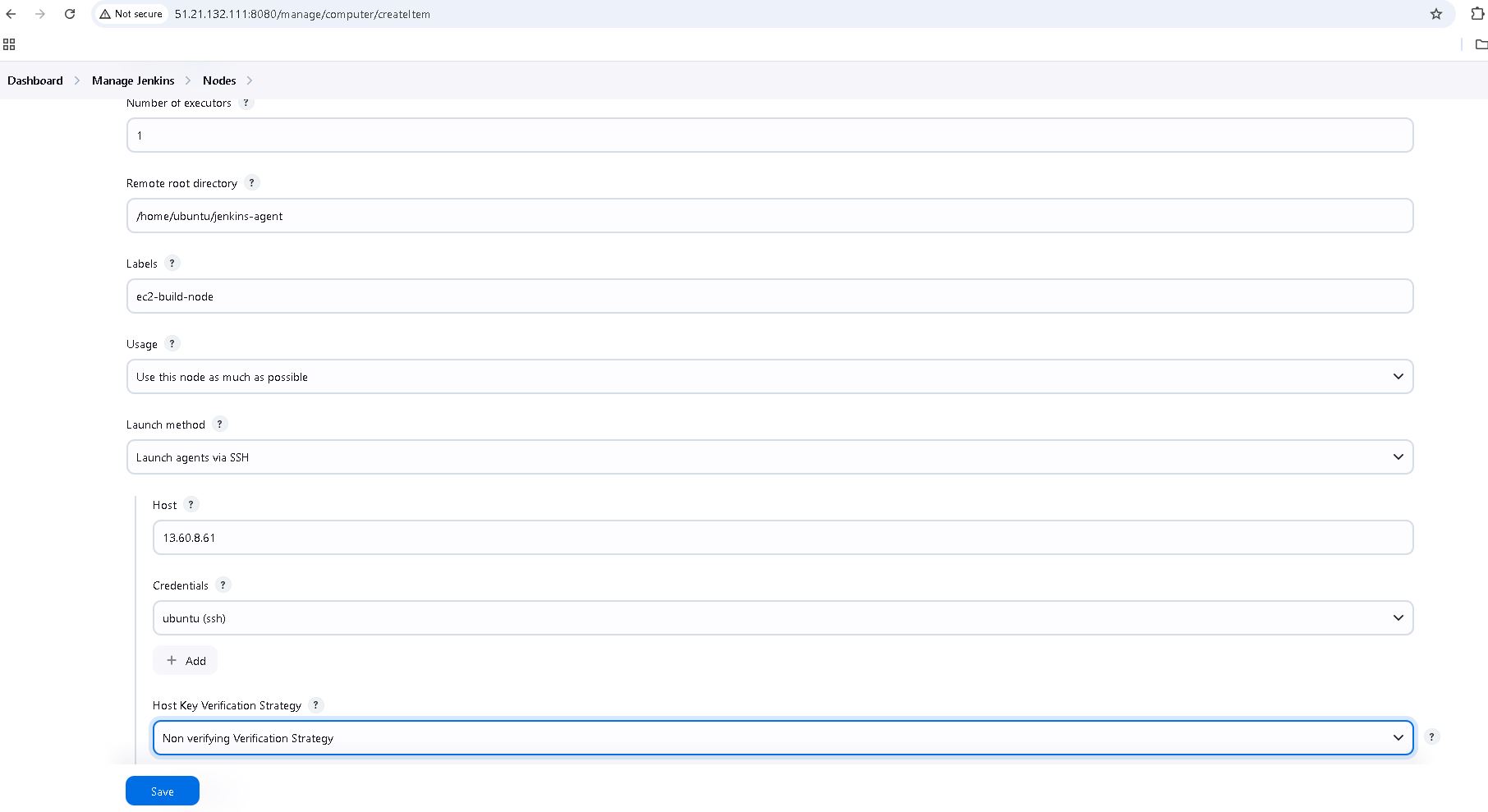
**3. Blue Ocean Plugin**

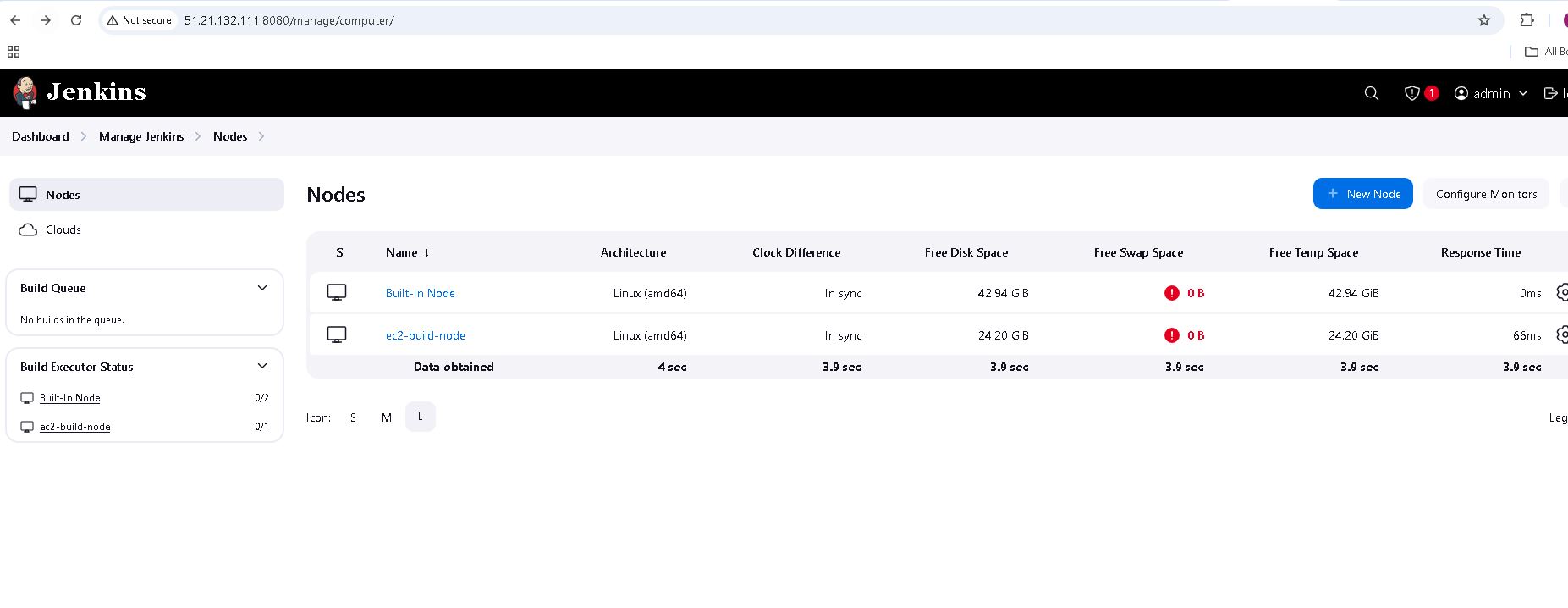
* **Purpose**: Provides a modern, user-friendly interface for Jenkins pipelines.
* **Usage**: Offers visual pipeline editing and monitoring, making it easier to understand and manage complex CI/CD workflows. It also supports pipeline visualization, parallel execution, and real-time status updates.
* **Example Scenario**: Viewing the stages of a CI/CD pipeline for a microservices deployment and debugging any failed steps visually.

