DevSecOps CI/CD Pipeline with Jenkins, Docker, Trivy, SonarQube, Prometheus, Grafana & GitHub Actions

A Secure CI/CD Automation Pipeline Built with Open Source Tools

Anita Nnamdi www.linkedin.com/in/anita-nnamdi
GitHub: https://github.com/Anita-ani/SECURESNAP

Table of Contents

Project Overview

SECURESNAP is a hands on DevSecOps pipeline built using Jenkins, Docker, GitHub Actions, SonarQube, GitHub Webhooks, Prometheus, and Grafana. to demonstrate secure automation from code commit to test and code analysis.

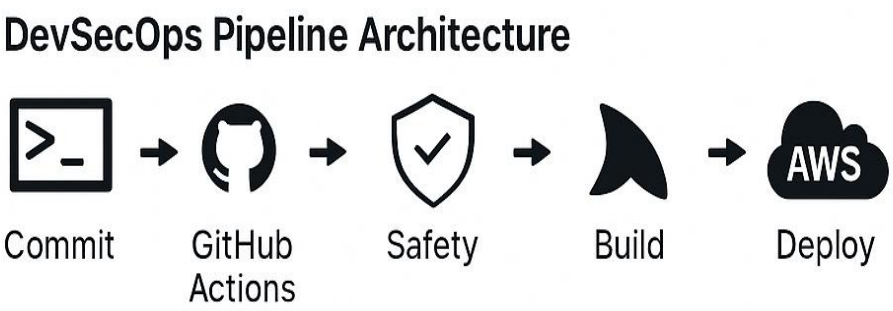
It showcases how to:

- Automate secure CI/CD pipelines
- Integrate static code analysis via SonarQube
- Use Dockerized Jenkins agents
- Monitor system health with Prometheus & Grafana
- Enforce DevSecOps principles from build to deploy

Tools Used

Tool	Purpose
Jenkins	CI/CD automation
Docker	Containerization of Jenkins/SonarQube
GitHub	Code repository & webhook trigger
SonarQube	Static code analysis
Prometheus	System metrics collection
Grafana	Metrics dashboard & visualization
GitHub Actions	Optional alternative pipeline
Trivy	Container vulnerability scanner

Architecture Diagram



DevSecOps Pipeline Architecture

This diagram shows:

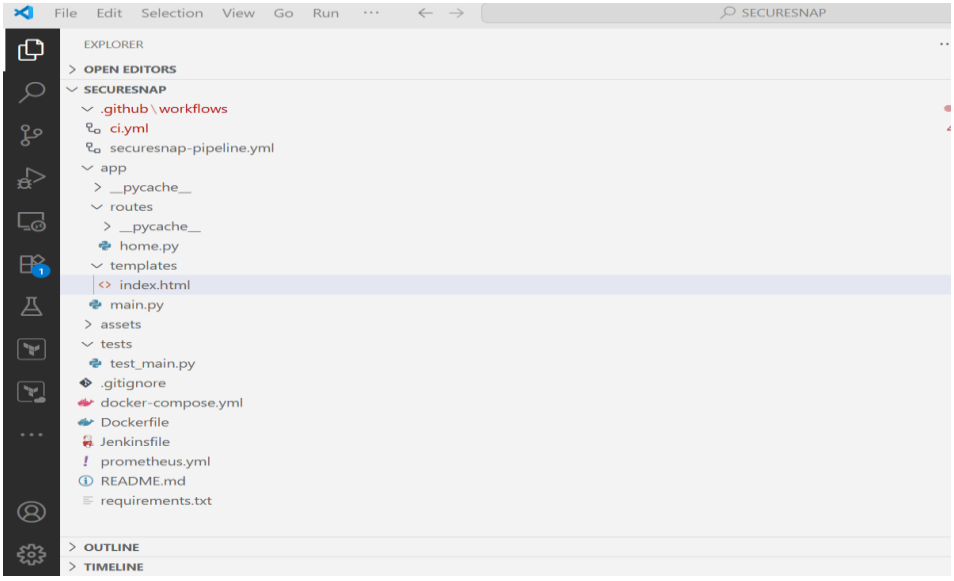
- Code **commit** triggers the pipeline
- **GitHub Actions** runs the CI/CD workflow
- **Safety** checks run (e.g., dependency scanning, linting, SAST)
- **Build** stage compiles/test/packages the app securely
- App is **deployed to AWS**
- Entire pipeline follows **DevSecOps best practices** by integrating security early

Step-by-Step Setup

a. Project Structure and Files

Create your project folder (SECURESNAP) with the following structure:

```
SECURESNAP/
- app/
  └── main.py
- tests/
   └── test_main.py
- Dockerfile
 docker-compose.yml
-  requirements.txt
-  .github/
   └── workflows/
       └── ci.yml
-  Jenkinsfile
```



