Project Report

On

DevSecOps for a Microservices App

Submitted in fulfillment for the award of

Post Graduate Diploma in IT Infrastructure System & Security

(PG-DITISS) from **CDAC ACTS (Pune)**

Guided By:

Mr. Soham Y

Presented By:

Anmol Pimpale	PRN: 250240123004
Srushti S. Pawar	PRN: 250240123031
Shreya R. Korde	PRN: 250240123042

Centre of Development of Advanced Computing (C-DAC), Pune

CERTIFICATE

TO WHOMSOEVER IT MAY CONCERN

This is to certify that

Anmol Pimpale	**PRN: 250240123004**
Srushti S. Pawar	**PRN: 250240123031**
Shreya R. Korde	**PRN: 250240123042**

Have successfully completed their project on

“DevSecOps for a Microservices App”

Under the guidance of

Mr. Soham Y

Project Guide

Mr. Soham Y

Project Supervisor

Mr.Roshan Gami

HOD ACTS

Mr. Soumyakant Behera

ACTS, Head Quarters,
Pune

ACKNOWLEDGEMENT

This project "DevSecOps for a Microservices App" was a great learning experience for us and we are submitting this work to Advanced Computing Training School (CDAC ACTS).

We all are very glad to mention the name of *Mr. Soham Y* for his valuable guidance to work on this project. His guidance and support helped us to overcome various obstacles and intricacies during the course of project work.

We are highly grateful to *Ms. Risha P.R.* (Manager, ACTS training Centre), C- DAC, for her guidance and support whenever necessary while doing this course Post Graduate Diploma in *IT Infrastructure, Systems and Security (PG-DITISS)* through C- DAC ACTS, Pune.

Our most heartfelt thanks go to *Ms. Divya* (Course Coordinator, PG- *DITISS*) who gave all the required support and kind coordination to provide all the necessities like required hardware, internet facility and extra Lab hours to complete the project and throughout the course up to the last day here in C- DAC ACTS, Pune.

Sincerely,

Anmol Pimpale

Srushti S. Pawar

Shreya R. Korde

ABSTRACT

This project presents a complete DevSecOps pipeline designed for deploying Python-based microservices in a secure, automated and scalable manner. Using GitHub for version control, Docker for containerization, and Jenkins for CI/CD orchestration, the pipeline ensures rapid and reliable delivery of services. SonarQube is integrated for static code analysis, enforcing secure coding practices. Helm charts are utilized for packaging microservices and deploying them to a Kubernetes cluster hosted on AWS (EKS).The architecture supports modular service deployment, rolling updates, and resource scaling. Security is enhanced through container image scanning and code quality gates. For observability, Prometheus is used to collect metrics, while Grafana provides visual dashboards for real-time monitoring of services and infrastructure. This end-to-end DevSecOps approach promotes automation, agility, and security throughout the software development lifecycle. The solution is suitable for modern cloud-native applications requiring rapid iterations with continuous monitoring and compliance.

TABLE OF CONTENTS

1. INTRODUCTION AND OVERVIEW OF PROJECT

1.1 PURPOSE

The purpose of this project is to design and implement a complete DevSecOps pipeline for Python-based microservices that ensures **secure, automated, and efficient development, testing, deployment, and monitoring**. With the rising complexity of **modern cloud-native applications, integrating development, security, and operations** into one seamless workflow becomes critical. This project leverages tools like **GitHub, Jenkins, Docker, SonarQube, Kubernetes, Grafana and AWS** to build a reliable and scalable infrastructure.

1.2 AIM AND OBJECTIVE

AIM: To build a secure and automated DevSecOps pipeline for deploying Python microservices using modern tools like Docker, Jenkins, SonarQube, Kubernetes, and AWS, ensuring fast, reliable, and scalable software delivery.

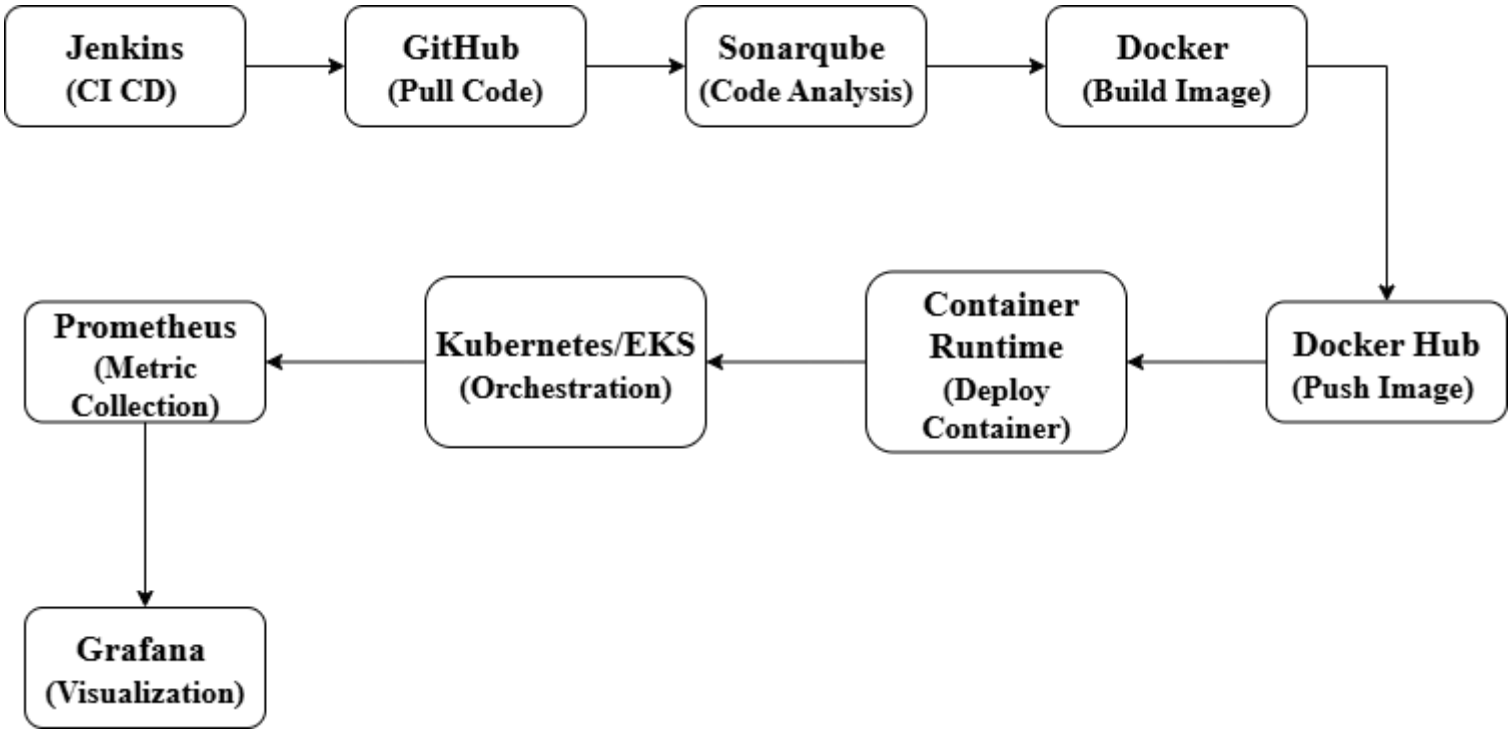
OBJECTIVE: The project aims to achieve the following objectives:

1. **Implement Python-based RESTful microservices** using a modular and scalable architecture.
2. **Integrate CI/CD automation with Jenkins** for building, testing, and deploying services.
3. **Containerize and orchestrate microservices** using Docker, Helm, and Kubernetes.
4. **Deploy infrastructure on AWS** ensuring scalability, reliability, and high availability.
5. **Apply DevSecOps practices** including code quality checks, vulnerability scanning, and monitoring.

2. OVERALL DESCRIPTION

2.1. INTRODUCTION:

This project presents a comprehensive DevSecOps solution for Python-based microservices, aimed at automating and securing the entire software development lifecycle. It combines development, security, and operations practices to deliver scalable, maintainable, and secure applications. Each microservice is independently developed using Python and containerized with Docker. GitHub is used for version control, while Jenkins automates the CI/CD pipeline. Code quality and security are enforced using SonarQube during integration. Helm is utilized for packaging and deploying services on a Kubernetes cluster hosted on AWS (EKS), ensuring high availability and scalability. For monitoring and observability, Prometheus is used to collect real-time metrics from the microservices and infrastructure, while Grafana provides rich, customizable dashboards for visualizing and analyzing these metrics. This end-to-end setup promotes faster delivery, early vulnerability detection, reduced manual effort, real-time monitoring, and improved reliability of services in a production-grade environment.



The above flow diagram represents the DevSecOps pipeline implemented for Python-based microservices, integrating key tools across the software development lifecycle. The process begins with the Development phase using Python, followed by Containerization with Docker to package the application. Continuous Integration is managed through Jenkins to automate builds and testing. Static Code Analysis using SonarQube ensures code quality, while Trivy performs Security Scanning to detect vulnerabilities. The application is then deployed using AWS cloud services and Argo CD for GitOps-based deployment automation. Finally, system Monitoring is carried

out using Prometheus, Grafana, and Node Exporter to track performance and ensure reliability. This comprehensive flow ensures secure, scalable, and efficient delivery of microservices in a cloud-native environment.

The process begins with application development in Python. Developers write modular, testable code in a version-controlled environment using Git. Once the code is committed, it is automatically pulled into a CI/CD pipeline managed by Jenkins. Jenkins orchestrates the various stages of automation, starting with building the Docker image of the application. Docker plays a key role here by packaging the application along with all its dependencies into a standardized container image, ensuring consistency across development, testing, and production environments.

In the sections that follow, we will delve into the intricacies of this project. We will explore the technologies, methodologies, and AWS services that underpin its foundation. From threat detection algorithms to **real-time alerting mechanisms**, from automated remediation to continuous improvement, this project will demonstrate how AWS can serve as an enabler of cutting-edge security practices in an ever-evolving digital landscape.

Before any deployment occurs, quality and security checks are enforced within the pipeline. Jenkins integrates with SonarQube to perform static code analysis, checking for bugs, code smells, and security vulnerabilities. This promotes code quality and maintains clean coding standards across the team. Simultaneously, Trivy scans the Docker image and its dependencies for known vulnerabilities and misconfigurations. These security scans are essential in identifying issues early in the lifecycle, preventing insecure applications from reaching the production environment.

Once the application passes all quality and security gates, it is deployed to a cloud platform like AWS. Argo CD, a GitOps-based continuous delivery tool, automates this deployment by monitoring the Git repository for changes in Kubernetes manifests and synchronizing them with the Kubernetes cluster hosted on AWS. This ensures that the desired state of the application is always reflected in the production environment, with the added benefits of version control, rollback capabilities, and increased deployment visibility.

Post-deployment, monitoring and observability become critical. Prometheus collects real-time metrics from the application and system infrastructure, including data like CPU usage, memory, and network activity, using exporters such as Node Exporter. Grafana then visualizes these metrics on interactive dashboards, enabling DevOps teams to monitor the system’s health and respond to issues proactively. This observability stack ensures reliability, performance optimization, and service-level assurance across the deployed applications. Overall, this DevSecOps pipeline architecture promotes speed, security, and scalability in a production-grade cloud environment.

2.2 Aspect and Description

Aspect	Description
Project Title	DevSecOps for a Microservices App
Introduction	- This project focuses on building a secure and automated DevSecOps pipeline for Python-based microservices using tools like GitHub, Jenkins, Docker, SonarQube, Kubernetes, and AWS. - It aims to integrate development, security, and operations into a single workflow that supports continuous integration, quality assurance, and scalable cloud deployment.
Challenges Addressed	- Secure Code: Detect bugs and vulnerabilities early with SonarQube. - Automated Deployment: Reduce errors using Jenkins and Docker. - Scalable Infra: Deploy efficiently with Kubernetes on AWS.
Project Goals	- Ensure secure and high-quality code with early vulnerability checks. - Achieve automated, error-free deployment through CI/CD. - Enable scalable and reliable microservice deployment on AWS using Kubernetes.