**3. Configure Jenkins Build Node**

1. Go to Manage Jenkins → Manage Nodes
2. Add new node:
   * Name: ec2-build-node
   * Permanent Agent: Yes
   * Remote root directory: /home/ubuntu/jenkins-agent
   * Labels: ec2-build-node
   * Launch method: Launch agent via SSH
   * Host: <EC2\_INSTANCE\_PUBLIC\_IP> (from Terraform output)
   * Credentials: Add SSH with private key
   * Host Key Verification Strategy: Non verifying





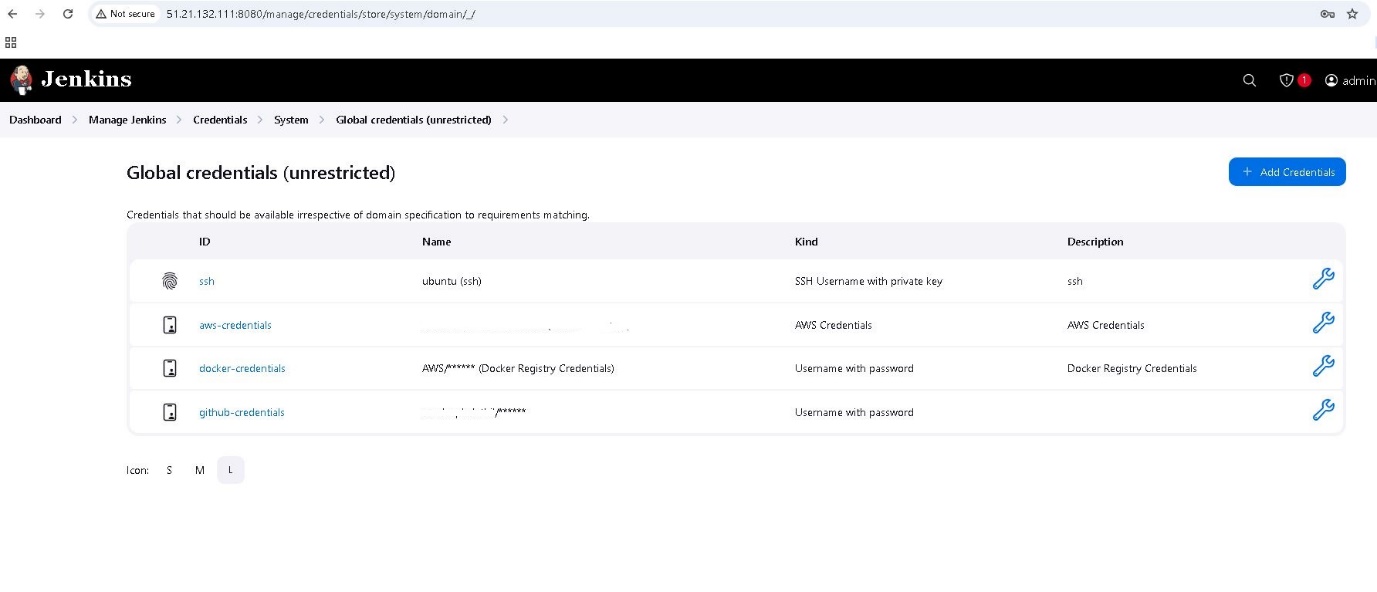


**4. Configure Jenkins Credentials**

1. AWS Credentials:  
   * Kind: AWS Credentials
   * ID: aws-credentials
   * Description: AWS Credentials
   * Access Key ID: Your AWS access key
   * Secret Access Key: Your AWS secret key
2. Docker Registry:  
   * Kind: Username with password
   * ID: docker-credentials
   * Description: Docker Registry Credentials
   * Username: AWS
   * Password: (Use AWS CLI get-login-password output)

Use command to generate password : aws ecr get-login-password --region eu-north-1

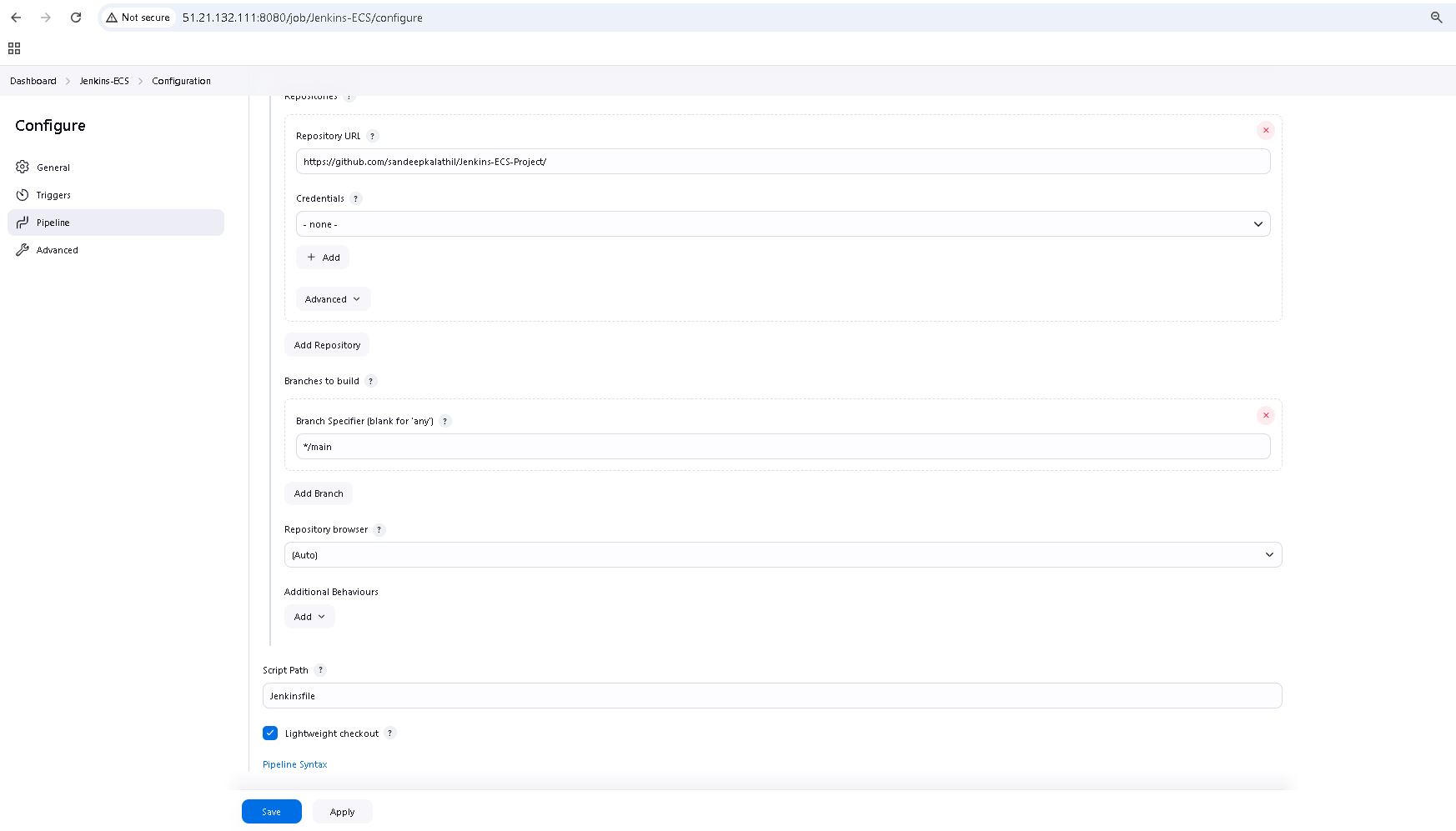
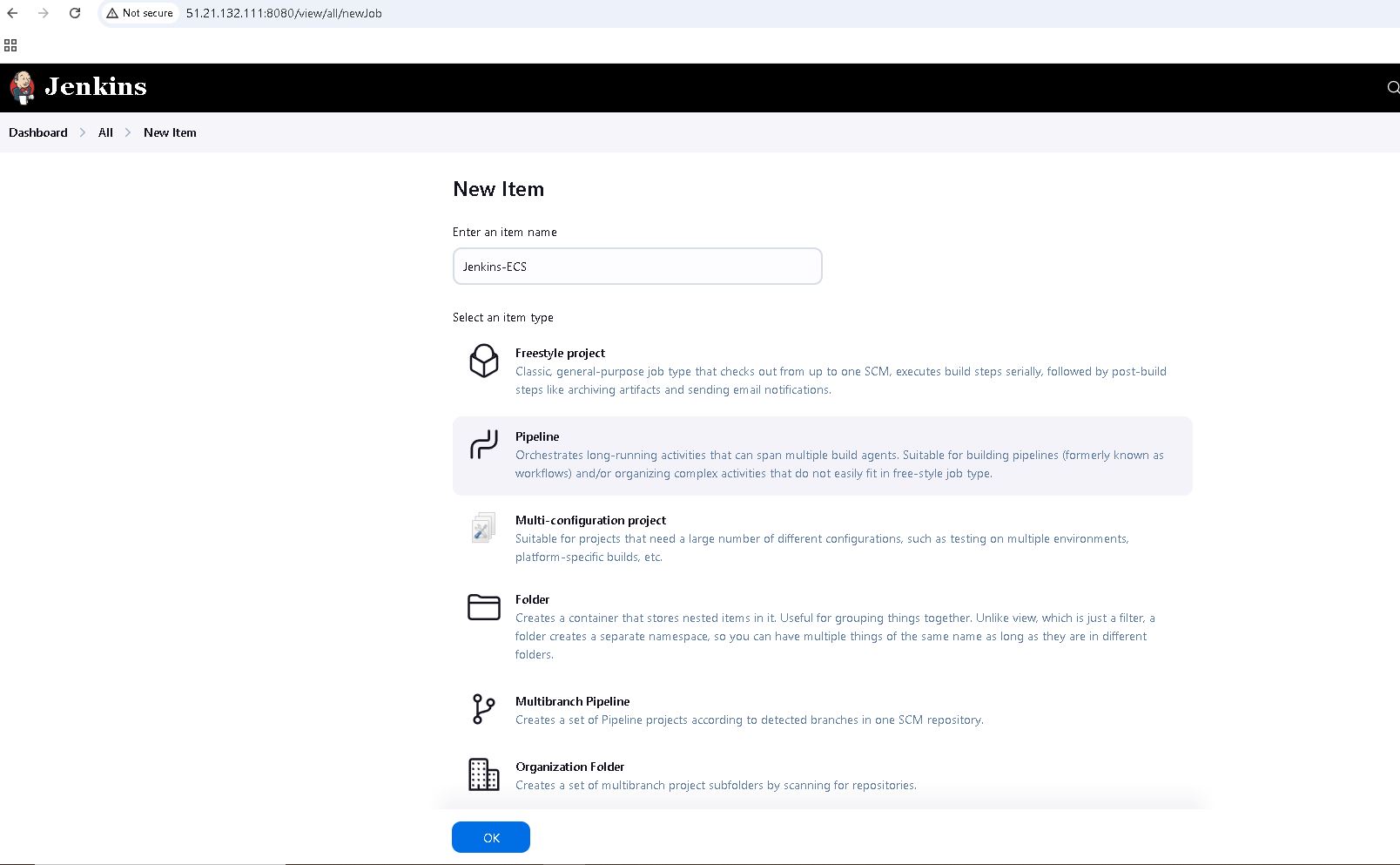
1. GitHub:  
   * Kind: Username with password
   * ID: github-credentials
   * Add your GitHub credentials