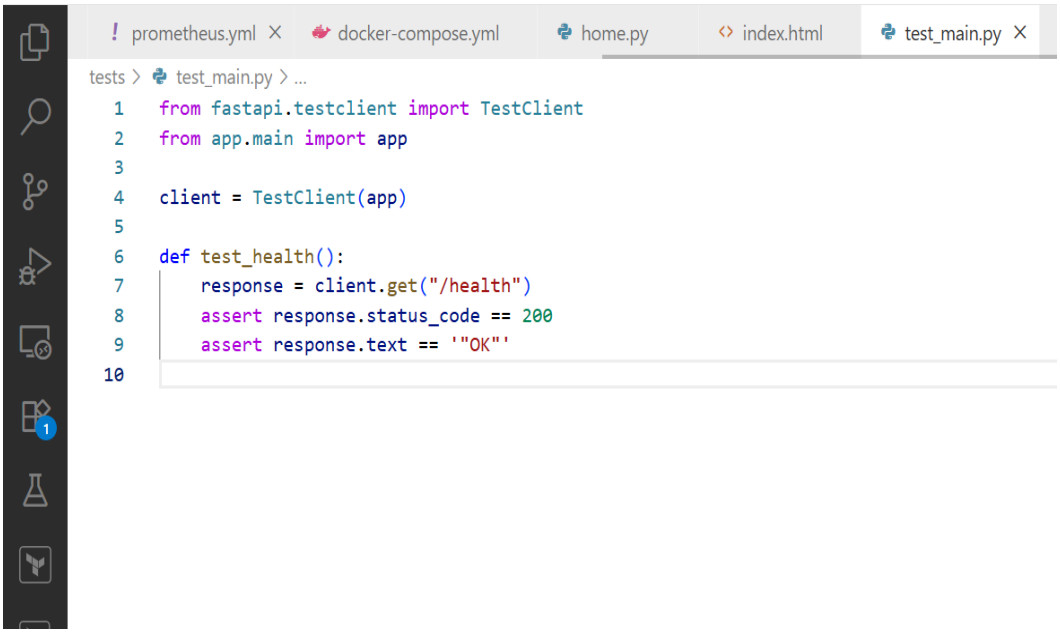
Build the FastAPI App

main.py in app/

This is the core FastAPI application file where the API logic lives. It defines routes, responses, and how the app behaves.

test_main.py

Contains automated tests to verify that your API endpoints work correctly. Helps catch bugs and ensures code reliability during development and deployment.



Docker Agent Setup

Create a Docker agent with required tools

In terminal run:

RUN apt update && \

apt install -y docker.io docker-compose nodejs npm && \

npm install -g snyk

SonarQube Integration

1. Launch SonarQube using:

- docker run -d --name sonarqube -p 9000:9000 sonarqube

2. Get the token from Sonar dashboard.

3. **Initial Setup for SonarQube**

- Visit `http://localhost:9000`
- Login (default: `admin` / `admin`)
- Change password
- Go to **My Account → Security → Generate Token**

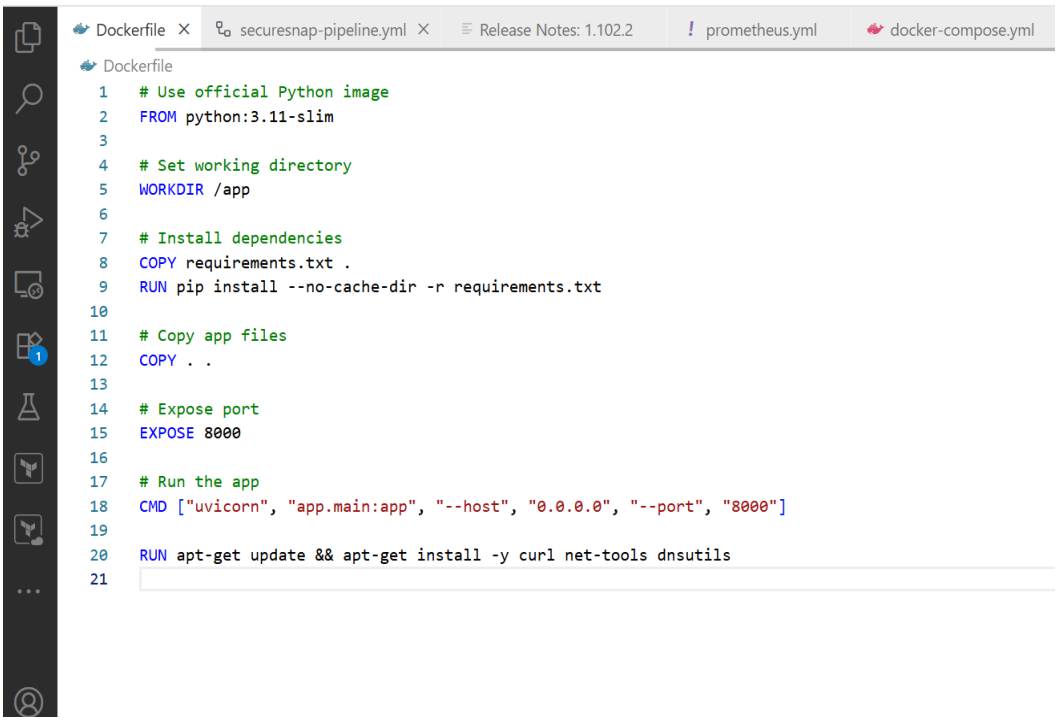
4. Add **Secret Text**:

- Name it something like `jenkins-sonar`
- Copy and save this token

Create a Dockerfile

`Dockerfile`

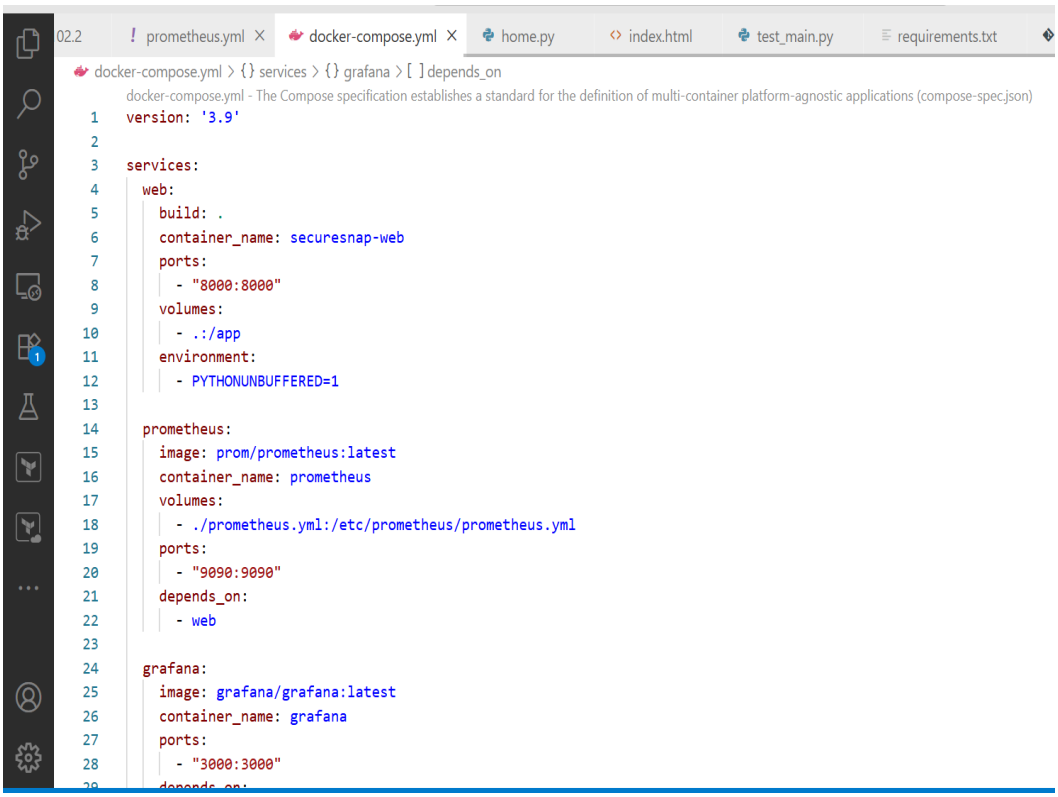
`Defines how to build the app's container image—sets up Python, installs dependencies, copies app files, exposes the app port, and runs the server.`



Create docker-compose.yml

docker-compose.yml

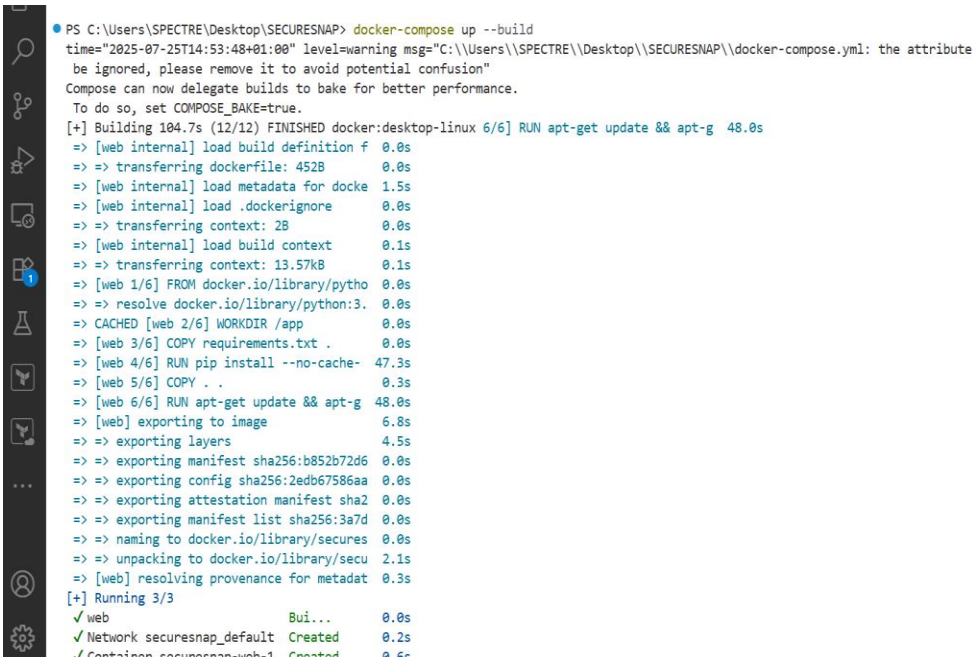
It specifies the Docker Compose file format version. Version 3.9 is stable and compatible with recent Docker versions, supporting features like named volumes, secrets, and improved service definitions.