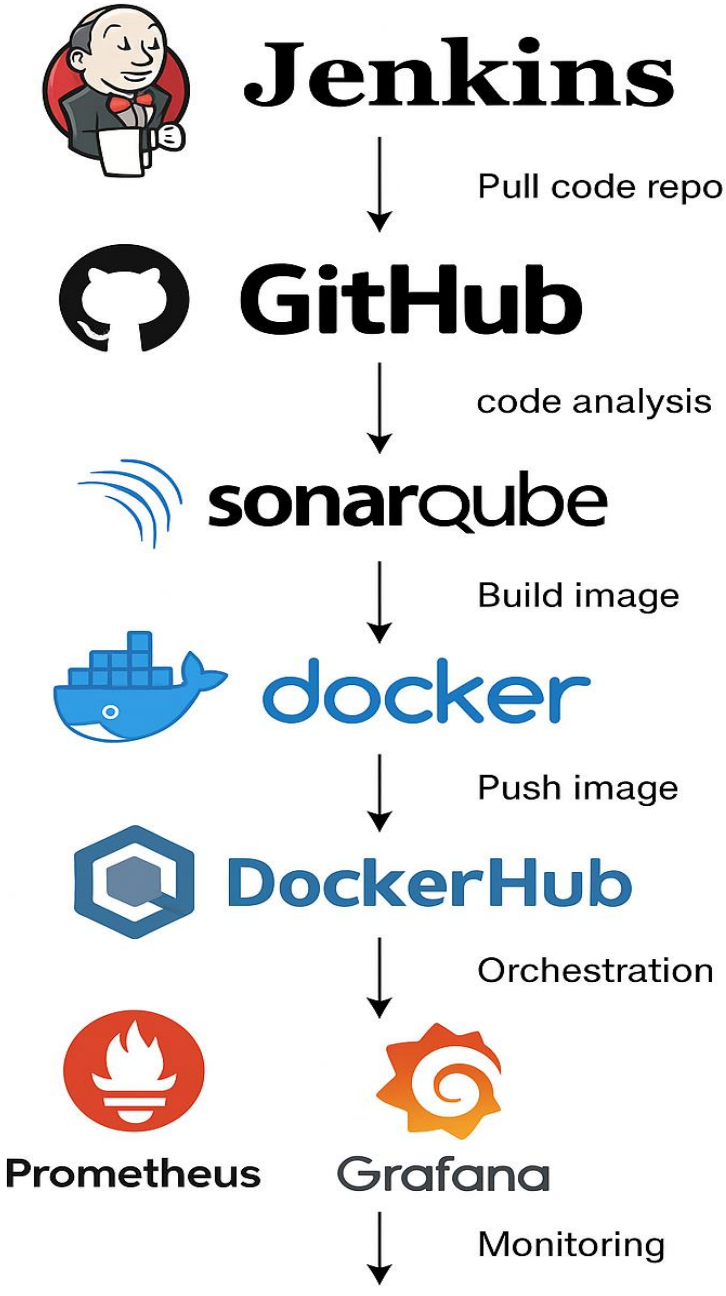
Key AWS Services	- Amazon EKS – Manages and runs Kubernetes clusters for microservice deployment. - Amazon EC2 – Hosts Jenkins, SonarQube, or supporting tools as virtual servers. - AWS IAM – Manages user and service permissions securely. - AWS VPC – Provides isolated and secure networking for cloud infrastructure.

Methodology	- Tool Integration: Set up GitHub, Jenkins, Docker, and SonarQube for secure CI/CD automation. - Deployment: Use Helm to deploy Python microservices on Kubernetes hosted on AWS. - Security & Monitoring: Perform code analysis, image scanning, and monitor services post-deployment.
Benefits	- Faster deployments through automated CI/CD. - Early security checks with SonarQube and image scanning. - Scalable and resilient architecture using Kubernetes on AWS. - Improved code quality via continuous analysis. - Modular microservices for easy maintenance and updates.
Project Significance	- Promotes secure and automated software delivery using modern DevSecOps practices. - Enables scalable, cloud-native deployment of Python microservices. - Bridges development, security, and operations for faster and reliable releases.
Project Components	- Python Microservices - GitHub – Source code and version control - Jenkins – CI/CD pipeline automation - Docker – Containerization of services

	- SonarQube – Code quality and security checks - Kubernetes – Service orchestration - AWS – Cloud infrastructure for hosting - CI/CD Pipeline – Automates build, test, deploy - Security Tools – Code scan & vulnerability check
Future Implications	- Enterprise Adoption: Can be scaled and adopted by large organizations for secure, cloud-native application delivery. - AI/ML Integration: The pipeline can support future AI-based microservices with automated model deployment. - Multi-Cloud Flexibility: Architecture can be extended to support deployments across multiple cloud providers (e.g., Azure, GCP). - Advanced Security Automation: Can integrate with tools like Trivy, Snyk, or OPA for enhanced automated security policies. - Continuous Innovation: Supports rapid feature updates and experiments through safe, automated delivery pipelines.

3.ARCHITECTURE

The project architecture is based on Python microservices deployed through a secure DevSecOps pipeline. Source code is managed in GitHub, while Jenkins handles continuous integration and delivery. SonarQube is used for static code analysis to ensure code quality and security. Each microservice is containerized using Docker, then stored in a container registry. Kubernetes (hosted on AWS) orchestrates deployment, scaling, and service management. This layered architecture ensures modular development, automated testing, secure delivery, and scalable deployment in a cloud-nati



4. System Requirements

4.1 Platform Used

EC2

In this project, an Amazon Elastic Compute Cloud (Amazon EC2) instance serves as the core environment for deploying and configuring key DevSecOps tools and microservices. The EC2 instance provides scalable and flexible compute power for setting up the development and deployment infrastructure.

Key Configuration Details:

- Operating System: Ubuntu (Linux)
- Instance Type: General-purpose (e.g., t2.medium or similar)
- Memory: 30 GB (EBS storage)
- Access Method: SSH using a .pem key from a local terminal
- Security Group: Custom-configured to allow only required ports

Purpose of the EC2 Instance:

- Jenkins Installation: Used to host Jenkins for automating the CI/CD pipeline.
- Docker Setup: Installed to build, manage, and run containers for Python-based microservices.
- Tool Hosting: The same EC2 instance hosts essential tools like:
  - SonarQube (for static code analysis)
  - Prometheus and Grafana (for monitoring and visualization)
- Microservice Testing: Used for deploying and testing Python microservices in a secure and controlled environment.

Security Group Configuration:
A custom security group was created and attached to the EC2 instance to control network traffic. Only the following ports were allowed:

- Port 22: SSH access from the local machine
- Port 8080: Jenkins dashboard
- Port 5100: Flask microservice
- Port 9000: SonarQube
- Port 9090: Prometheus
- Port 3000: Grafana

Instance Usage:

- A single Ubuntu-based EC2 instance was provisioned.
- Manual installation and configuration of required DevSecOps tools were performed via SSH.
- The instance remained the central hub for building, deploying, and monitoring microservices.

IAM [Identity and Access Management]

AWS Identity and Access Management (IAM) plays a vital role in ensuring secure access to resources and controlling user permissions. Here's a brief description of how IAM works within this project:

Purpose of IAM Role:

- To authenticate and authorize the user or service creating and managing the EKS cluster.
- To grant fine-grained access control for AWS resources used by Kubernetes components.
- To separate responsibilities and secure access without exposing permanent credentials.

How IAM Role is Used:

- A dedicated IAM role with EKS-related permissions was created to allow users or services to:
  - Create and manage EKS clusters
  - Launch and configure EC2 worker nodes (node groups)
  - Interact with other AWS services such as VPC, IAM, and Auto Scaling

- The IAM role includes managed policies such as:
  - AmazonEKSClusterPolicy
  - AmazonEKSWorkerNodePolicy
  - AmazonEC2ContainerRegistryReadOnly
  - AmazonEKS_CNI_Policy
  - AmazonVPCFullAccess (if networking setup is manual)

- This role was assumed during the EKS cluster creation process via the AWS Management Console or CLI.
- For node groups (EC2 instances running Kubernetes workloads), a Node Instance Role was created and attached, enabling nodes to join the cluster and pull container images securely.

Benefits of IAM Role Usage:

- Secure Access Control: Ensures only authorized users and services can perform EKS-related operations.
- Separation of Duties: Role-based permissions avoid over-privileged users.
- Temporary Credentials: Uses temporary security credentials, improving security posture.
- Auditable: All actions performed using the role are logged in AWS CloudTrail for auditing.

Amazon EKS

Amazon Elastic Kubernetes Service (AWS EKS) is used in this project to deploy and manage containerized applications in a scalable and production-ready Kubernetes environment. It serves as the orchestration platform for running the Dockerized Python microservices built through the DevSecOps pipeline.

Purpose of Using AWS EKS:

- To run microservices on a managed Kubernetes cluster
- To handle container orchestration, load balancing, rolling updates, and auto-scaling
- To simplify infrastructure management with AWS-managed control plane
- To ensure high availability, fault tolerance, and secure deployments

How EKS Is Used in the Project:

- **Application Deployment**
  The Dockerized application deployed on EKS worker nodes using Helm charts. Kubernetes manifests define deployments, services, and ingress configurations.

- **Node Management**
  EKS automatically manages the Kubernetes control plane, while the application runs on EC2 worker nodes (configured as part of an EKS node group).

- **Networking and Access**
  Kubernetes Services expose the microservices to be accessible within the cluster or externally (via LoadBalancer). Required ports are opened in the security group to allow traffic to reach the deployed application.

- **Scalability and High Availability**
  EKS supports auto-scaling of nodes and pods, ensuring that the application can scale according to traffic and resource needs while maintaining availability.

- **Security and IAM Integration**
  IAM roles and service accounts control access to resources, ensuring that only authorized services and users can interact with the cluster and its workloads.

- **Integration with CI/CD**

After Docker images are built and pushed to DockerHub by Jenkins, they are deployed to EKS using Helm in the final stage of the pipeline, achieving automated deployment to Kubernetes.

Benefits of Using AWS EKS:

- Fully managed Kubernetes environment with automatic control plane management
- Seamless integration with AWS services like EC2, IAM, VPC.
- Scalable and resilient architecture for running microservices
- Supports rolling updates and zero-downtime deployments

Enables secure and compliant containerized application hosting

4.2 Software Requirement

Jenkins in the CI/CD Pipeline

Jenkins is used in this project as the primary automation server to implement a robust CI/CD (Continuous Integration and Continuous Deployment) pipeline. It orchestrates the entire software delivery process—from source code integration to security scanning, containerization, and deployment—ensuring rapid, reliable, and secure delivery of Python-based microservices.

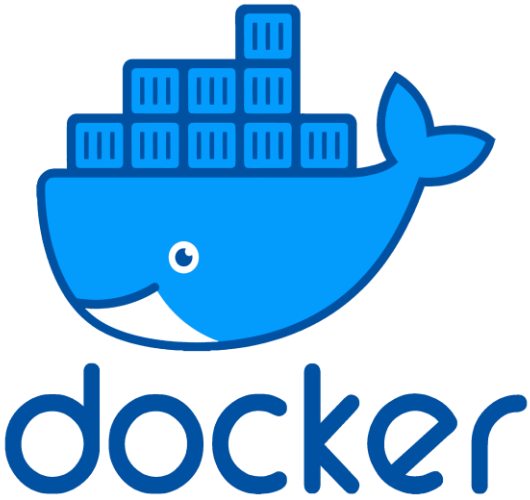
Key Benefits of Using Jenkins:

- Automation of repetitive build, test, and deployment tasks
- Integration with tools like GitHub, SonarQube, Docker, Trivy, and Docker Scout
- Scalability through plugins and custom pipelines
- Centralized control over the software lifecycle

CI/CD Pipeline Stages in Jenkins:

1. Clean Workspace
   Ensures that the Jenkins workspace is cleared before each pipeline run to avoid conflicts from previous builds.
2. Git Checkout
   Pulls the latest code from the project's GitHub repository (main branch). This ensures the pipeline always runs on the most recent commit.
3. SonarQube Analysis
   Performs static code analysis using SonarQube to check for code quality issues, bugs, and security vulnerabilities early in the development cycle.

4. Quality Gate Evaluation
   Waits for the SonarQube Quality Gate result. The build proceeds only if the code meets predefined quality and security standards, enforcing a shift-left security approach.
5. OWASP Dependency-Check (FS Scan)
   Runs a scan to identify known vulnerabilities in third-party dependencies using the OWASP Dependency-Check plugin. It also generates security reports for visibility.
6. Trivy File System Scan
   Uses Trivy, a container security scanner, to analyze the source code and filesystem for vulnerabilities, secrets, and misconfigurations before containerization.
7. Docker Image Build
   Builds a Docker image of the application locally using a Dockerfile, preparing it for deployment in any containerized environment.
8. Tag and Push to DockerHub
   Tags the newly built image and pushes it to a public or private DockerHub repository, making it available for deployment or sharing.
9. Docker Scout Image Analysis
   Runs Docker Scout to analyze the pushed image for CVEs (Common Vulnerabilities and Exposures) and security recommendations, enhancing post-build visibility.
10. Deploy to Container
    Deploys the Docker container on the local machine (or server) by running it with the necessary port mapping, making the application live and accessible



Docker is used in this project to containerize Python-based microservices and essential DevSecOps tools, enabling consistent and reliable deployment across different environments. It plays a vital role in streamlining the development-to-deployment lifecycle by packaging applications and their dependencies into lightweight, portable containers.