**Pipeline Setup**

**1. Create Jenkins Pipeline**

1. New Item → Pipeline
2. Configure Pipeline:
   * Definition: Pipeline script from SCM
   * SCM: Git
   * Repository URL: Your repository URL
   * Credentials: github-credentials ( in case of Private Repo)
   * Branch Specifier: \*/main
   * Script Path: Jenkinsfile



**2. Pipeline Stages**

**1. Build and Test**

* Runs in Docker container on build node
* NPM install and build
* Unit tests
* Static code analysis

**2. Docker Image Creation**

* Builds Docker image
* Tests image configuration

**3. Security Scan (Trivy)**

* Scans for vulnerabilities

**4. Push to ECR**

* Authenticates with ECR
* Pushes image with versioning

**5. Deployment**

* Updates ECS service
* Performs health checks

**Jenkins Pipeline Overview**

This pipeline is designed to build, test, scan, and deploy a Docker-based application to AWS ECS. It leverages Docker containers for build isolation, performs security scanning with Trivy, and pushes the Docker image to AWS ECR. Finally, it updates the ECS service and performs a health check on the deployed service.

**Stages Breakdown**

**Stage 1: Build and Test**

* Runs on an ec2-build-node with a custom workspace.
* Steps:
  1. Clean workspace.
  2. Checkout code from the SCM (e.g., Git).
  3. Run the build in a Docker container (node:18-alpine) with limited memory and CPU.
  4. Install dependencies (npm ci), run tests, lint, and build the app.
  5. Archive the build artifacts (dist.tar.gz).

**Stage 2: Build Docker Image**

* Builds the Docker image with --no-cache to ensure a fresh build.
* Runs a basic check (nginx -t) inside the container to validate the image.

**Stage 3: Trivy Security Scan**

* Runs a Trivy container to scan for vulnerabilities in the Docker image.
* Scans for HIGH and CRITICAL severity vulnerabilities and exits with a non-zero code on failure.

**Stage 4: Push to ECR**

* Logs in to AWS ECR and pushes the Docker image with retry logic.
* If the branch is main, tags the image as latest and pushes it.

**Stage 5: Deploy to ECS**

* Updates the ALB health check settings.
* Fetches the current ECS task definition and modifies the image reference and port mapping using jq.
* Registers a new task definition and forces a new deployment in the ECS service.
* Waits for the service to stabilize.

**Stage 6: Health Check**

* Retrieves the ALB DNS name.
* Performs periodic health checks by sending requests to the service endpoint.

**3. Post Actions**

* **Success**: Prints a success message.
* **Failure**: Prints a failure message.
* **Always**: Cleans up Docker images, removes unused containers, and clears the workspace.

**4. Error Handling and Retry Logic**

* Handles exceptions for each stage and sets the build status to FAILURE.
* Retries critical steps like ECR login and image push to handle transient errors.

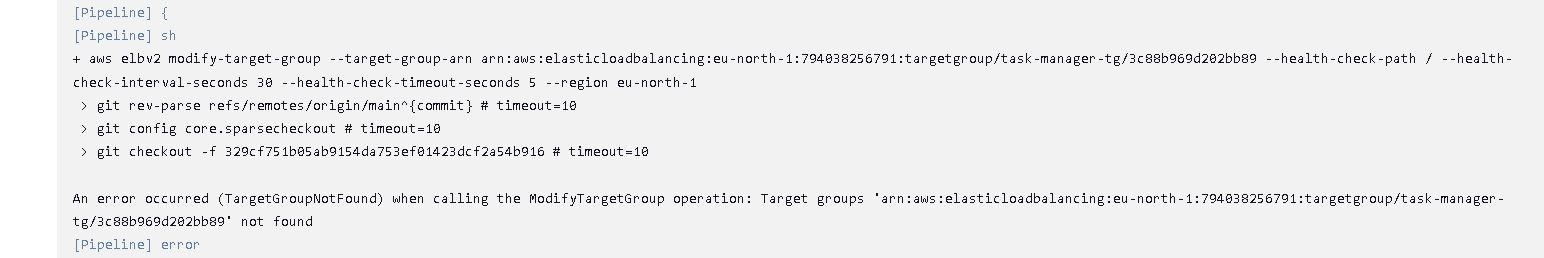
**5. AWS CLI Commands Used**

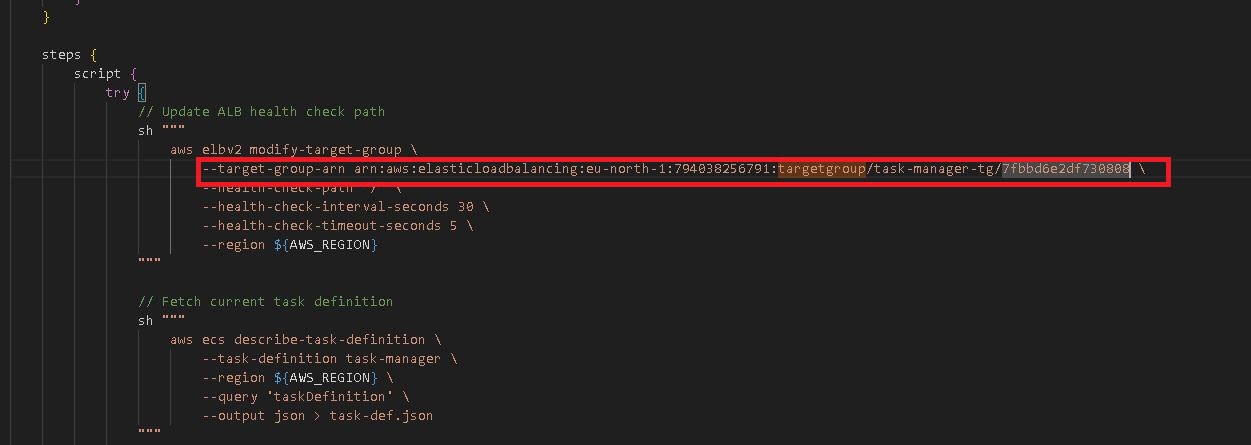
* aws elbv2 modify-target-group: Updates ALB health check.
* aws ecs describe-task-definition: Fetches current task definition.
* aws ecs register-task-definition: Registers a new task definition.
* aws ecs update-service: Deploys the new task definition to ECS.
* aws ecs wait services-stable: Waits for the service to stabilize.