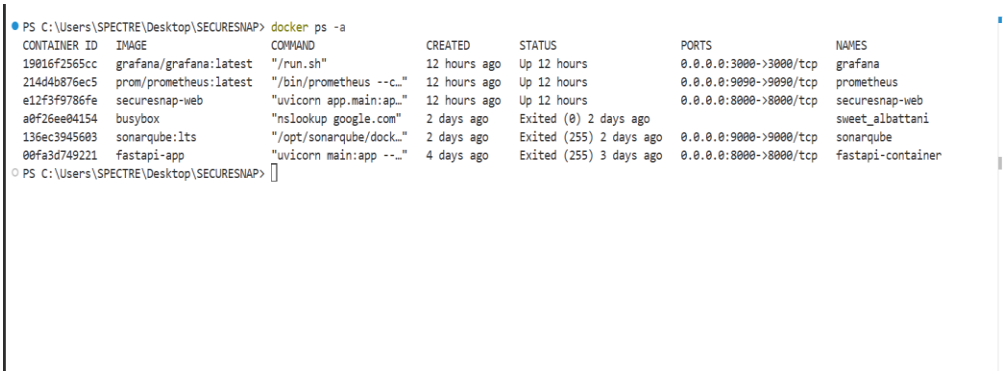
Build and Run

- `docker-compose build`

-`docker-compose up`



-docker ps –a to view running container

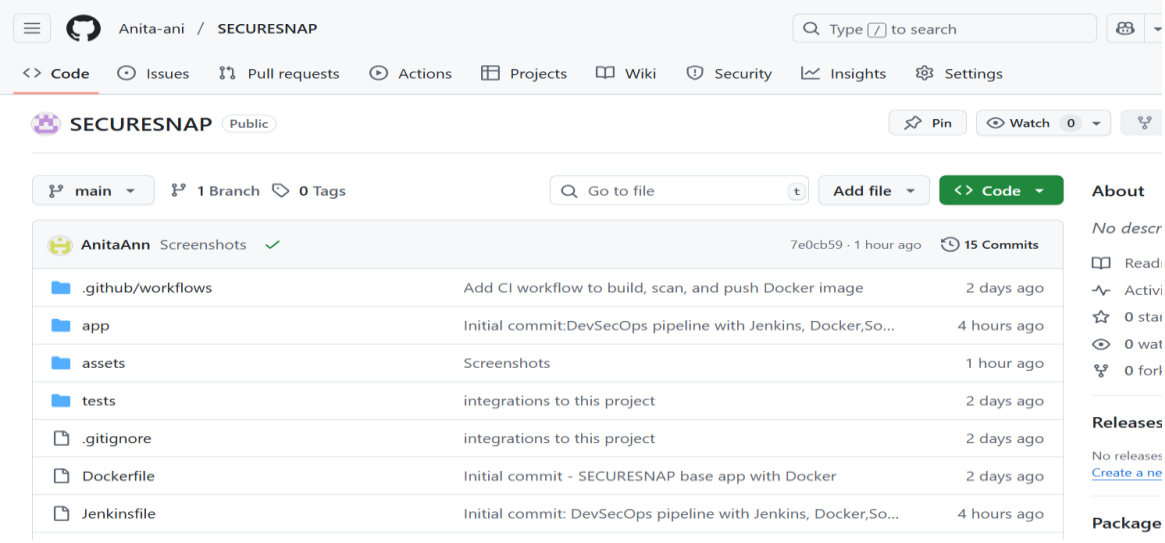


Push to Github

git add .

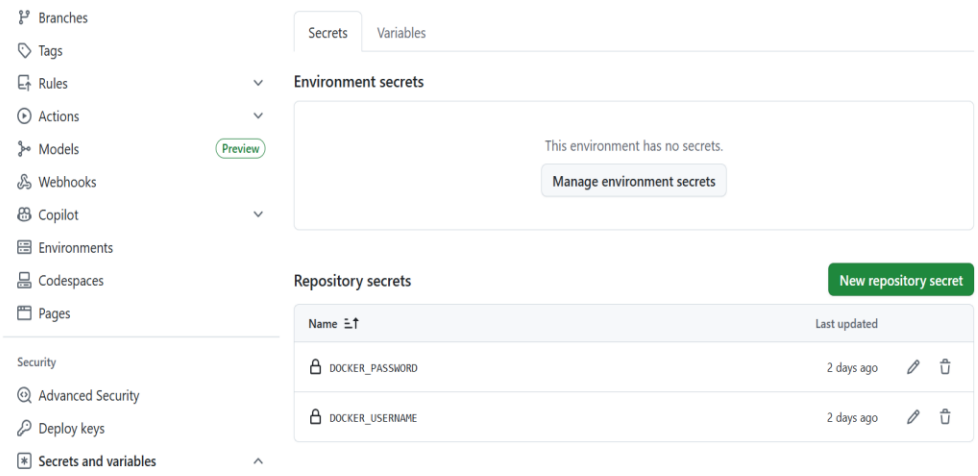
git commit -m "Initial commit"

git push origin main



Add DockerHub Secrets to GitHub

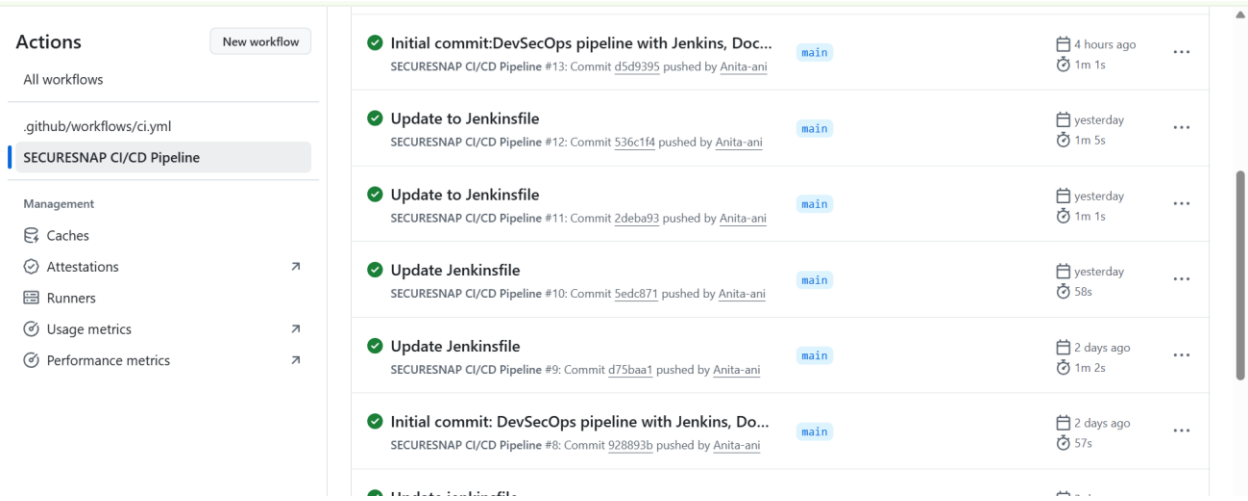
- Go to your repo → Settings → Secrets → Actions.
- Add:
- DOCKER_USERNAME
- DOCKER_PASSWORD
- Log into docker hub to generate Personal access token, copy token immediately and paste in DOCKER_PASSWORD and docker username in DOCKER_USERNAME