Purpose and Role of Docker:

- **Containerization of Microservices**
  Each application component is packaged into its own container, allowing services to run independently and avoiding conflicts between dependencies.

- **Efficient Resource Utilization**
  Docker containers are lightweight and share the host OS kernel, making them more resource-efficient than traditional virtual machines.

- **Environment Consistency**
  Whether running locally, on EC2, or inside a Kubernetes cluster, Docker ensures that the application behaves the same everywhere.

- **Support for CI/CD Automation**
  Integrated with the Jenkins pipeline, Docker automates the build and deployment process. It ensures that once code is committed to GitHub, a container image is built, tested, scanned, and deployed without manual intervention.

- **Container Security Scanning**
  The project uses tools like **Trivy** and **Docker Scout** to scan Docker images for vulnerabilities and provide insights into security risks. This helps in identifying and mitigating issues early in the pipeline.

- **Image Management and Distribution**
  Built images are tagged and pushed to **DockerHub**, making them accessible for deployment or reuse. This versioning also aids in rollback and traceability.

- **Simplified Deployment**
  Docker makes it easy to run services on any system with the Docker engine, including local machines, cloud instances (like EC2), or within orchestration platforms like Kubernetes (EKS).

Trivy is an open-source vulnerability scanner used in this project as a critical security tool within the CI/CD pipeline. It helps identify security issues early in the development lifecycle, aligning with the "shift-left" security approach in DevSecOps.

Purpose of Using Trivy:

- To scan application files and Docker images for known vulnerabilities
- To detect misconfigurations, secrets, and exposed credentials
- To provide actionable security reports before deployment
- To ensure that only secure and compliant images are pushed to DockerHub

How Trivy Is Used in the Project:

- File System Scan (Trivy FS)
  During the pipeline execution, Trivy scans the entire project directory (including source code and configuration files) to detect vulnerabilities and secrets that might be accidentally included in the codebase.

- Docker Image Scan (Optional)
  Trivy can also scan built Docker images before pushing them to DockerHub, identifying security flaws such as outdated packages, vulnerable libraries, and configuration issues inside the image.

- Automated Integration with Jenkins
  Trivy is integrated into the Jenkins pipeline, ensuring scans are executed automatically during every build. The results are saved in a report file (e.g., trivy.txt) for later review.

- Lightweight and Fast
  Trivy is chosen for its speed and simplicity—it does not require heavy setup or configurations, making it ideal for quick and continuous scanning in development workflows.

Benefits of Using Trivy:

- Enhances security posture by catching vulnerabilities early
- Reduces manual code review effort for security checks
- Prevents insecure images from being deployed to production
- Supports continuous security validation without disrupting the pipeline

SonarQube is integrated into this project's DevSecOps pipeline to perform static code analysis and enforce code quality and security standards during the development process. It helps identify bugs, vulnerabilities, code smells, and security hotspots in the Python-based microservices.

Purpose of Using SonarQube:

- To analyze source code for bugs, security flaws, and technical debt
- To maintain code quality and reliability
- To detect common security vulnerabilities like SQL injection, hardcoded credentials, and unvalidated inputs
- To enforce customizable coding standards through Quality Gates

How SonarQube Is Used in the Project:

- **Integration with Jenkins Pipeline**
  SonarQube is integrated into the CI/CD pipeline using Jenkins. After the code is pulled from GitHub, it is automatically analyzed using the SonarQube scanner before proceeding to the next stages.

- **Project Analysis**
  Each microservice or application is treated as a separate project in SonarQube, where metrics such as code coverage, duplication, complexity, and security issues are tracked.

- **Quality Gate Enforcement**
  The pipeline includes a Quality Gate check that evaluates the code against predefined thresholds (e.g., zero critical vulnerabilities or bugs). The build proceeds only if the project passes the Quality Gate, ensuring only secure and clean code gets deployed.

- **Custom Dashboards and Reports**
  SonarQube provides a user-friendly web interface where developers can view real-time dashboards and detailed reports on code health, helping them continuously improve code quality.

Benefits of Using SonarQube:

- Promotes clean code practices across all microservices
- Detects and prevents critical vulnerabilities before deployment
- Improves team productivity by highlighting maintainability issues
- Helps in continuous improvement with historical tracking and trend analysis

Prometheus is used in this project as a monitoring and alerting tool for tracking the performance and health of microservices and infrastructure components deployed via Docker and Kubernetes. It plays a crucial role in ensuring observability and real-time metrics collection, which are essential in any DevSecOps-enabled environment.

Purpose of Using Prometheus:

- To collect time-series metrics from applications, containers, and the host system
- To monitor resource usage, such as CPU, memory, disk I/O, and network
- To support real-time analysis and troubleshooting
- To enable integration with Grafana for visualization

How Prometheus Is Used in the Project

Metrics Collection

Prometheus scrapes metrics from exporters (like Node Exporter for system-level metrics or custom metrics from Python applications) at regular intervals.

- Monitoring Containers and Services
  Prometheus monitors Docker containers and services running inside the EC2 instance or Kubernetes cluster. It provides insights into resource usage, uptime, and service responsiveness.

- Alerting Capability (Optional)
  Though not configured in this basic setup, Prometheus supports alerting rules that can trigger notifications when metrics cross predefined thresholds (e.g., high CPU usage or container crashes).

- Integration with Grafana
  Prometheus serves as the data source for Grafana, which is used to create rich, interactive dashboards that visualize the collected metrics in real-time.

Key Benefits of Using Prometheus:

- Provides real-time visibility into application and system performance
- Helps identify performance bottlenecks and unusual behaviors
- Enables proactive monitoring rather than reactive troubleshooting
- Supports scalability by monitoring multiple containers and services

Grafana is used in this project as the visualization and dashboarding tool to monitor the health and performance of microservices, containers, and infrastructure components. It works in conjunction with Prometheus, which acts as the metrics data source.

Purpose of Using Grafana:

- To visualize real-time metrics collected from Prometheus
- To create interactive dashboards for monitoring system health
- To help developers and operators identify issues quickly
- To enhance observability and support better decision-making

How Grafana Is Used in the Project:

- **Integration with Prometheus**
  Grafana is configured to use Prometheus as its primary data source. All metrics scraped by Prometheus (e.g., CPU usage, memory consumption, container uptime) are available for visualization in Grafana.

- **Dashboard Creation**
  Custom dashboards are created to monitor key performance indicators (KPIs) such as:
  - Application response time
  - Resource utilization (CPU, RAM, Disk)
  - Container and microservice health
  - Network traffic and system load
  -

- **Live Monitoring**
  Grafana dashboards display data in real time, allowing developers and administrators to observe system behavior as it happens. This helps in early detection of anomalies or performance degradation.

User-Friendly Interface

Grafana's web-based UI is highly customizable, allowing users to add graphs, gauges, tables, and alerts using a drag-and-drop interface with minimal configuration**.**

5. Configurations

EC2 CONFIGURATION

1. **Instance Launch:**
   Log in to the AWS Management Console and navigate to the EC2 dashboard.
   Click on "Launch Instance" to start the instance creation process.

2. **Choose an Amazon Machine Image (AMI):**
   Select an appropriate AMI based on your operating system and application requirements.

3. **Choose an Instance Type:**
   Choose the instance type that suits your workload's CPU, memory, and storage needs.

4. **Configure Instance Details:**
   Choose the VPC, subnet, and availability zone where the instance will be launched.
   Configure network settings, including assigning a public IP if required.
   Optionally, configure IAM roles and instance metadata.

5. **Configure Security Group:**
   Create a new security group or select an existing one to control inbound and outbound traffic. Define rules to allow specific protocols, ports, and IP ranges.

6. **Review Instance Launch:**
   Review the instance configuration settings and make any necessary adjustments.

7. **Launch Instance:**
   Choose an existing key pair or create a new one to securely access the instance via SSH or RDP. Launch the instance and wait for it to initialize.

8. **Accessing the Instance:**
   Use SSH (Linux) or RDP (Windows) to connect to the instance using the assigned public IP or DNS. Optionally, use an Elastic IP for a static public IP address.

9. **Instance Configuration:**
   Install and configure software, applications, and services as needed on the instance.
   Set up security updates and patches to keep the instance secure.

10. **Monitoring and Logging:**
    Install CloudWatch agent to collect and send instance-level metrics, logs, and custom data to CloudWatch.

11. Security and IAM:

Regularly review and update security groups, IAM roles, and instance permissions. Consider using instance roles to securely access AWS services from the instance.

12. Instance Termination:

When an instance is no longer needed, terminate it to stop incurring charges.

ELASTIC IP CONFIGURATION

1. **Allocate Elastic IP:**
   Log in to the AWS Management Console, go to the **EC2 dashboard**, and click on **Elastic IPs** under the "Network & Security" section. Click **"Allocate Elastic IP address"** and choose "Amazon's pool of IPv4 addresses." Click **Allocate**.
2. **View Allocated IP:**
   After allocation, note down the **Elastic IP address**. You can use it to associate with any instance within your account in the same region.
3. **Associate Elastic IP:**
   Select the newly allocated Elastic IP and click on **Actions → Associate Elastic IP address**. Choose the target **EC2 instance** and the **private IP** of that instance. Click **Associate**.
4. **Update Security Group (if required):**
   Ensure the EC2 instance's **Security Group** allows the necessary inbound traffic. For example:
   - TCP port 22 for SSH (Linux)
   - TCP port 80/443 for web servers
   - TCP port 3389 for RDP (Windows)
5. **Access the Instance Using EIP:**
   Use the associated Elastic IP to SSH (for Linux) or RDP (for Windows) into the EC2 instance, or access the hosted application directly in a browser using the static IP.
6. **Elastic IP Persistence:**
   Even if the instance is stopped and restarted, the **Elastic IP remains attached**, ensuring a **consistent public IP** for accessibility.
7. **Disassociate or Release Elastic IP (Optional):**
   When the IP is no longer needed, go to **Elastic IPs**, select the address, and choose **Actions → Disassociate Elastic IP**. To avoid unnecessary charges, you can then **Release** the Elastic IP.
8. **Cost Note:**
   AWS does not charge for EIPs **when they are associated with a running instance**. However, idle EIPs (not associated) incur charges, so they should be released if unused.
9. **Project Use Case:**
   In this project, an Elastic IP was associated with a public-facing EC2 instance hosting a Flask-based web application. This allowed **stable, predictable access** to the application across sessions and reboots.

CI/CD Jenkins Pipeline Stages:

1. **Create EKS Cluster**
   Create an Amazon EKS cluster using eksctl with the required configuration (cluster name, region, and Kubernetes version).
2. **Associate IAM OIDC Provider**
   Associate an IAM OpenID Connect (OIDC) provider with the EKS cluster for secure authentication.
3. **Create EC2 Keypair**
   Generate an EC2 keypair to enable SSH access to worker nodes.
4. **Create EKS Node Group with Add-ons**
   Add a managed node group to the EKS cluster and install essential add-ons like VPC CNI, kube-proxy, and CoreDNS.
5. **Clean Workspace**
   Clear old files and artifacts from the Jenkins workspace before starting the build process.
6. **Git Checkout**
   Pull the latest application source code from the GitHub repository.
7. **SonarQube Analysis**
   Run static code analysis using SonarQube to check for code quality and vulnerabilities.
8. **Quality Gate**
   Verify that the code passes SonarQube quality gate standards before proceeding.
9. **OWASP FS Scan**
   Perform an OWASP security scan to detect vulnerabilities in files.
10. **Trivy File Scan**
    Scan source files for vulnerabilities using Trivy.
11. **Build Docker Image**
    Build a Docker image from the application source code.
12. **Tag & Push to DockerHub**
    Tag the Docker image and push it to the DockerHub registry.
13. **Docker Scout Image**
    Scan the Docker image with Docker Scout for vulnerabilities and compliance issues.
14. **Deploy to Container**
    Deploy the Docker image from DockerHub to the EKS cluster using Kubernetes manifests (deployment.yaml, service.yaml).