**6. Docker and Trivy Integration**

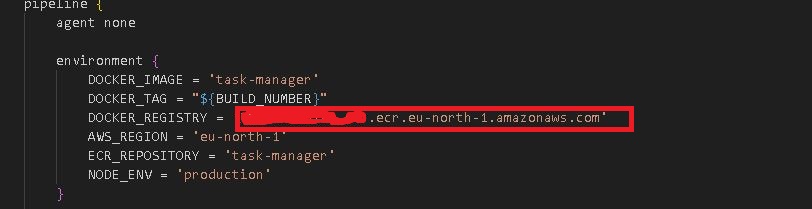
* Docker is used for build isolation and running tests.
* Trivy is used to scan Docker images for security vulnerabilities.

**Before proceeding with the build update the “Jenkinsfile” with the Target group ARN as shown below.**

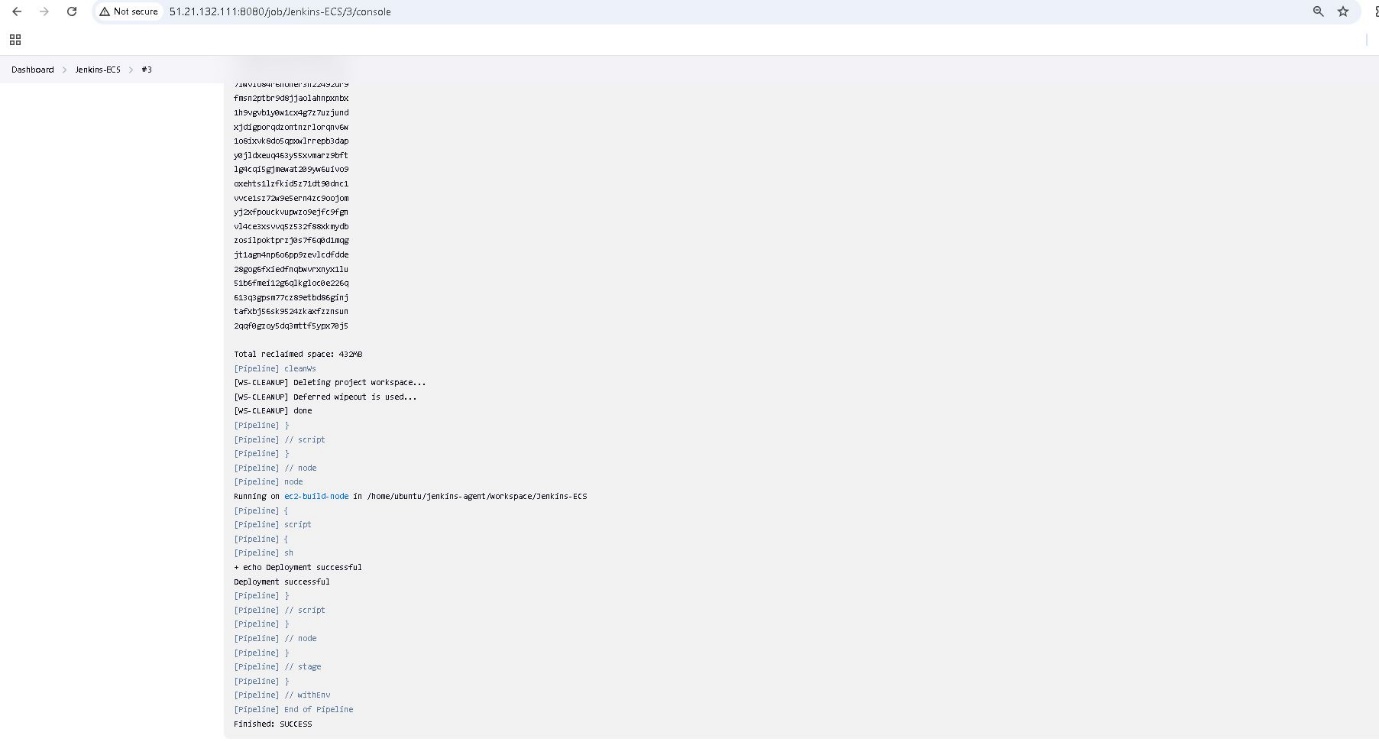


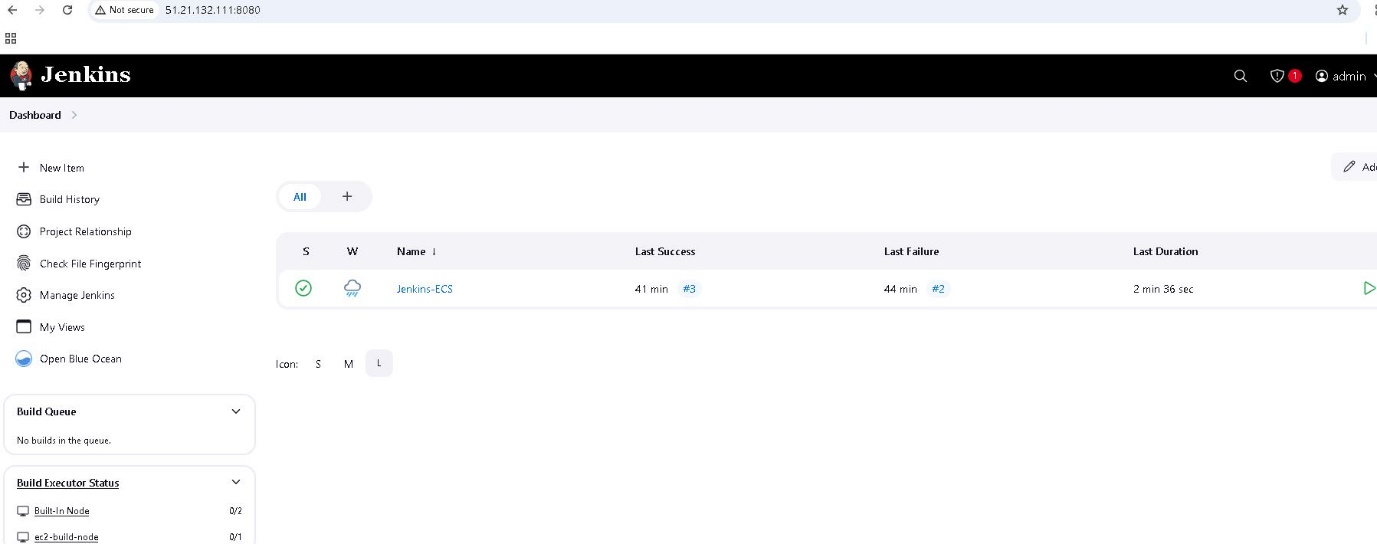


 Also in the Jenkins file update the ECR repository name ( Which can be found from the Terraform output)