Create CI Workflow with Trivy

.github/workflows/ci.yml

To ensure our Docker image is secure and free from known vulnerabilities, we integrate **Trivy**, a powerful container scanning tool, into our GitHub Actions CI workflow. This workflow automatically scans the Docker image every time we push changes to the `main` branch, helping us catch CVEs and misconfigurations early in the development cycle.

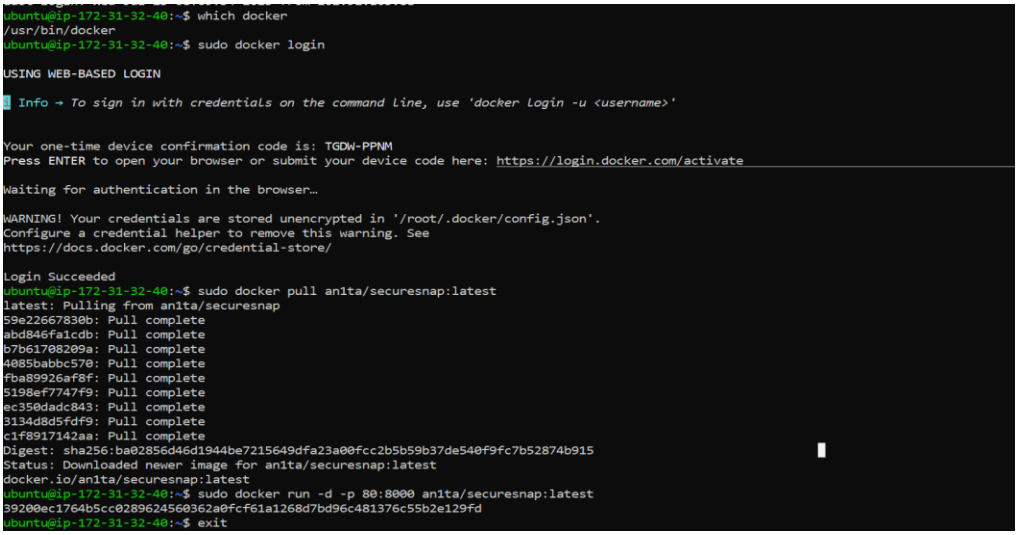


Deploy to AWS EC2

Steps:

1. Launch EC2 (Ubuntu)
   -Go to security groups and apply inbound rules, port 22, 80 and 443.
   -Find

2. Create a key pair and download using .pem
3. ls to find downloaded key pair then log into EC2 using SSH
4. SSH into instance using ssh -i "your-key.pem" ubuntu@<public-ip>(click on connect to find).



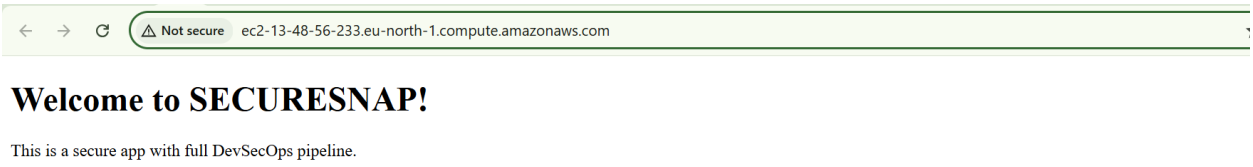
5. Install docker using:
   - curl -fsSL https://get.docker.com -o get-docker.sh
   sudo sh get-docker.sh
6. Log into docker using:
   -sudo docker login
   -Copy https://login.docker.com/activate in terminal and paste to browser.
   -Enter the confirmation code provided in terminal.

7. Pull image using:

-docker pull an1ta/securesnap:latest

-docker run -d -p 80:8000 an1ta/securesnap:latest

8. Copy EC2 instance ip (Connection to ec2-13-48-56-233.eu-north-1.compute.amazonaws.com)



Step-by-step setup to integrate Prometheus and Grafana via Docker so you can scrape FastAPI metrics and visualize them.