SONARQUBE CONFIGURATION

1. **Launch EC2 Instance:**
   Launch an EC2 instance (Ubuntu) with at least 2 vCPUs and 4 GB RAM.
2. **Install Docker and Docker Compose:**
   Update the system and install Docker and Docker Compose using system package manager.
3. **Create Docker Compose File:**
   Create a docker-compose.yml file defining services for SonarQube and PostgreSQL.
4. **Define SonarQube and PostgreSQL Containers:**
   Use the sonarqube:community image for SonarQube and postgres:13 for the database with appropriate environment variables.
5. **Start SonarQube Services:**
   Run docker-compose up -d to launch both containers in detached mode.
6. **Verify Container Status:**
   Check running containers using docker ps to ensure both services are active.
7. **Access SonarQube Web Interface:**
   Open a browser and go to http://<Elastic-IP>:9000 to access the SonarQube dashboard.
8. **Login with Default Credentials:**
   Use username admin and password admin for initial login and then change the password.
9. **Generate Authentication Token:**
   In SonarQube dashboard, navigate to My Account → Security and generate a token for Jenkins integration.
10. **Integrate SonarQube with Jenkins:**
    In Jenkins, go to **Manage Jenkins → Configure System**, and add the SonarQube server URL and token.

CONFIGURATION IAM

1. **Access Control Basics:**
   IAM is AWS's identity management service that controls access to AWS resources. Start by understanding IAM concepts like users, groups, roles, policies, and permissions.

2. **Least Privilege Principle:**
   Apply the principle of least privilege when assigning permissions.
   Only grant users or roles the minimum permissions necessary to perform their tasks.

3. **User and Group Management:**
   Create individual IAM users for human users and assign them to appropriate groups. Create IAM groups based on roles or responsibilities (e.g., administrators, analysts).

4. **Roles and Permissions:**
   Create IAM roles with specific permissions that Lambda functions or services need. Define trust relationships to specify which entities (e.g., Lambda, EC2) can assume the role.

5. **Cross-Account Access:**
   Use IAM roles for cross-account access to allow trusted AWS accounts to assume roles in your account. Helps centralize management and maintain separation of concerns.

6. **Temporary Security Credentials:**
   For automated prevention, use roles to grant Lambda functions temporary security credentials. Enables Lambda functions to interact with AWS services securely without storing long-term credentials.

7. **Condition Keys:**
   Use condition keys in IAM policies to add context to permissions.
   For example, restrict actions based on time of day or IP address range.

8. **Policy Structure:**
   Create custom IAM policies that define the actions and resources a user, group, or role can access. Attach policies to entities to grant appropriate permissions.

9. **Managed Policies:**
   Use AWS managed policies for common use cases like AdministratorAccess or ReadOnlyAccess. Reduces the need to create custom policies from scratch.

10. **Resource-Based Policies:**
    Some services, like S3 and Lambda, use resource-based policies.
    These policies control who can access specific resources within the service.

11. **IAM Access Analyzer:**
    Utilize IAM Access Analyzer to identify overly permissive permissions.
    Helps you maintain a secure permission structure.

12. **Policy Simulation:**
    Test IAM policies using policy simulation to understand how permissions are applied. Helps ensure policies work as intended.

13. **Regular Review and Auditing:**
    Conduct periodic reviews of IAM policies and permissions.
    Remove unnecessary permissions and roles.

14. **Access Key Management:**
    For programmatic access, manage access keys securely.
    Rotate keys regularly and avoid hardcoding them in code.

GRAFANA CONFIGURATION

1. **Launch EC2 Instance:**
   Launch an Ubuntu EC2 instance with internet access. Open port **3000** in the security group to allow access to the Grafana web interface.
2. **Update System Packages:**
   SSH into the instance and run system update commands to ensure all packages are up to date.
3. **Add Grafana APT Repository:**
   Add Grafana's official Debian/Ubuntu package repository to the system sources list.
4. **Import Grafana GPG Key:**
   Download and add the Grafana GPG key to verify package authenticity.
5. **Install Grafana:**
   Use the package manager to install Grafana from the added repository.
6. **Start and Enable Grafana Service:**
   Start the Grafana server and enable it to launch on system boot using systemctl commands.
7. **Verify Grafana Service:**
   Check the status of the Grafana service to ensure it's running correctly.
8. **Access Grafana Web Interface:**
   Open a web browser and go to http://<Elastic-IP>:3000 to access the Grafana dashboard.
9. **Login to Grafana:**
   Use default credentials (admin/admin) to log in and reset the password when prompted.
10. **Add Data Source:**
    Navigate to **Configuration → Data Sources**, select the appropriate data source (e.g., Prometheus), enter the necessary connection details, and save.
11. **Import or Create Dashboards:**
    Go to **Dashboards → Import**, use a Grafana dashboard ID or upload a JSON file to visualize metrics and data.

PROMETHEUS CONFIGURATION

1. **Download and Extract**
   Download the Prometheus package from the official website and extract it on the server.
2. **Basic Setup**
   Navigate to the extracted directory and locate the prometheus.yml file. This file is used to define jobs and targets for monitoring.
3. **Configure Targets**
   Edit prometheus.yml to add monitoring targets (such as the application, Node Exporter, etc.).
4. **Run Prometheus**
   Start Prometheus by executing the Prometheus binary with the configuration file. Prometheus starts on default port 9090.
5. **Access Dashboard**
   Open a browser and go to http://<your-ip>:9090 to access the Prometheus web UI.
6. **Verify Metrics**
   Confirm that Prometheus is scraping metrics from defined targets and visualizing time-series data.
7. **Integration (Optional)**
   Integrate Prometheus with Grafana for better visualization and alerting.

6.PIPELINE

6.1 CI CD Pipeline

```
pipeline {
    agent any
    environment {
        SCANNER_HOME = tool 'sonar-scanner'
    }
    stages {

        stage("Step 01: Create EKS Cluster") {
            steps {
                sh '''
                eksctl create cluster --name=cluster1 \
                                --region=us-east-1 \
                                --zones=us-east-1a,us-east-1b \
                                --without-nodegroup
                '''
            }
        }

        stage("Step 02: Associate IAM OIDC Provider") {
            steps {
                sh '''
                eksctl utils associate-iam-oidc-provider \
                      --region us-east-1 \
                      --cluster cluster1 \
                      --approve
                '''
            }
        }

        stage("Step 03: Create EC2 Keypair Note") {
            steps {
                echo 'Ensure EC2 Keypair "awskey1" exists manually in us-east-1 region.'
            }
        }

        stage("Step 04: Create EKS Node Group with Add-ons") {
            steps {
                sh '''
                eksctl create nodegroup --cluster=cluster1 \
```

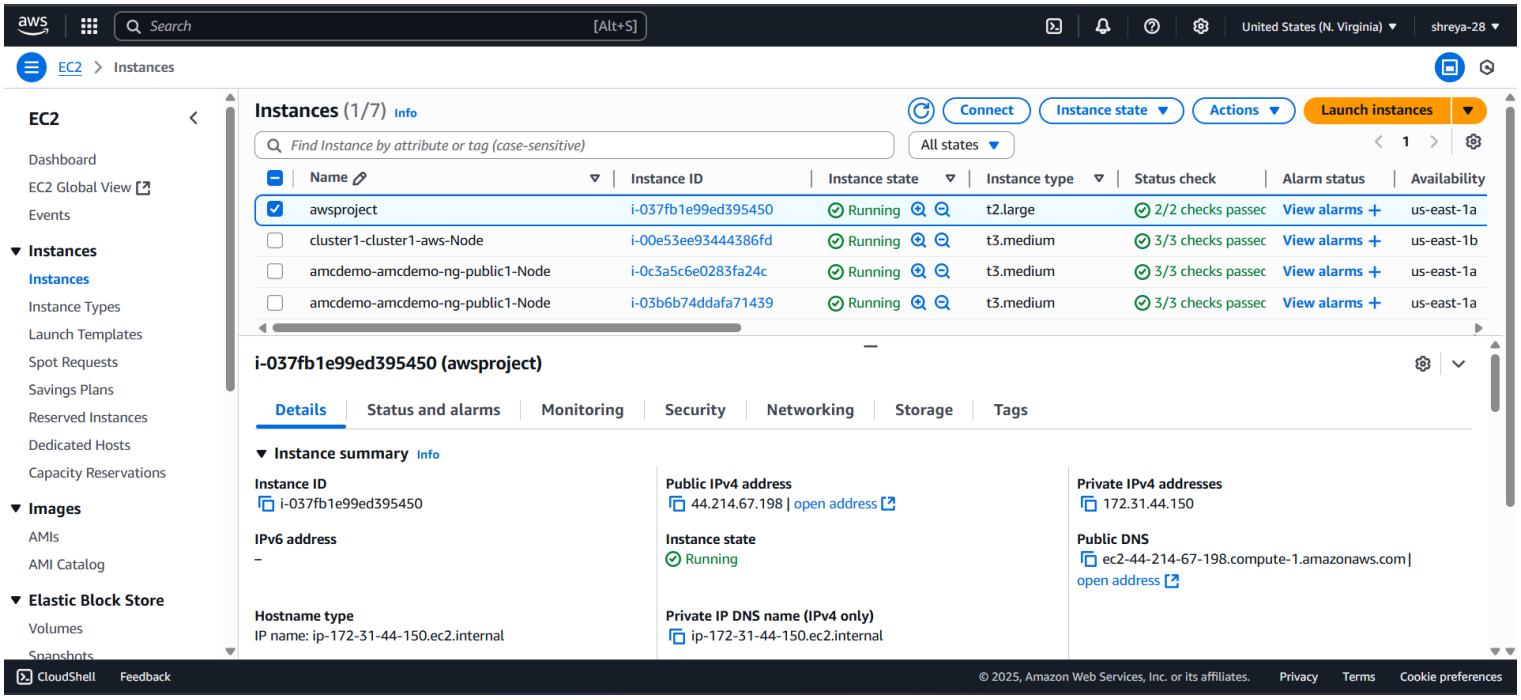
```
                --region=us-east-1 \
                --name=cluster1-aws \
                --node-type=t3.medium \
                --nodes=2 \
                --nodes-min=2 \
                --nodes-max=4 \
                --node-volume-size=20 \
                --ssh-access \
                --ssh-public-key=awskey1 \
                --managed \
                --asg-access \
                --external-dns-access \
                --full-ecr-access \
                --appmesh-access \
                --alb-ingress-access
            '''
        }
    }

    // ------------------ EXISTING PIPELINE ------------------
    stage ("Clean workspace") {
        steps {
            cleanWs()
        }
    }

    stage ("Git checkout") {
        steps {
            git branch: 'main', url: 'https://github.com/yeshwanthlm/background-remover-python-app.git'
        }
    }

    stage("SonarQube Analysis") {
        steps {
            withSonarQubeEnv('sonar-server') {
                sh ''' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=background-remover-python-
app \
                -Dsonar.projectKey=background-remover-python-app '''
            }
        }
    }

    stage("Quality Gate") {
        steps {
            script {
                waitForQualityGate abortPipeline: false, credentialsId: 'Sonar-token'
            }
```