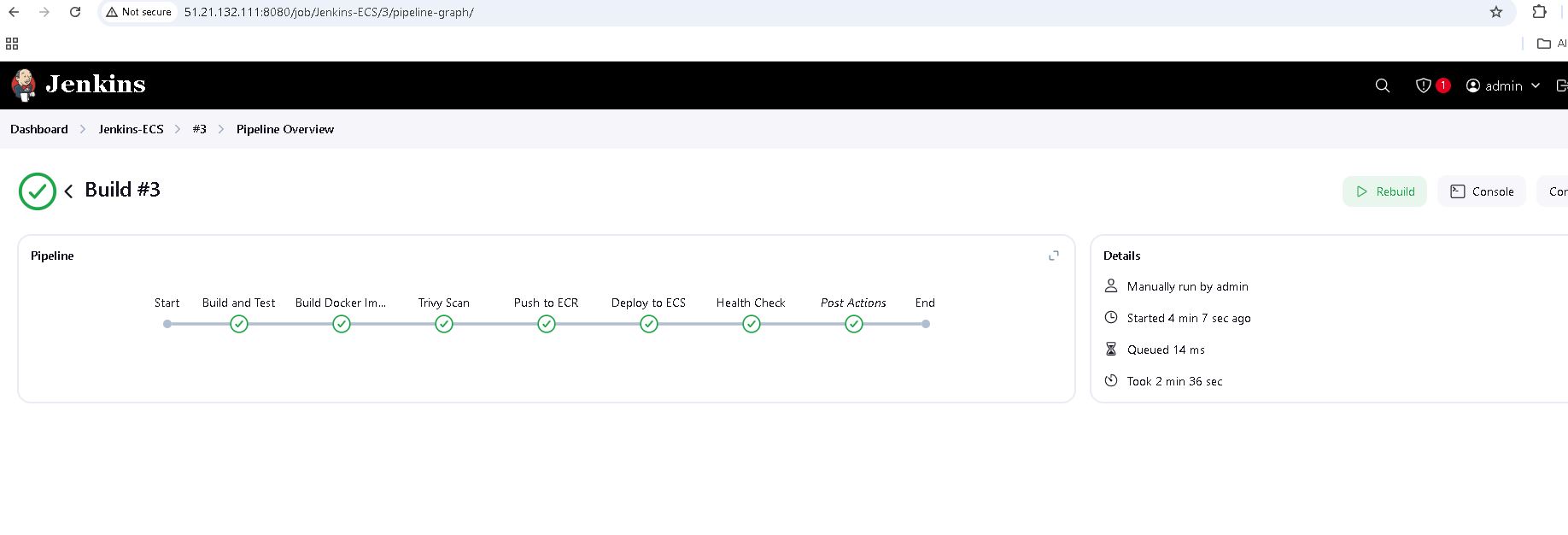


Once the build is complete the Jenkins build console will show Success message.

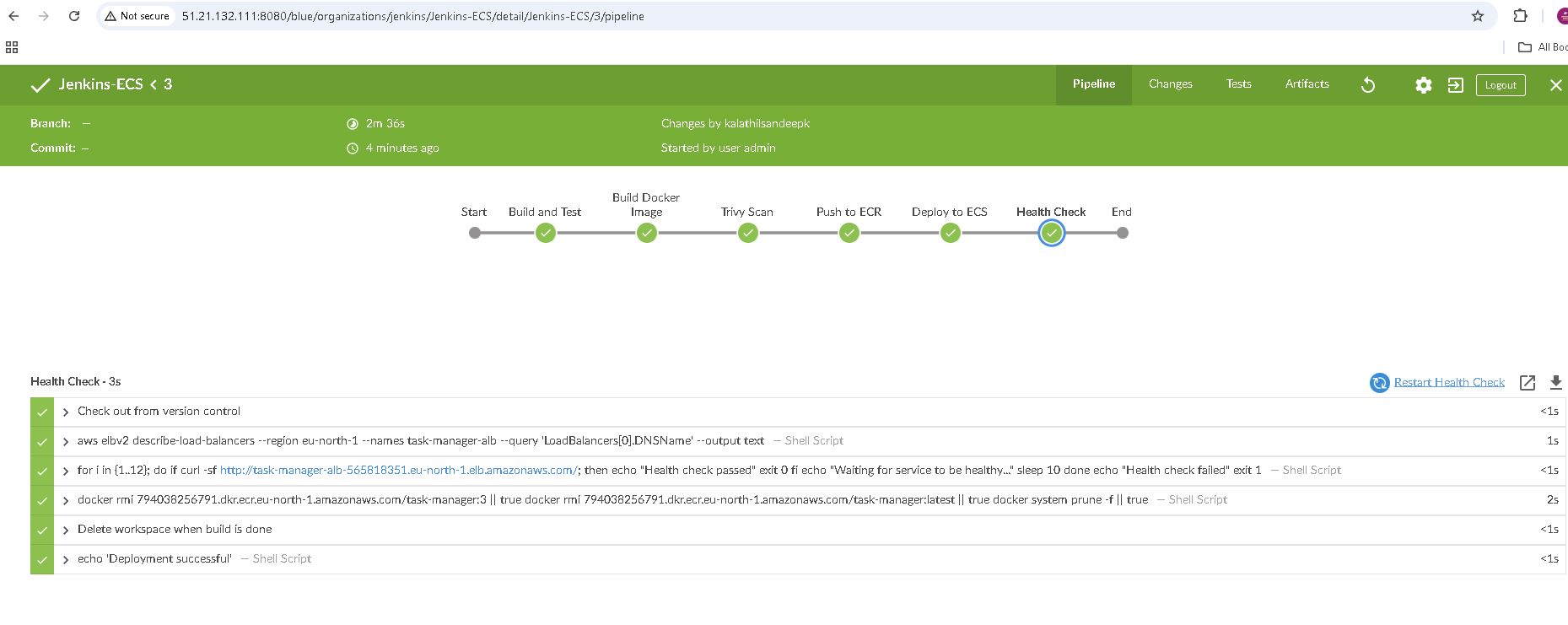




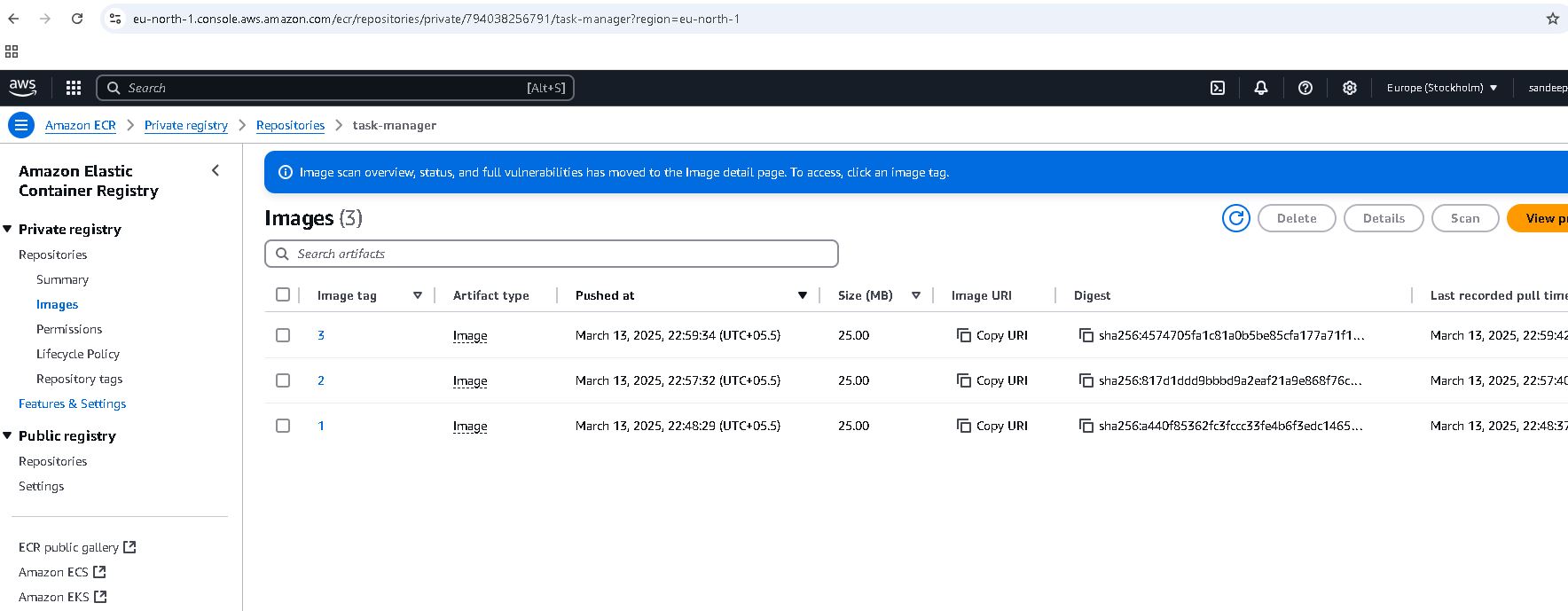
 The image shows various steps used in the build and its status.