Step1: Expose fastAPI Metrics

First, install and expose metrics from your FastAPI app:

In **requiremenst.txt** include:

prometheus-client

In **main.py include:**

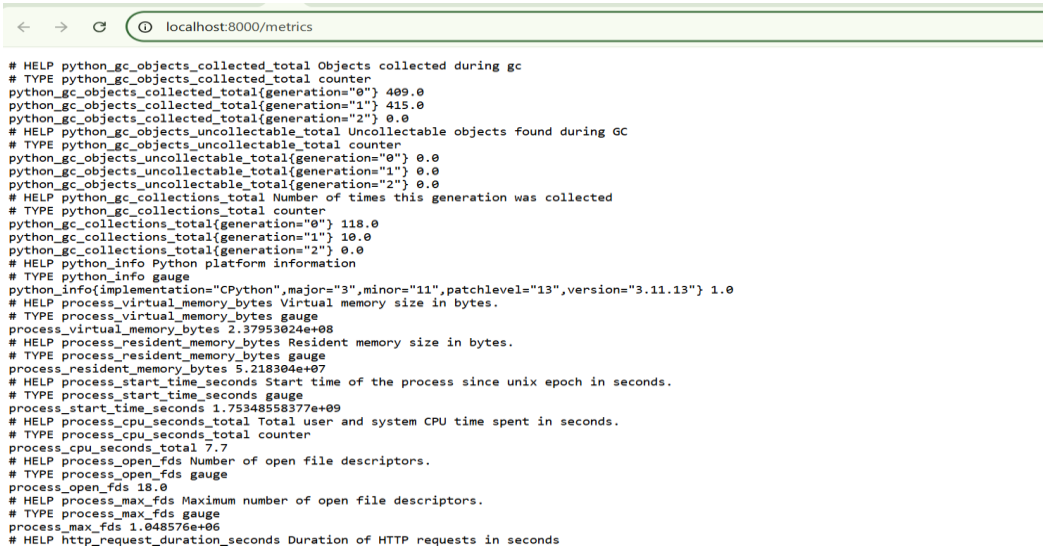
from fastapi import FastAPI from prometheus_client import start_http_server, Counter import threading

app = FastAPI()

REQUEST_COUNT = Counter("app_requests_total", "Total number of requests")

@app.get("/") def read_root(): REQUEST_COUNT.inc() return {"message": "Hello, Prometheus!"}

Refer to Repo



Prometheus + Grafana Monitoring Setup

Step by step:

1. Install Prometheus

 - sudo useradd --no-create-home --shell /bin/false prometheus

-sudo mkdir /etc/prometheus /var/lib/prometheus

2. prometheus.yml

This is Prometheus’ configuration file. It tells Prometheus **where to scrape metrics from**—such as your FastAPI app or other services—by defining jobs, targets, and how often to collect data.

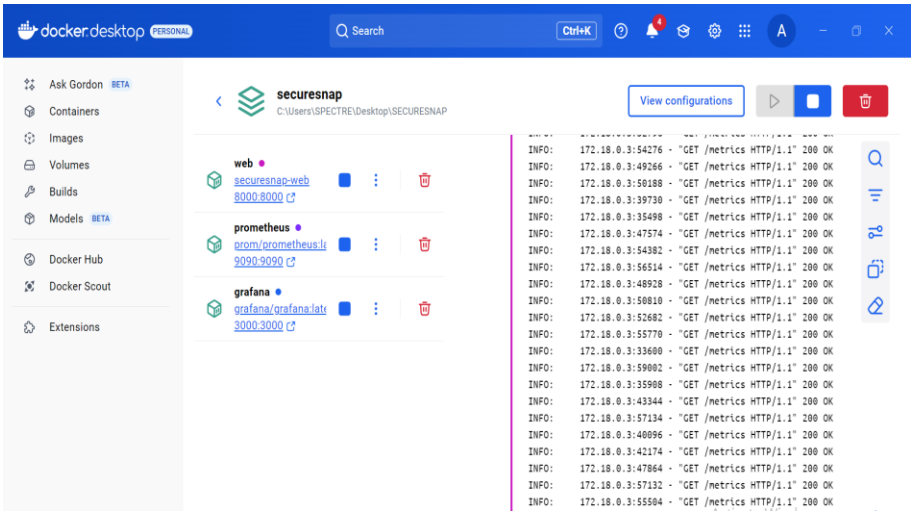
Grafana

1. Install Grafana
   - sudo apt-get install -y apt-transport-https software-properties-common
   - sudo add-apt-repository "deb https://packages.grafana.com/oss/deb stable main"
   - sudo apt-get update
   - sudo apt-get install grafana
   - sudo systemctl start grafana-server

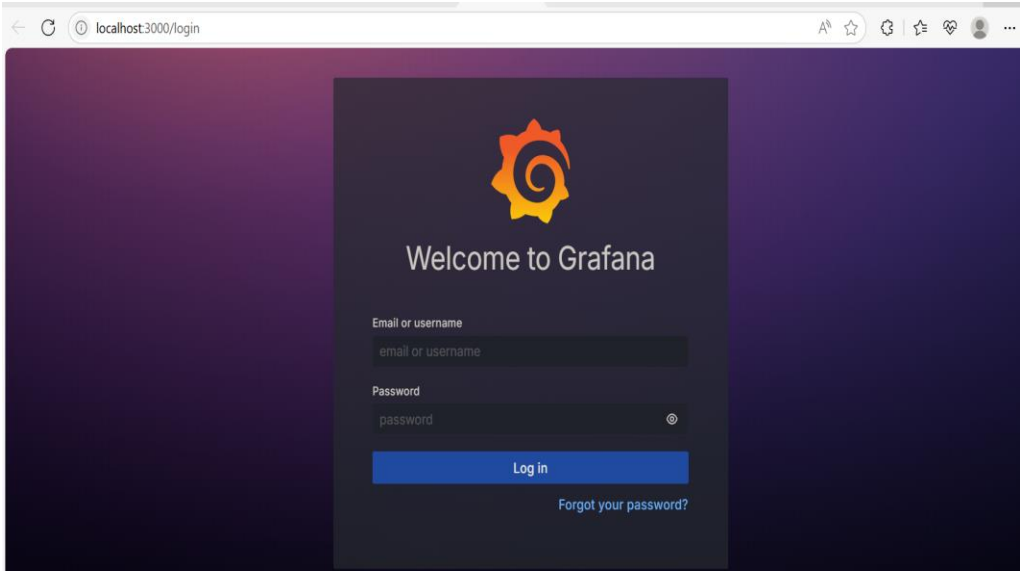
Login: http://localhost:3000/

Start Everything using:

-docker-compose up –build



- Visit docker desktop to view running container
- Visit Prometheus: http://localhost:9090
- Visit Grafana: http://localhost:3000 (login with admin/admin)



Configure Grafana

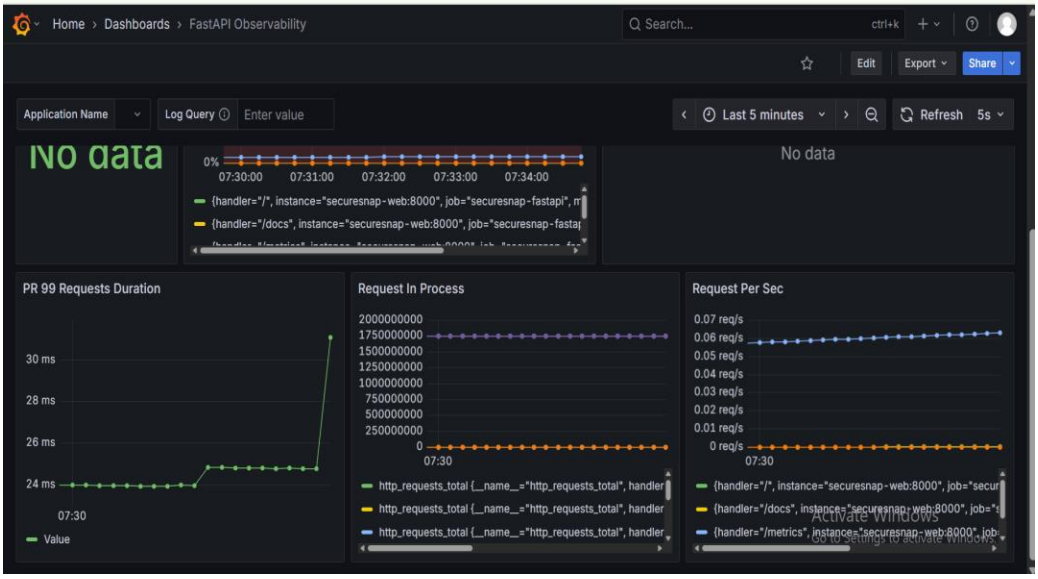
- Go to **http://localhost:3000**
- Login: `admin / admin`
- Add **Prometheus** as a data source:
- Set URL to: `http://prometheus:9090`
- `Save and test`
- Import a dashboard (or create one) to visualize metrics
- You can use a popular Prometheus + FastAPI dashboard like ID: 18739
- Go to the **+ (Create)** menu → Click **Import**
- In **Import via Grafana.com**, paste:
- Click load and Prometheus data source you just added
- Click Import

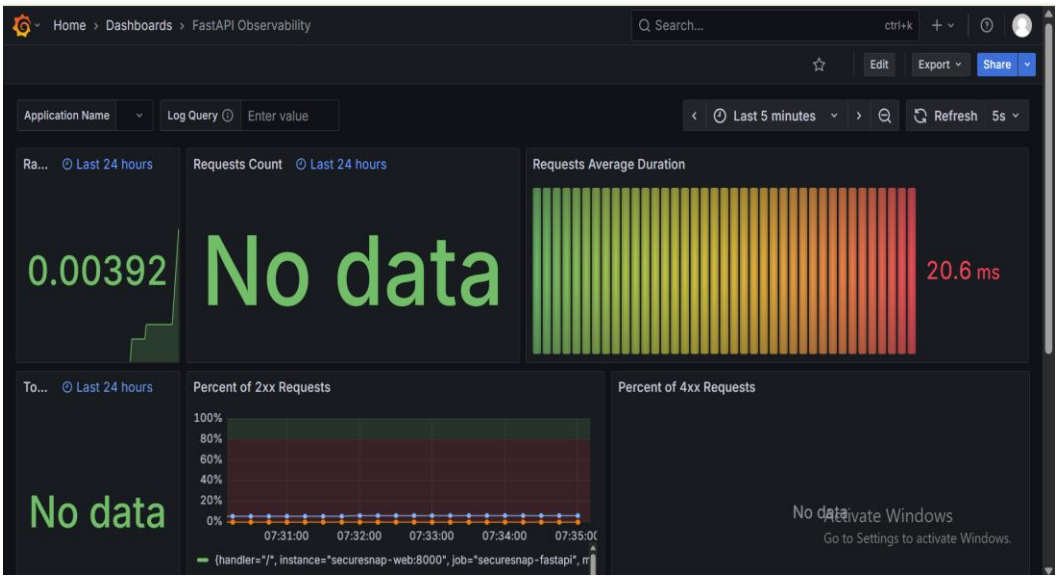
View the FastAPI Dashboard

Once imported:

You'll see a ready-made dashboard showing:

- Request count
- Response time
- Error rates
- Uptime
- CPU/RAM (if exposed)