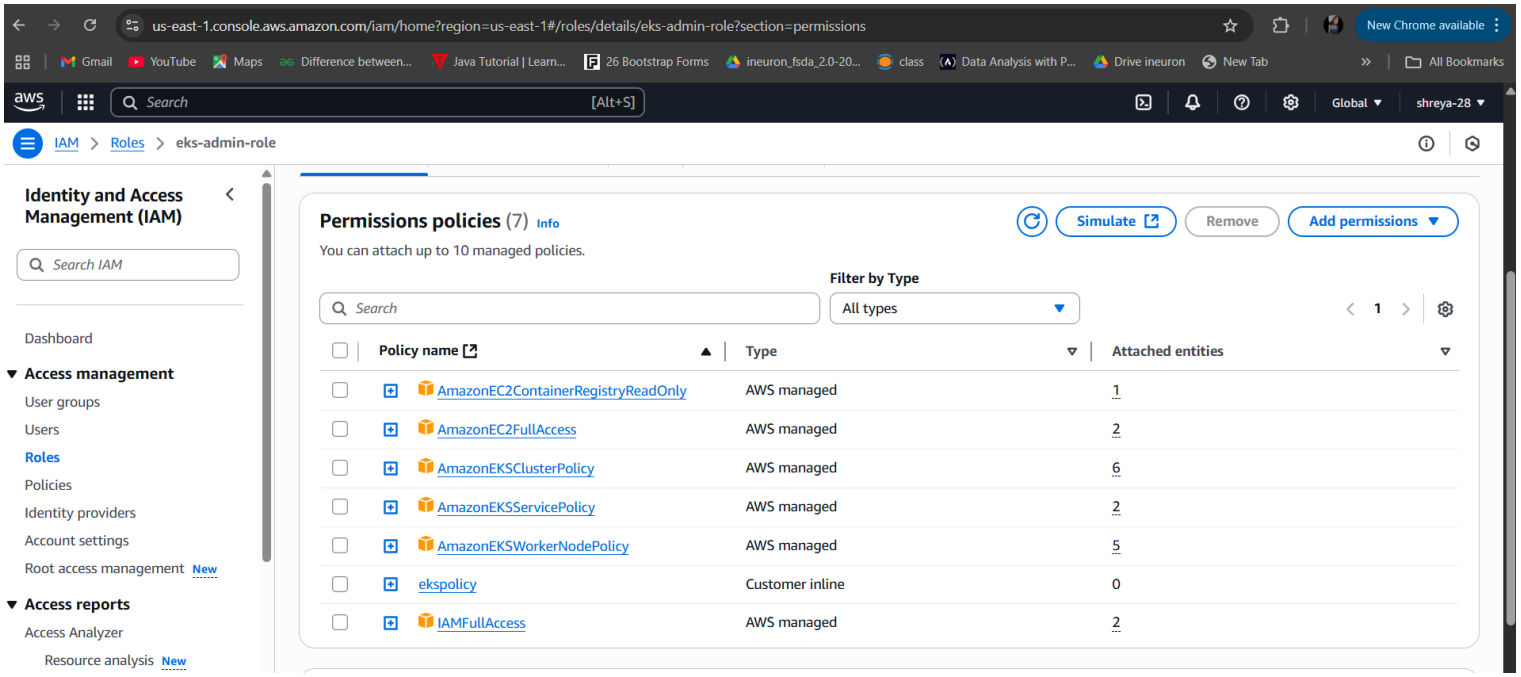
```
        }
    }

    stage('OWASP FS Scan') {
        steps {
            dependencyCheck additionalArguments: '--scan ./ --disableYarnAudit --disableNodeAudit',
odcInstallation: 'DP-Check'
            dependencyCheckPublisher pattern: '/dependency-check-report.xml'
        }
    }

    stage ("Trivy File Scan") {
        steps {
            sh "trivy fs . > trivy.txt"
        }
    }

    stage ("Build Docker Image") {
        steps {
            sh "docker build -t background-remover-python-app ."
        }
    }

    stage ("Tag & Push to DockerHub") {
        steps {
            script {
                withDockerRegistry(credentialsId: 'docker') {
                    sh "docker tag background-remover-python-app shre28/background-remover-python-app:latest"
                    sh "docker push shre28/background-remover-python-app:latest"
                }
            }
        }
    }

    stage('Docker Scout Image') {
        steps {
            script {
              withDockerRegistry(credentialsId: 'docker', toolName: 'docker') {
                 sh 'docker-scout quickview shre28/background-remover-python-app:latest'
                 sh 'docker-scout cves shre28/background-remover-python-app:latest'
                 sh 'docker-scout recommendations shre28/background-remover-python-app:latest'
              }
            }
        }
    }

    stage ("Deploy to Container") {
```

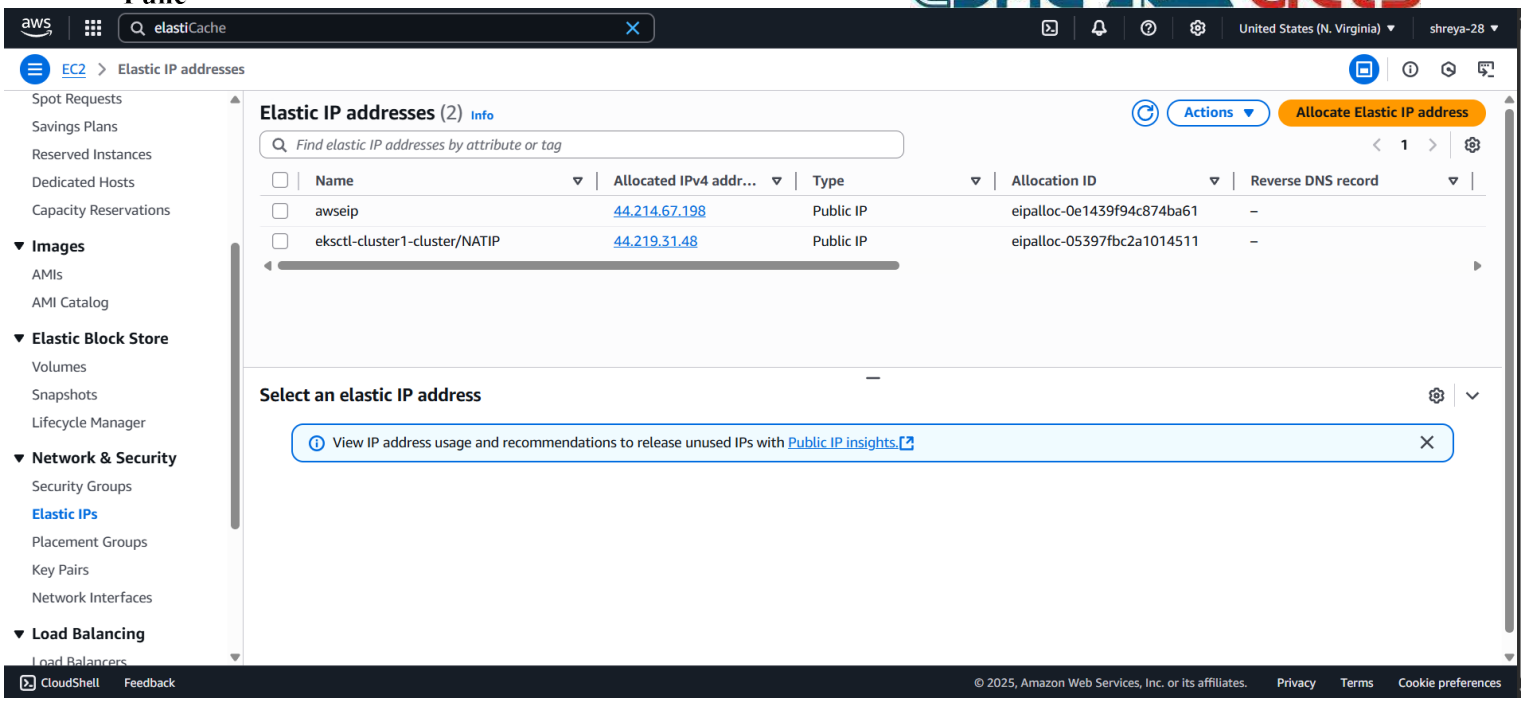
```
        steps {
            sh 'docker run -d --name background-remover-python-app -p 5100:5100 shre28/background-remover-python-app:latest'
        }
    }
  }
}
```