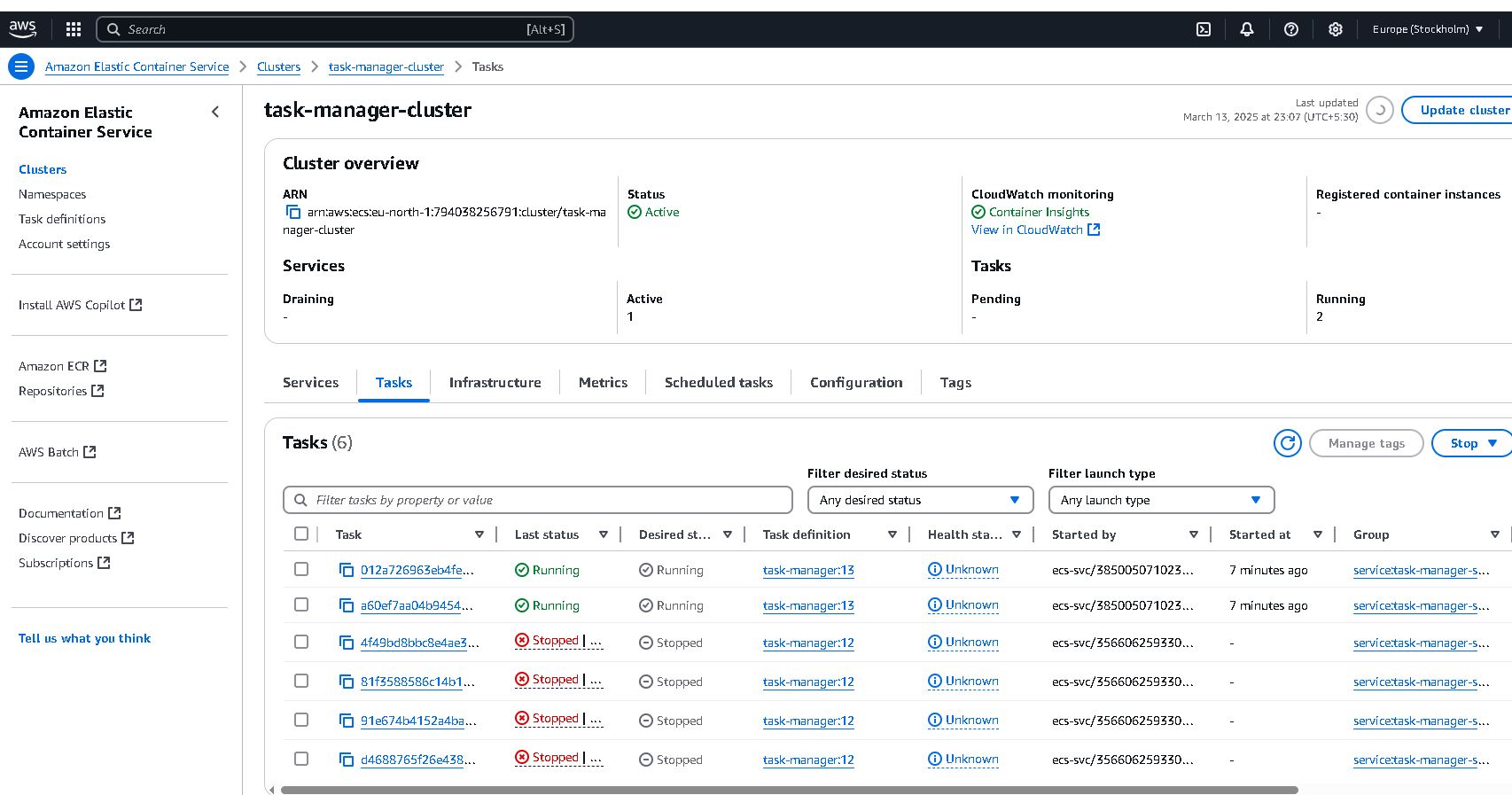
The below image is from Blue Ocean plugin interface. This also shows the Status in the pipeline.

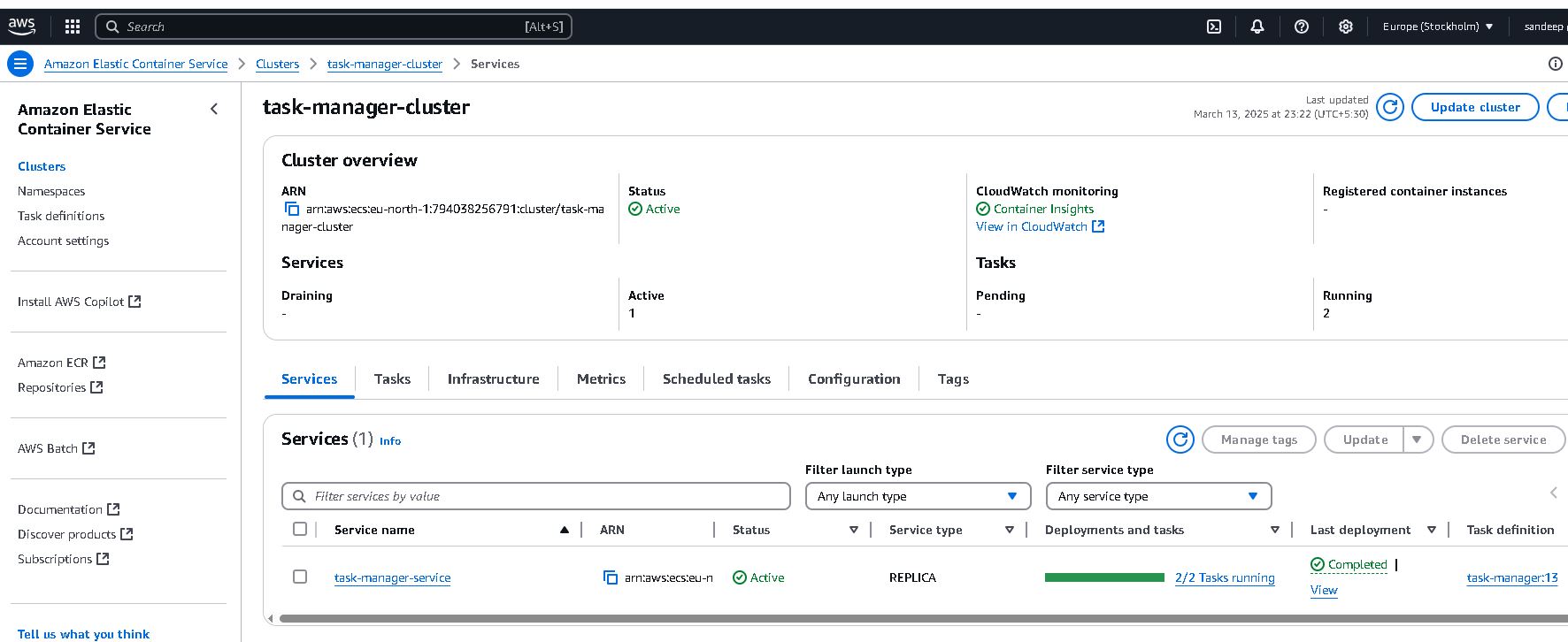


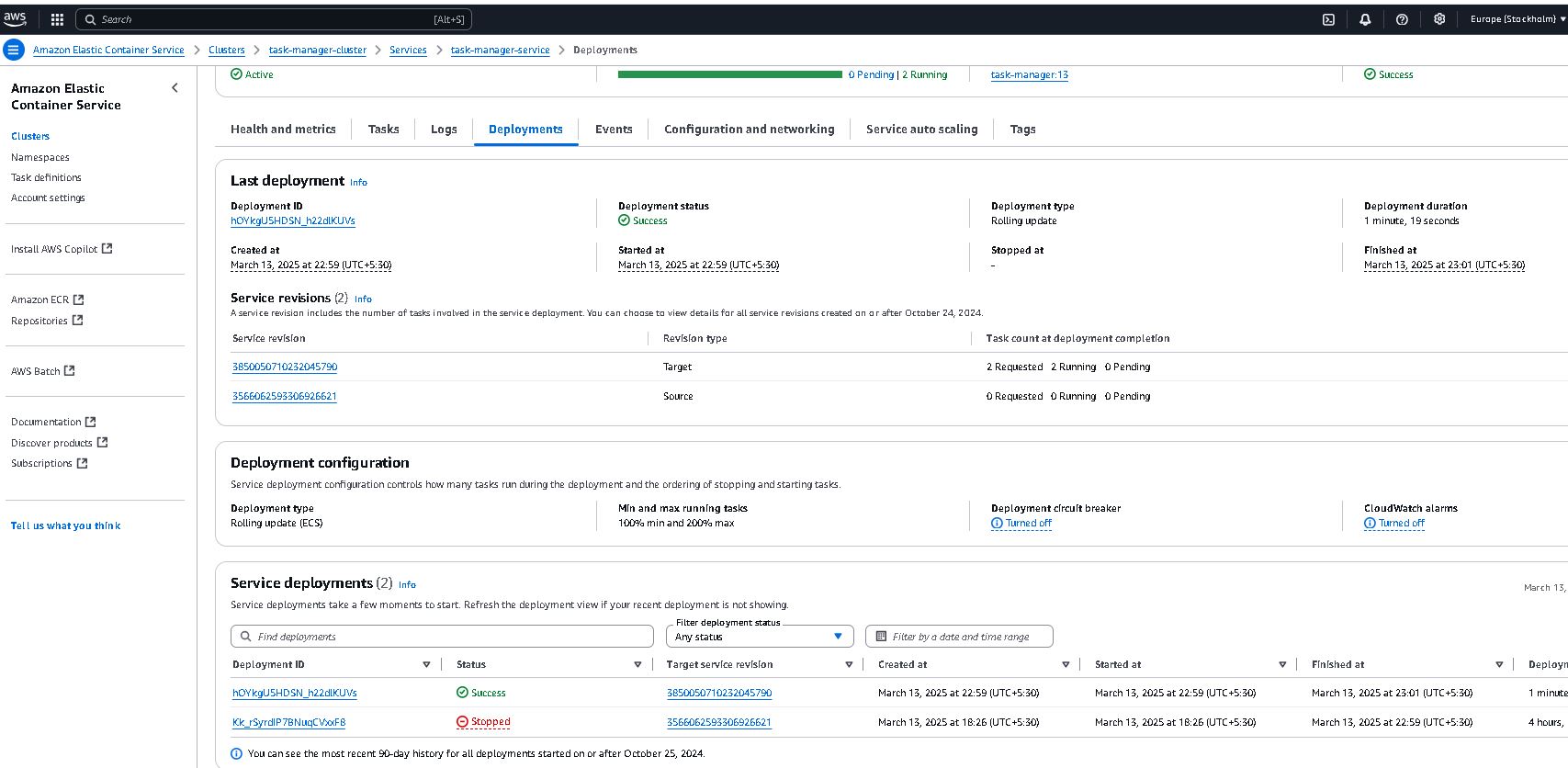
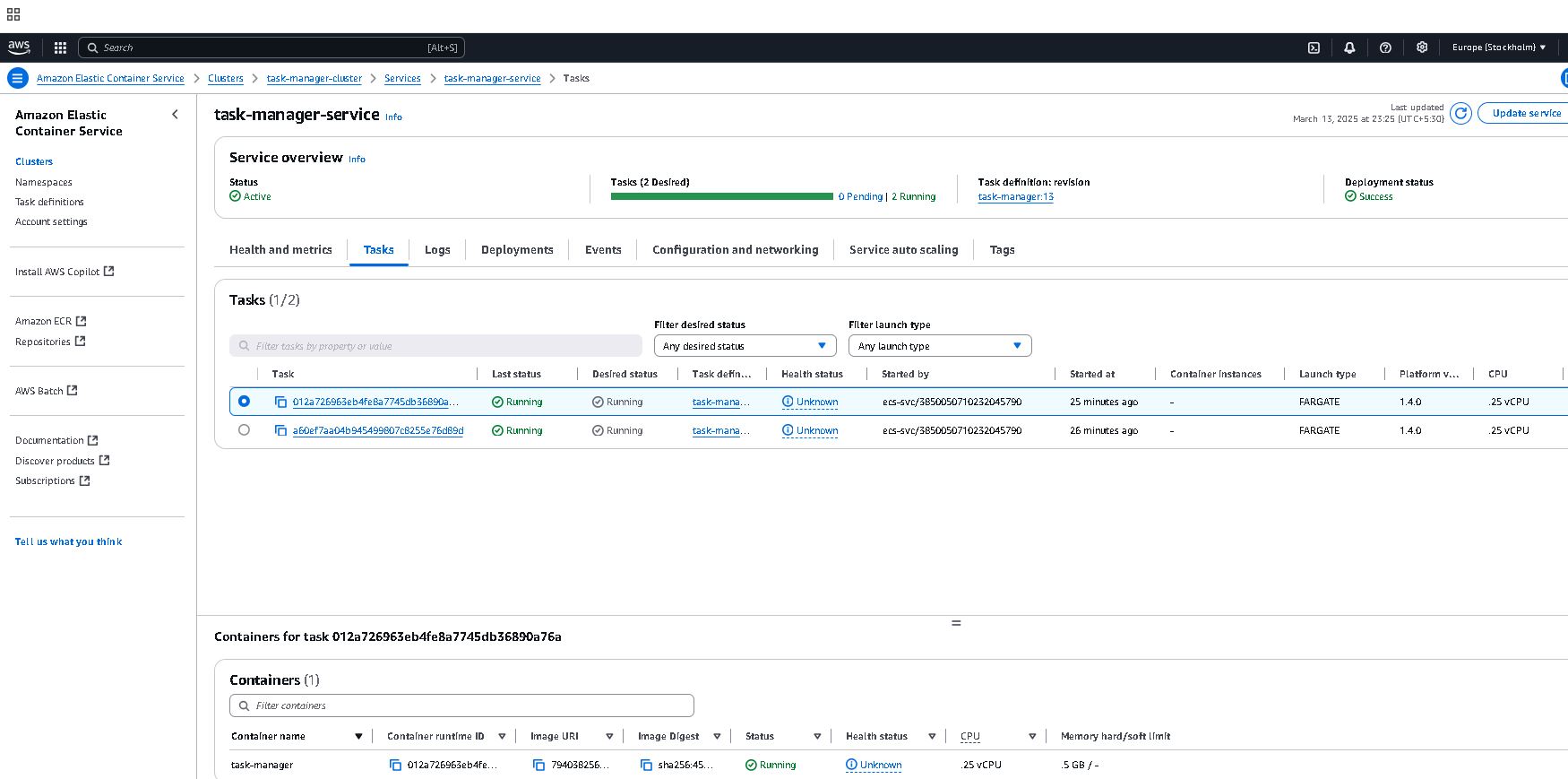
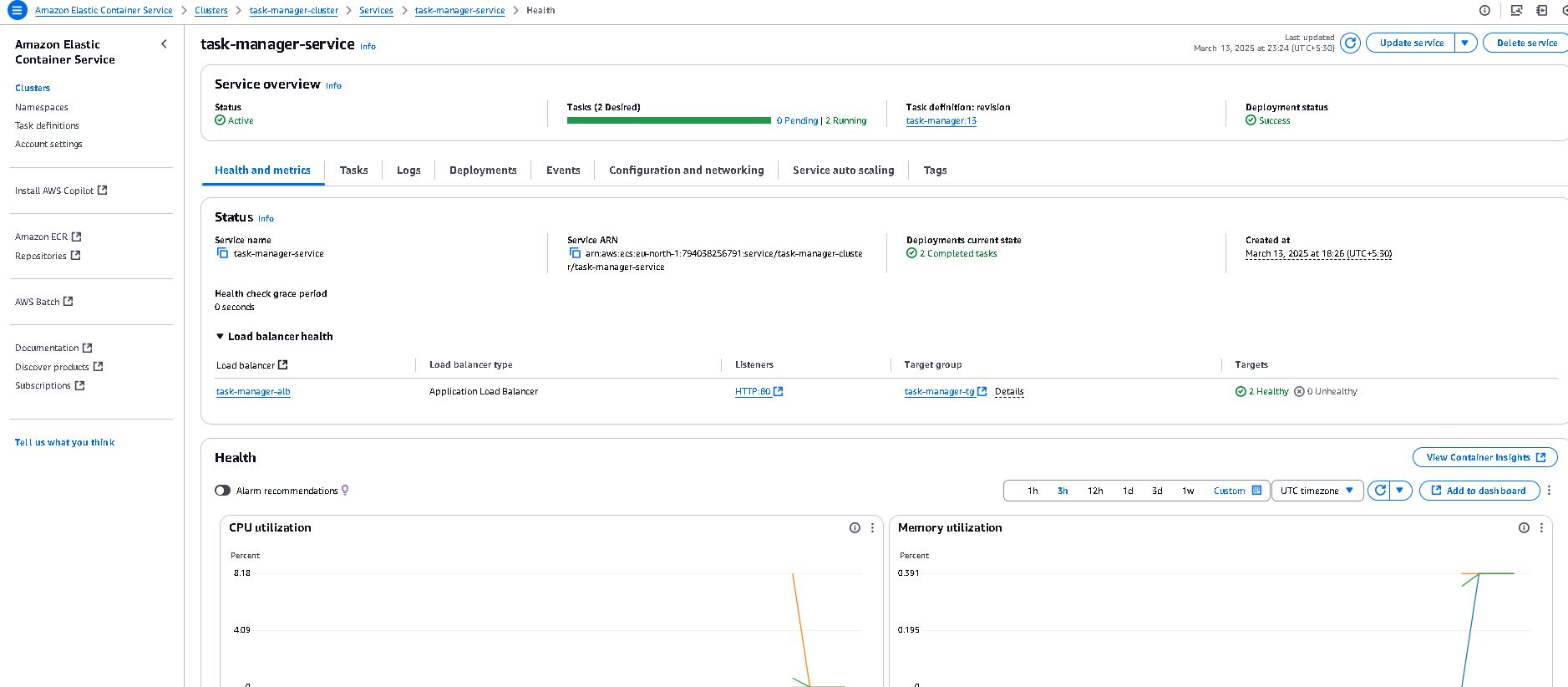
Verify that the newly built Docker images are present in ECR.