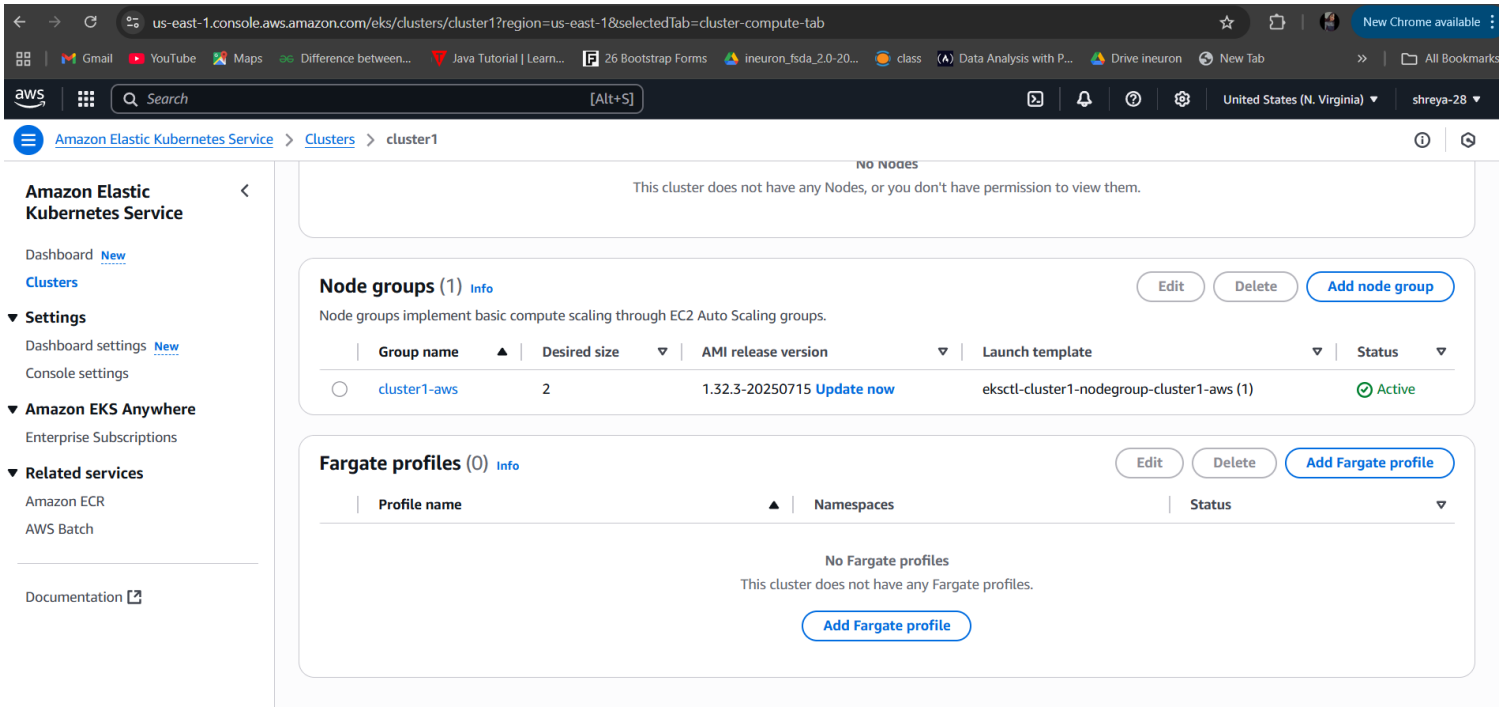
7.RESULTS

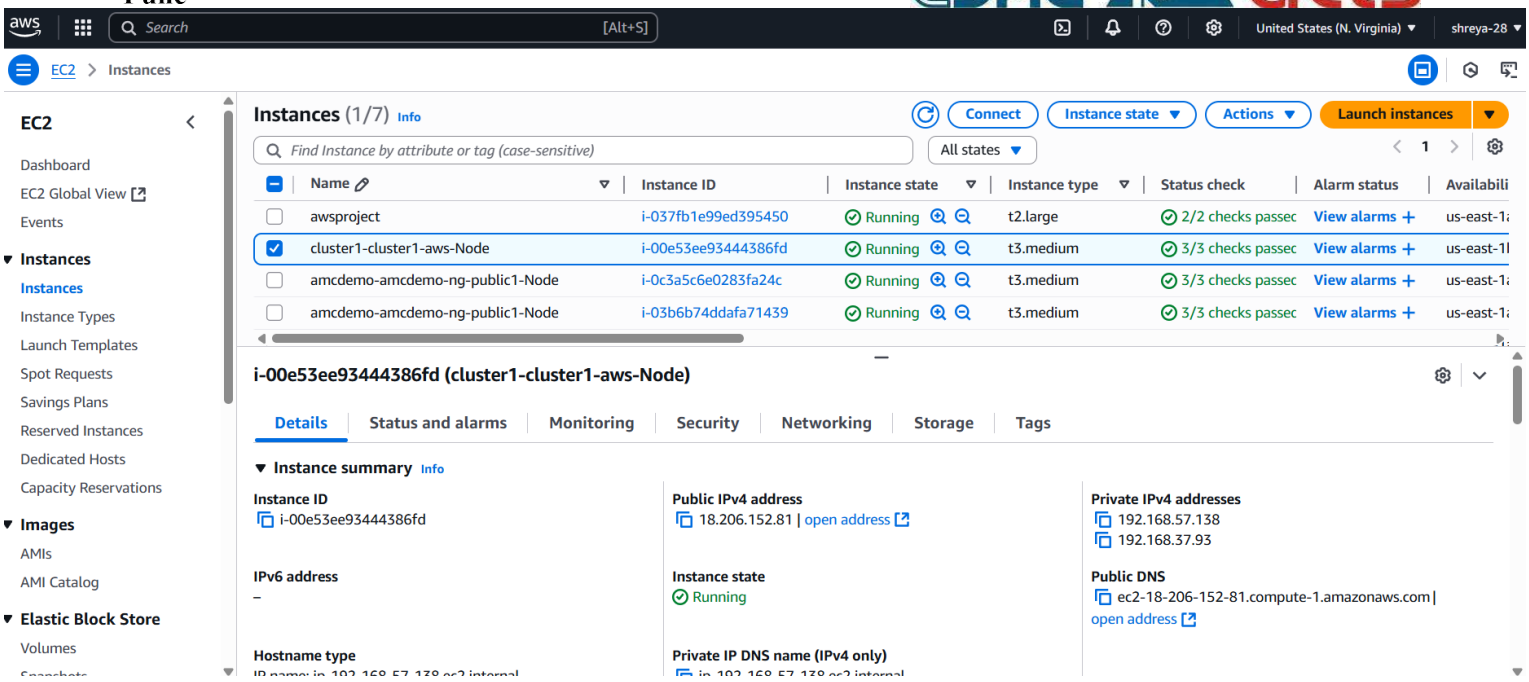
7.1 Results

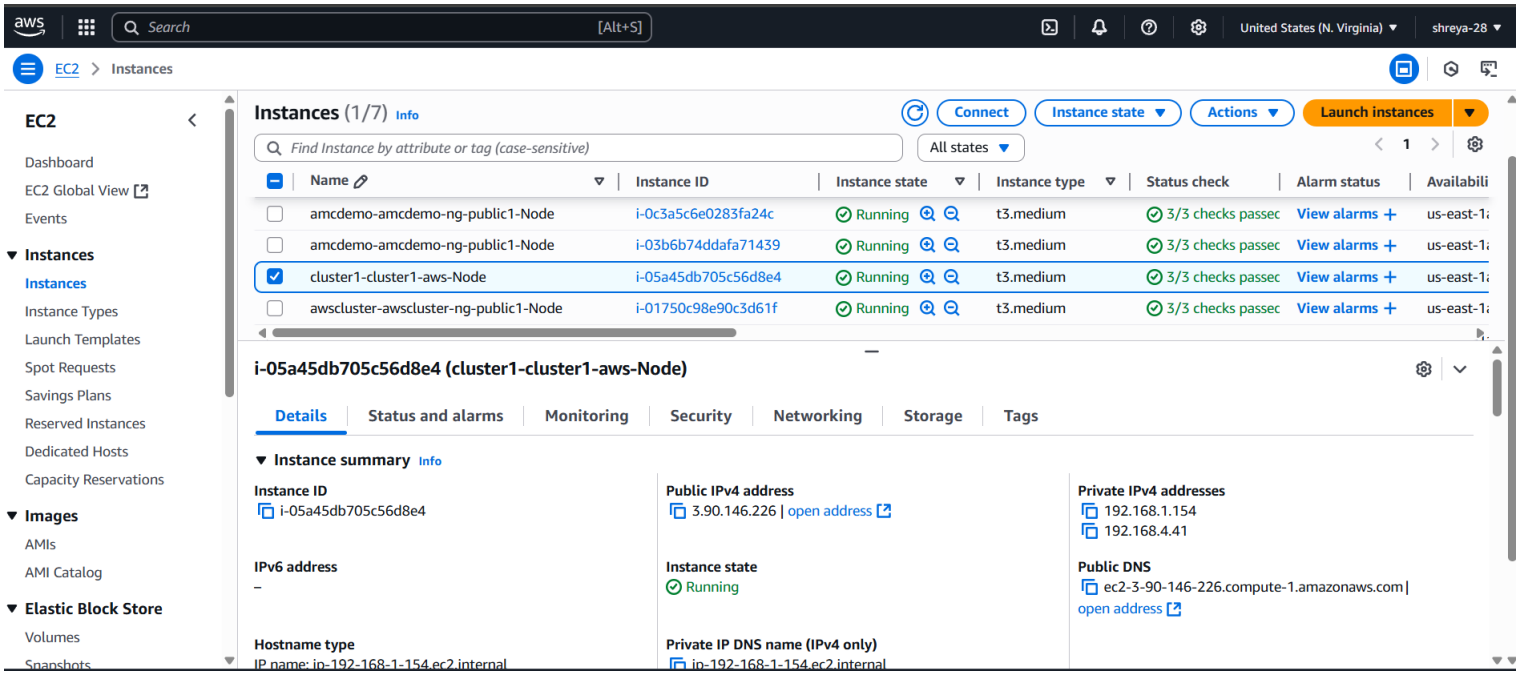


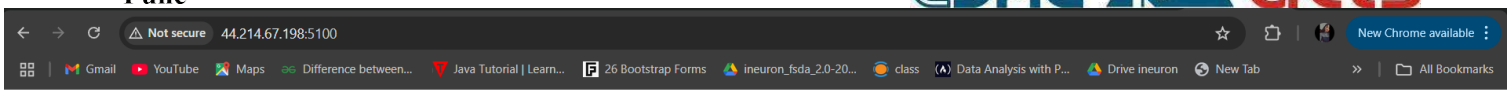



elastiCache
United States (N. Virginia)
shreya-28
EC2 > Elastic IP addresses
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Snapshots
Lifecycle Manager
Network & Security
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
Load Balancing
Load Balancers
Elastic IP addresses (2) Info
Actions
Allocate Elastic IP address
Find elastic IP addresses by attribute or tag
Name
Allocated IPv4 addr...
Type
Allocation ID
Reverse DNS record
awseip
44.214.67.198
Public IP
eipalloc-0e1439f94c874ba61
eksctl-cluster1-cluster/NATIP
44.219.31.48
Public IP
eipalloc-05397fbc2a1014511
Select an elastic IP address
View IP address usage and recommendations to release unused IPs with Public IP insights.
CloudShell
Feedback
© 2025, Amazon Web Services, Inc. or its affiliates.
Privacy
Terms
Cookie preferences


us-east-1.console.aws.amazon.com/eks/clusters/cluster1?region=us-east-1&selectedTab=cluster-compute-tab
New Chrome available
Search
[Alt+S]
United States (N. Virginia)
shreya-28
Amazon Elastic Kubernetes Service > Clusters > cluster1
Amazon Elastic Kubernetes Service
Dashboard New
Clusters
Settings
Dashboard settings New
Console settings
Amazon EKS Anywhere
Enterprise Subscriptions
Related services
Amazon ECR
AWS Batch
Documentation
No Nodes
This cluster does not have any Nodes, or you don't have permission to view them.
Node groups (1) Info
Edit
Delete
Add node group
Node groups implement basic compute scaling through EC2 Auto Scaling groups.
Group name
Desired size
AMI release version
Launch template
Status
cluster1-aws
2
1.32.3-20250715 Update now
eksctl-cluster1-nodegroup-cluster1-aws (1)
Active
Fargate profiles (0) Info
Edit
Delete
Add Fargate profile
Profile name
Namespaces
Status
No Fargate profiles
This cluster does not have any Fargate profiles.
Add Fargate profile


aws
Search
[Alt+S]
United States (N. Virginia)
shreya-28
EC2 > Instances
EC2
Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Instances (1/7) Info
Connect
Instance state
Actions
Launch instances
Find Instance by attribute or tag (case-sensitive)
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
awsproject i-037fb1e99ed395450 Running t2.large 2/2 checks passec View alarms
cluster1-cluster1-aws-Node i-00e53ee93444386fd Running t3.medium 3/3 checks passec View alarms
amcdemo-amcdemo-ng-public1-Node i-0c3a5c6e0283fa24c Running t3.medium 3/3 checks passec View alarms
amcdemo-amcdemo-ng-public1-Node i-03b6b74ddafa71439 Running t3.medium 3/3 checks passec View alarms
i-00e53ee93444386fd (cluster1-cluster1-aws-Node)
Details
Status and alarms
Monitoring
Security
Networking
Storage
Tags
Instance summary Info
Instance ID
i-00e53ee93444386fd
Public IPv4 address
18.206.152.81 | open address
Private IPv4 addresses
192.168.57.138
192.168.37.93
IPv6 address
–
Instance state
Running
Public DNS
ec2-18-206-152-81.compute-1.amazonaws.com | open address
Hostname type
Private IP DNS name (IPv4 only)

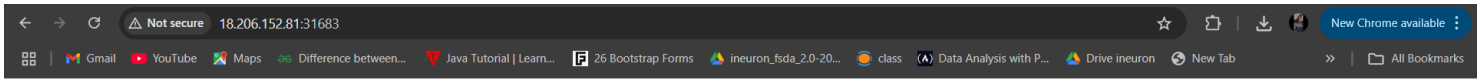

aws
Search
[Alt+S]
United States (N. Virginia)
shreya-28
EC2 > Instances
EC2
Dashboard
EC2 Global View
Events
Instances
Instances
Instance Types
Launch Templates
Spot Requests
Savings Plans
Reserved Instances
Dedicated Hosts
Capacity Reservations
Images
AMIs
AMI Catalog
Elastic Block Store
Volumes
Snapshots
Instances (1/7) Info
Connect
Instance state
Actions
Launch instances
Find Instance by attribute or tag (case-sensitive)
All states
Name
Instance ID
Instance state
Instance type
Status check
Alarm status
amcdemo-amcdemo-ng-public1-Node i-0c3a5c6e0283fa24c Running t3.medium 3/3 checks passec View alarms
amcdemo-amcdemo-ng-public1-Node i-03b6b74ddafa71439 Running t3.medium 3/3 checks passec View alarms
cluster1-cluster1-aws-Node i-05a45db705c56d8e4 Running t3.medium 3/3 checks passec View alarms
awscluster-awscluster-ng-public1-Node i-01750c98e90c3d61f Running t3.medium 3/3 checks passec View alarms
i-05a45db705c56d8e4 (cluster1-cluster1-aws-Node)
Details
Status and alarms
Monitoring
Security
Networking
Storage
Tags
Instance summary Info
Instance ID
i-05a45db705c56d8e4
Public IPv4 address
3.90.146.226 | open address
Private IPv4 addresses
192.168.1.154
192.168.4.41
IPv6 address
–
Instance state
Running
Public DNS
ec2-3-90-146-226.compute-1.amazonaws.com | open address
Hostname type
IP name: ip-192-168-1-154.ec2.internal
Private IP DNS name (IPv4 only)
ip-192-168-1-154.ec2.internal



Upload Image to Remove Background

Choose File No file chosen

Remove Background

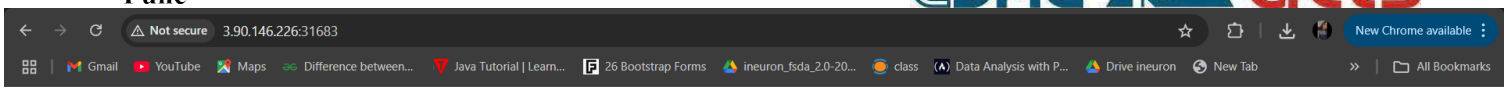


Upload Image to Remove Background

Choose File No file chosen

Remove Background

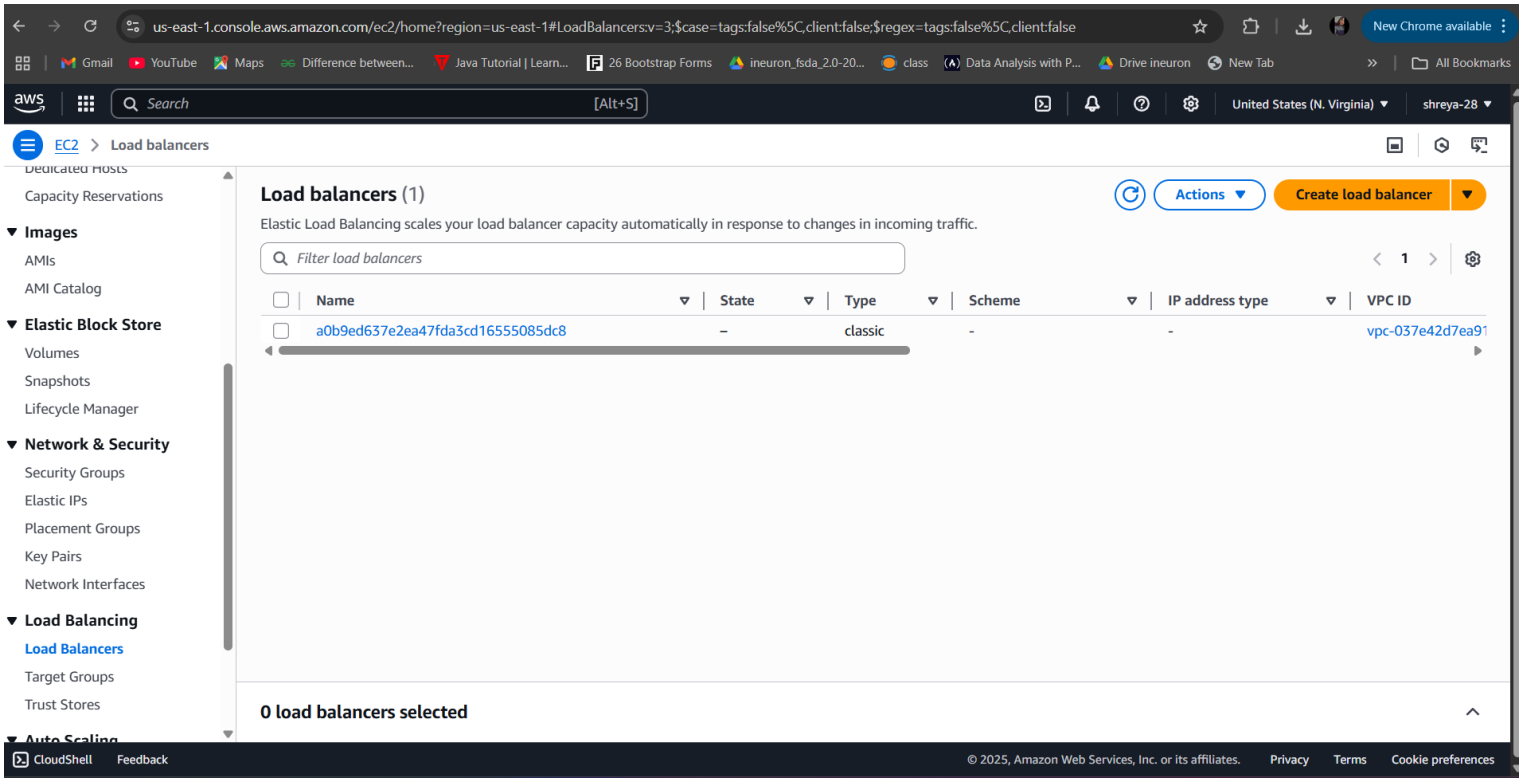
ACTS, Head Quarters,
Pune



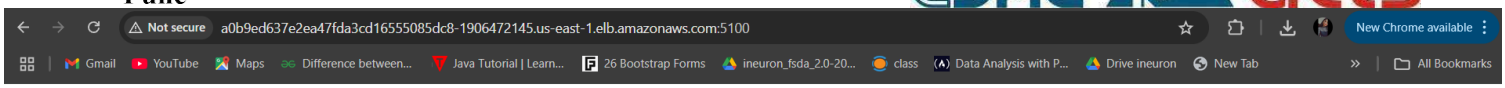
Upload Image to Remove Background

Choose File No file chosen

Remove Background



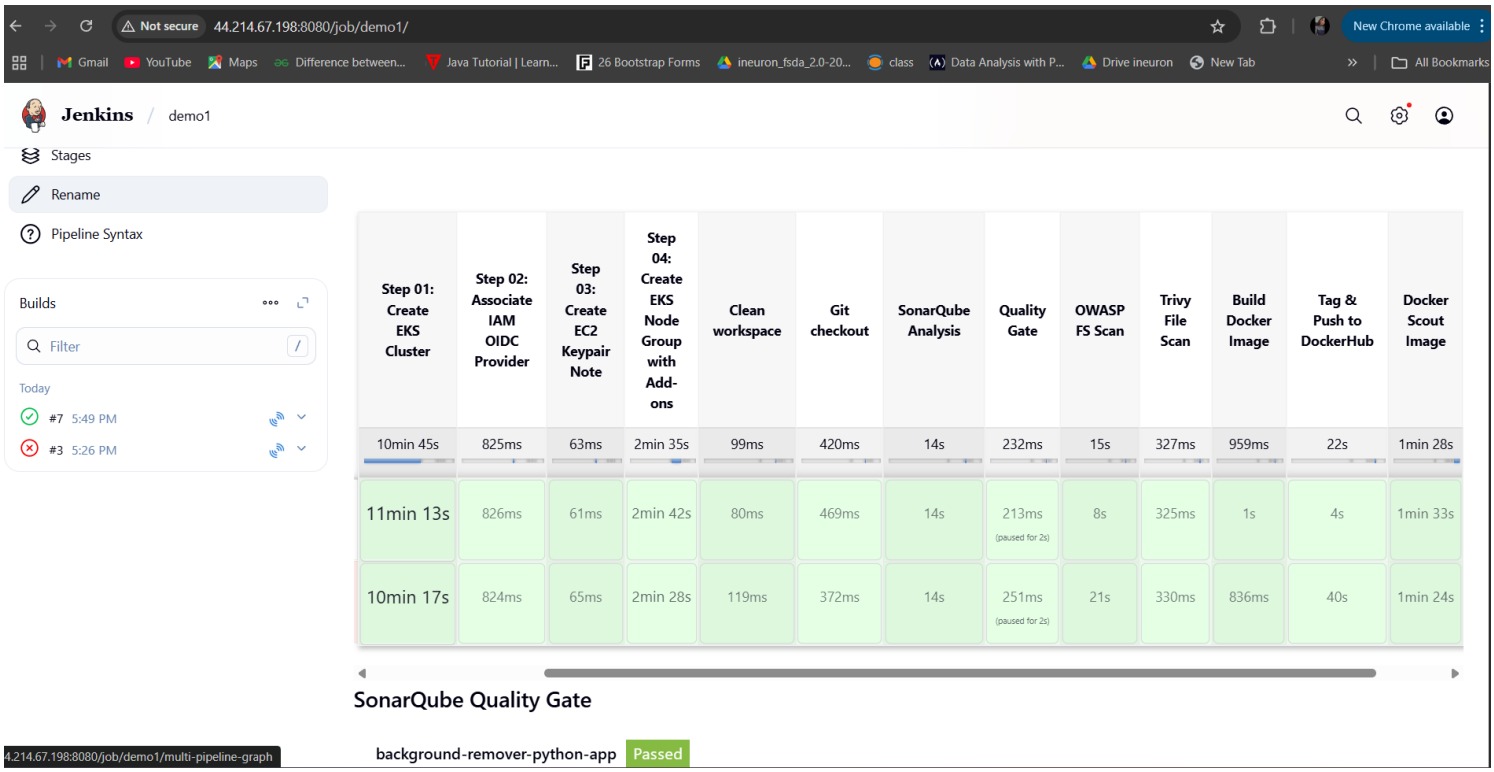
ACTS, Head Quarters,
Pune

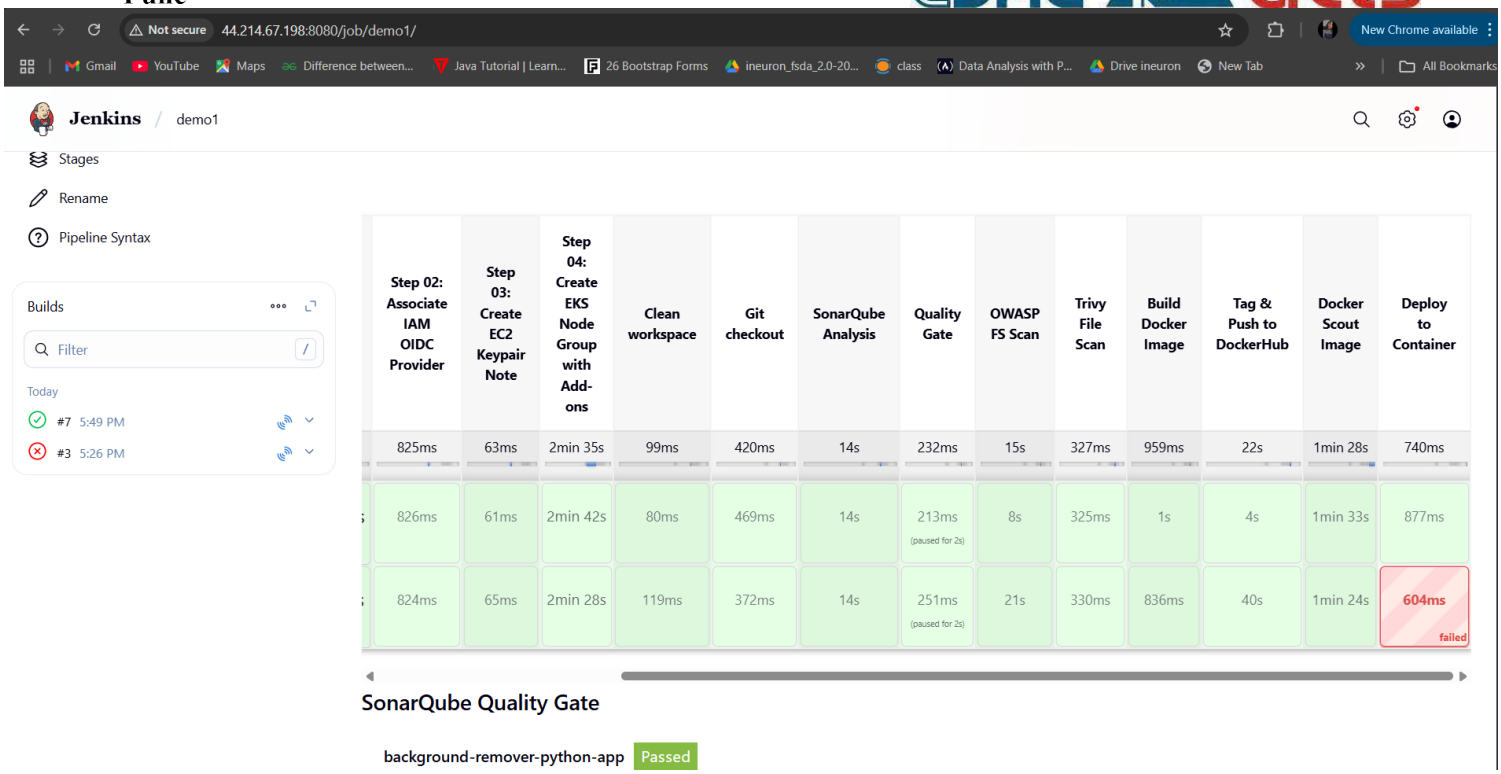


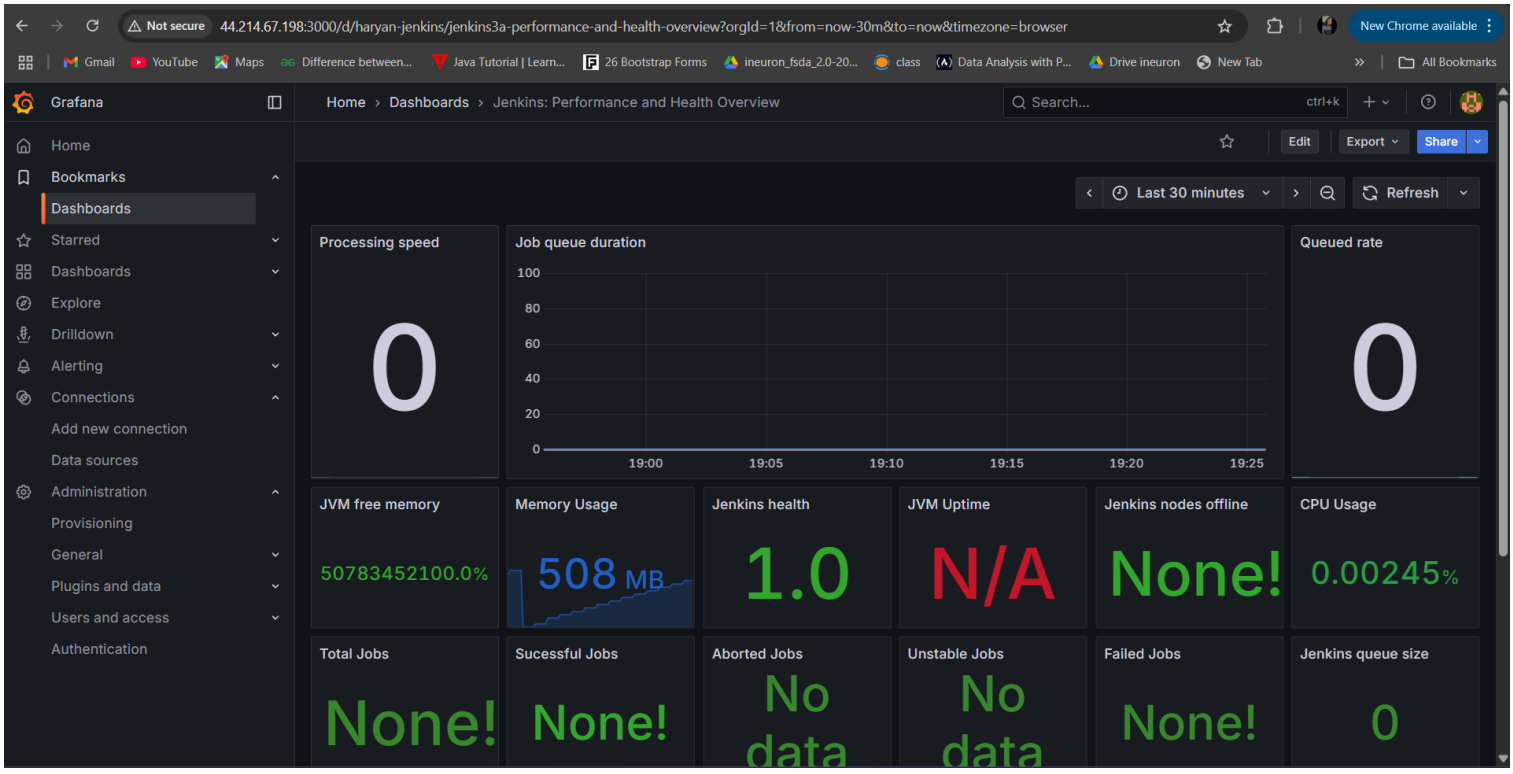
Upload Image to Remove Background

Choose File No file chosen

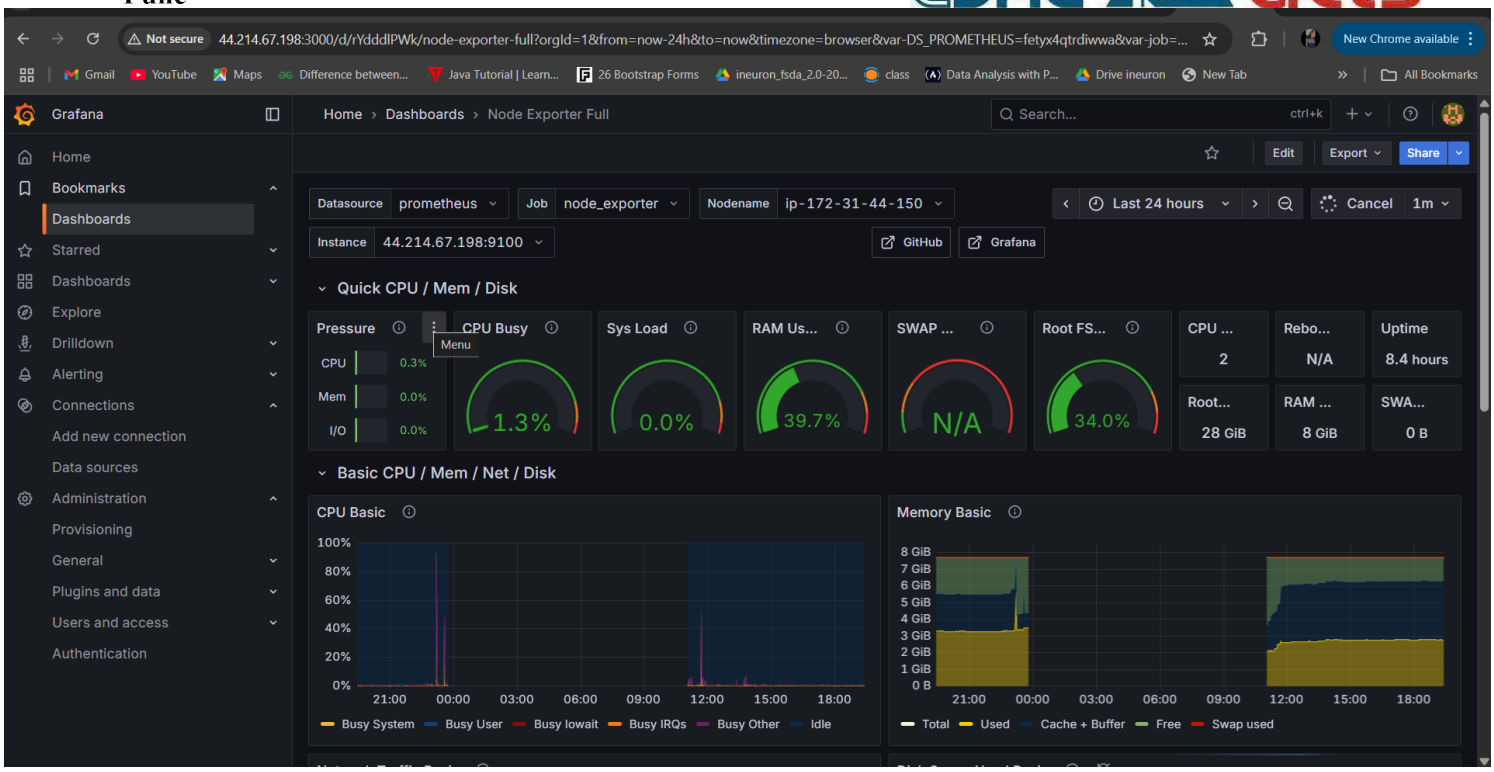
Remove Background

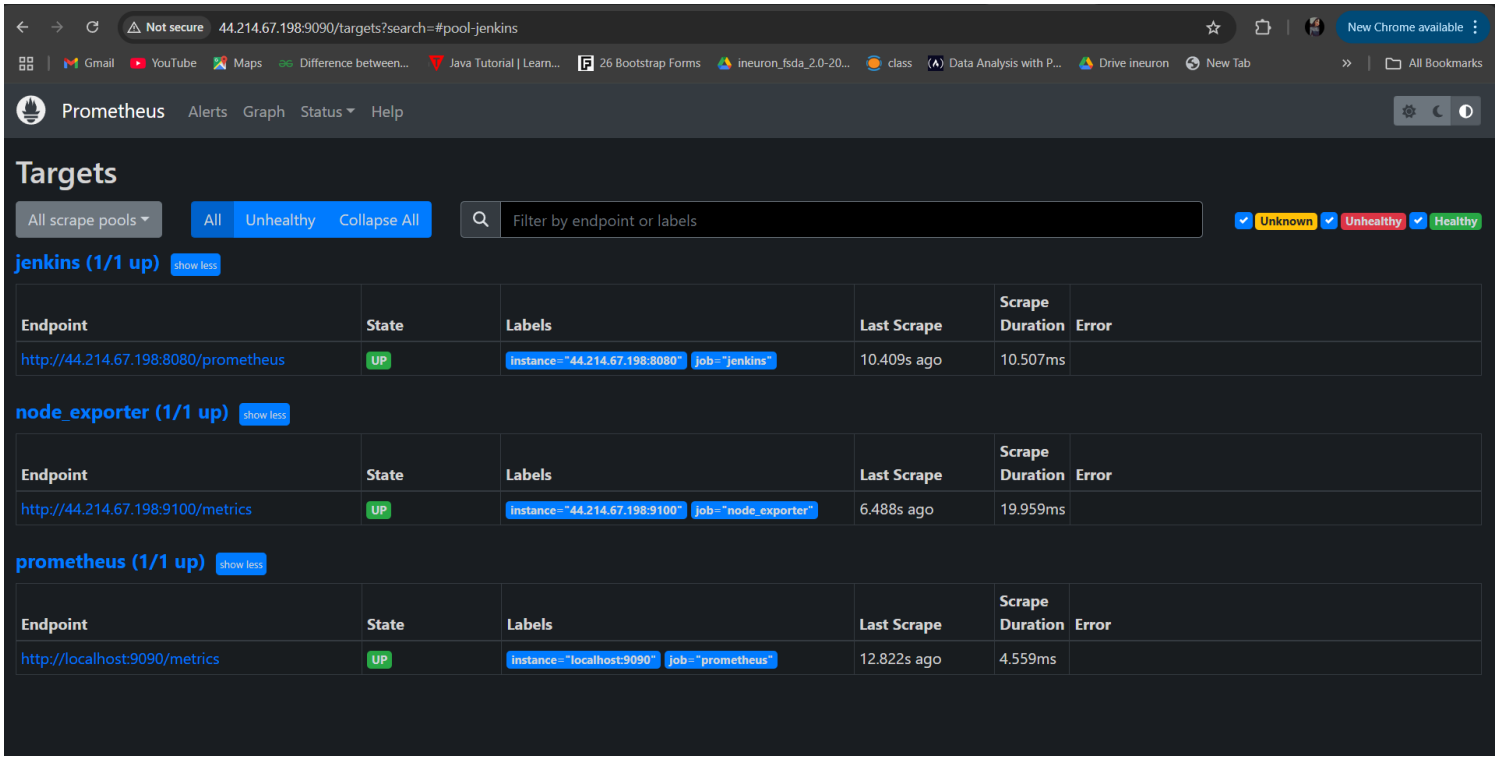



Not secure 44.214.67.198:8080/job/demo1/
New Chrome available
Jenkins / demo1
Stages
Rename
Pipeline Syntax
Builds
Filter
Today
#7 5:49 PM
#3 5:26 PM
Step 02: Associate IAM OIDC Provider
Step 03: Create EC2 Keypair Note
Step 04: Create EKS Node Group with Add-ons
Clean workspace
Git checkout
SonarQube Analysis
Quality Gate
OWASP FS Scan
Trivy File Scan
Build Docker Image
Tag & Push to DockerHub
Docker Scout Image
Deploy to Container
825ms 63ms 2min 35s 99ms 420ms 14s 232ms 15s 327ms 959ms 22s 1min 28s 740ms
826ms 61ms 2min 42s 80ms 469ms 14s 213ms 8s 325ms 1s 4s 1min 33s 877ms
824ms 65ms 2min 28s 119ms 372ms 14s 251ms 21s 330ms 836ms 40s 1min 24s 604ms failed
SonarQube Quality Gate
background-remover-python-app Passed


Not secure 44.214.67.198:3000/d/haryan-jenkins/jenkins3a-performance-and-health-overview?orgId=1&from=now-30m&to=now&timezone=browser
New Chrome available
Grafana
Home › Dashboards › Jenkins: Performance and Health Overview
Search... ctrl+k
Edit Export Share
Last 30 minutes Refresh
Home
Bookmarks
Dashboards
Starred
Dashboards
Explore
Drilldown
Alerting
Connections
Add new connection
Data sources
Administration
Provisioning
General
Plugins and data
Users and access
Authentication
Processing speed
0
Job queue duration
100 80 60 40 20 0
19:00 19:05 19:10 19:15 19:20 19:25
Queued rate