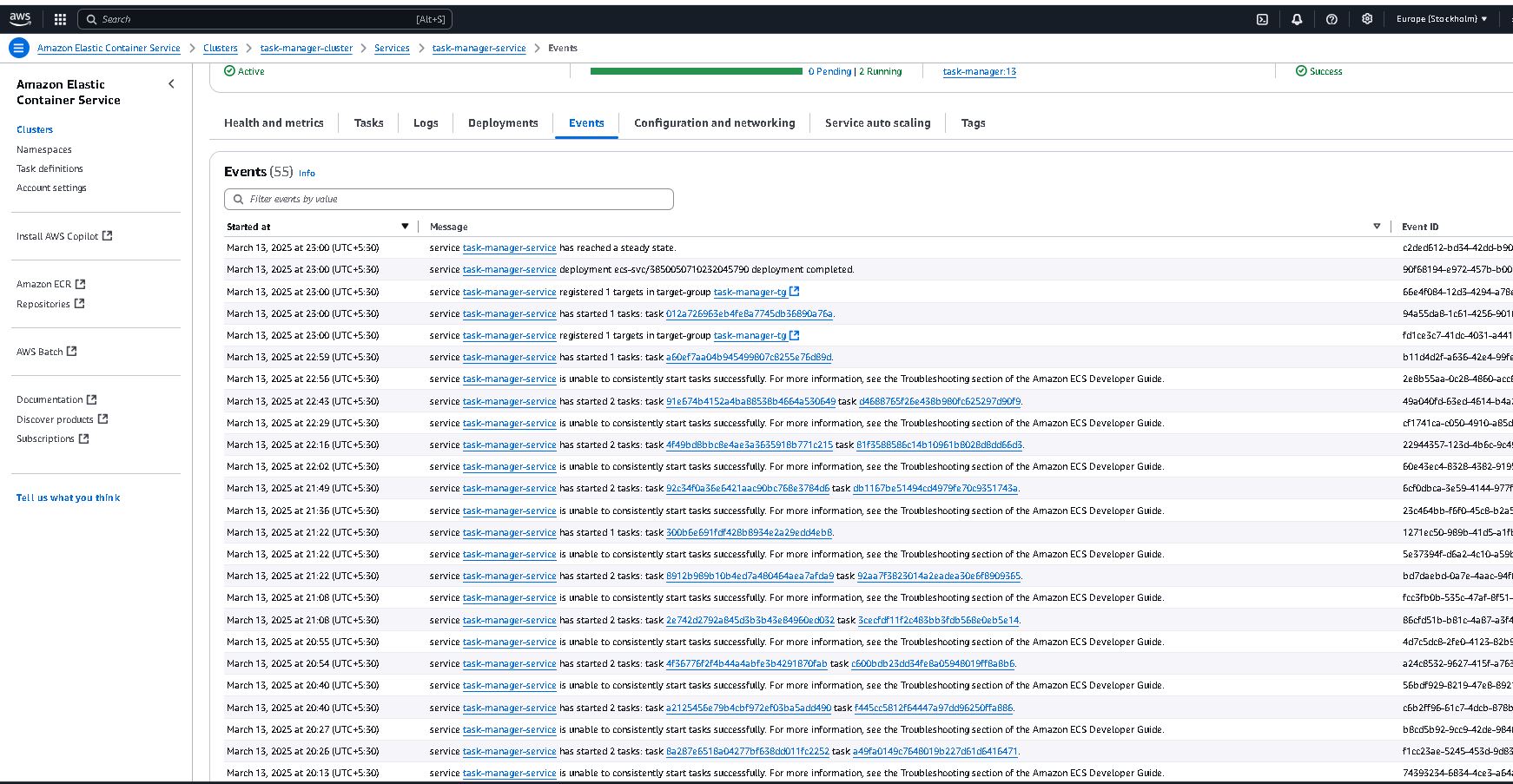


Check the ECS console to ensure tasks are running and services are active.

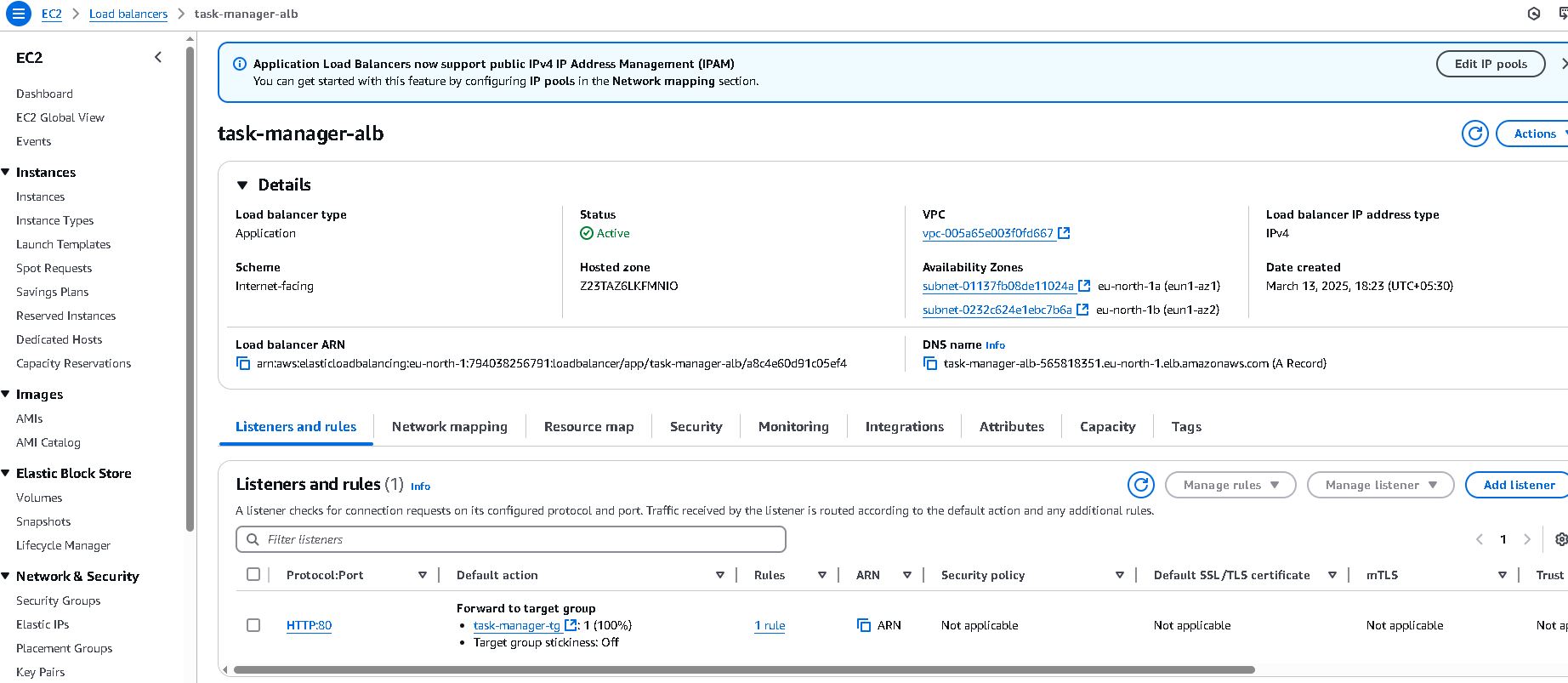








Use the ALB DNS name to access the deployed website in the browser.



Confirm that the website loads successfully and functions as expected.