0
JVM free memory
50783452100.0%
Memory Usage
508 MB
Jenkins health
1.0
JVM Uptime
N/A
Jenkins nodes offline
None!
CPU Usage
0.00245%
Total Jobs
None!
Sucessful Jobs
None!
Aborted Jobs
No data
Unstable Jobs
No data
Failed Jobs
None!
Jenkins queue size
0

ACTS, Head Quarters,
Pune





8.CONCLUSION AND FUTURE SCOPE

CONCLUSION:

This project successfully implemented a DevSecOps pipeline for a microservices application, integrating CI/CD, security checks, and monitoring. Tools like Jenkins, SonarQube, Trivy, and Grafana ensured secure, automated, and reliable deployments on Kubernetes (EKS), demonstrating the benefits of DevSecOps in cloud-native environments.

FUTURE SCOPE:

1. Implement advanced security scanning tools like **Aqua Security** or **Anchore** for runtime protection.
2. Enable **multi-cloud deployments** using platforms like **Azure AKS** or **Google GKE** for improved flexibility.
3. Adopt **Infrastructure as Code (IaC)** using **Terraform** or **AWS CloudFormation** to automate environment provisioning.
4. Enhance monitoring with **real-time alerting**, **log aggregation** (e.g., **ELK Stack**), and **AI-based anomaly detection**.
5. Apply a **Zero Trust security model** with fine-grained identity and access controls.
6. Introduce a **Service Mesh** (e.g., **Istio**, **Linkerd**) for better traffic management and security between microservices.
7. Expand the CI/CD pipeline with **automated performance, security, and compliance testing**.
8. Use **Secrets Management tools** like **HashiCorp Vault** or **AWS Secrets Manager** for secure credential handling.
9. Integrate **chaos engineering practices** to test system resilience under failure conditions.
10. Improve scalability and availability through **auto-scaling groups** and **load balancers** in production.

9.REFERENCE

- https://docs.aws.amazon.com/eks/latest/userguide/what-is-eks.html
- https://kubernetes.io/docs/home/
- https://www.jenkins.io/doc/
- https://www.sonarsource.com/products/sonarqube/
- https://grafana.com/docs/