Jenkins Setup

1. Install Java using:
   -sudo apt install -y openjdk-17-jdk

2. Install Jenkins using:
   - wget -q -O - https://pkg.jenkins.io/debian/jenkins.io.key | sudo apt-key add -
   -sudo sh -c 'echo deb https://pkg.jenkins.io/debian binary/ > /etc/apt/sources.list.d/jenkins.list'
   -sudo apt update
   -sudo apt install jenkins

3. Start jenkins and login into your EC2 (eg,...http://ec2-13-48-56-233.eu-north-1.compute.amazonaws.com/)

4. Jenkins Plugins to Install

Go to **Manage Jenkins → Plugins → Available**, and install:

- GitHub Integration
- Docker Pipeline
- OWASP Dependency-Check
- Trivy Plugin
- Blue Ocean
- Pipeline
- SonarQUBE Scanner
- Docker Pipeline
- Prometheus metrics plugins

5. Create Jenkinsfile

The `Jenkinsfile` defines a **CI/CD pipeline** for automating the build, test, and deployment of your FastAPI app. It tells Jenkins what steps to run and in what order.

Key Stages:

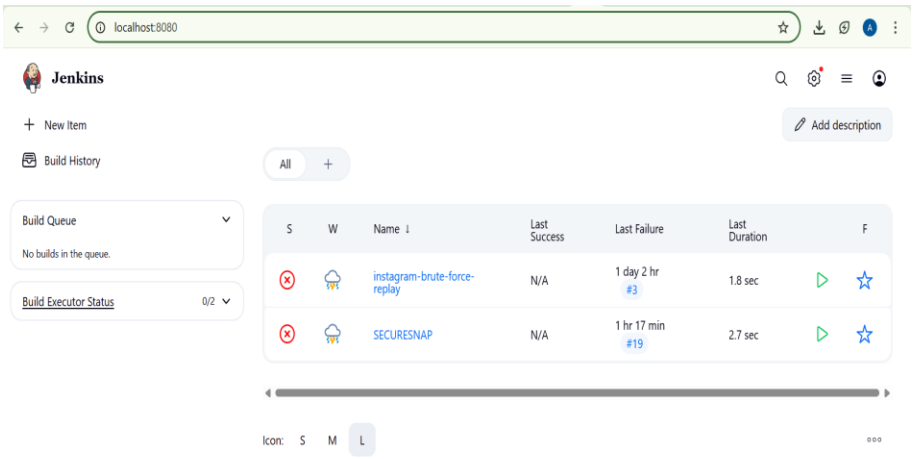
- `Build`: Uses Docker to build the FastAPI app image.
- `Test`: Runs unit tests (`pytest`) to catch issues early.
- `Security Scan`: Scans the image (e.g. with Trivy) for vulnerabilities.
- `Push`: Pushes the Docker image to Docker Hub (if tests pass).
- `Deploy`: Deploys the image to a server or container platform.

This keeps your delivery secure, reliable, and automated.

```
pipeline {
    agent any
<<<<<<< HEAD

    environment {
        PYTHONUNBUFFERED = 1
    }

    stages {
        stage('Clone Repo') {
            steps {
                echo 'Cloning repository...'
                // This is optional if Jenkins is pulling via a webhook
                // git url: 'https://github.com/your-username/your-repo.git'
            }
        }

        stage('Set up Python') {
            steps {
                echo 'Setting up Python...'
                sh 'python3 -m venv venv'
                sh '. venv/bin/activate && pip install --upgrade pip setuptools'
            }
        }
```

Add GitHub Repo in Jenkins

- Create a new **Pipeline Job (New item)**
- Set name (Item name)
- Click on Pipeline then Ok
- Navigate to Pipeline
- Connect it to GitHub under "Pipeline from SCM"
- Under SCM choose Git
- Paste Repo URL
- Branch main or master then save
- Use your GitHub personal access token if private



GitHub Webhook

This allows Jenkins to automatically build your project whenever you **push changes to GitHub**, by using **webhooks** to notify Jenkins of a new event.

Configure Jenkins Job:

1. Go to your Jenkins job > **Configure**
2. Under **Source Code Management**, select:
    a. Git
    b. Add your **GitHub repo URL**
    c. Use credentials if needed (GitHub token or SSH key)
3. Under **Build Triggers**, check:
    a. `GitHub hook trigger for GITScm polling`
4. Save the configuration.

Setup Recap

Automatically trigger Jenkins pipeline builds from GitHub commits and monitor Jenkins using Prometheus and Grafana.

Components:

- GitHub repo with credentials/token set up
- Jenkins (with Docker agents or controller)
- GitHub Webhook → Jenkins webhook endpoint
- Prometheus scrapes Jenkins metrics
- Grafana visualizes Prometheus data

Security Enhancements

Component	Security Best Practices
Jenkins Server	Use HTTPS (e.g. behind NGINX), disable anonymous access
GitHub Webhook	Use **Secret Token** (set in webhook and Jenkins plugin)
Credentials	Use Jenkins **Credentials Plugin** to store GitHub tokens securely
Docker Builds	Run agents with limited privileges, avoid root in Dockerfiles
Prometheus	Restrict Prometheus UI access and scrape targets with firewalls
Grafana	Use strong passwords, disable anonymous dashboards, restrict sharing

Common Errors and Fixes

Error Message	Cause	Fix
GitHub webhook returns 403 or timeout	Jenkins not publicly accessible	Use `ngrok`, configure reverse proxy e.g. NGINX

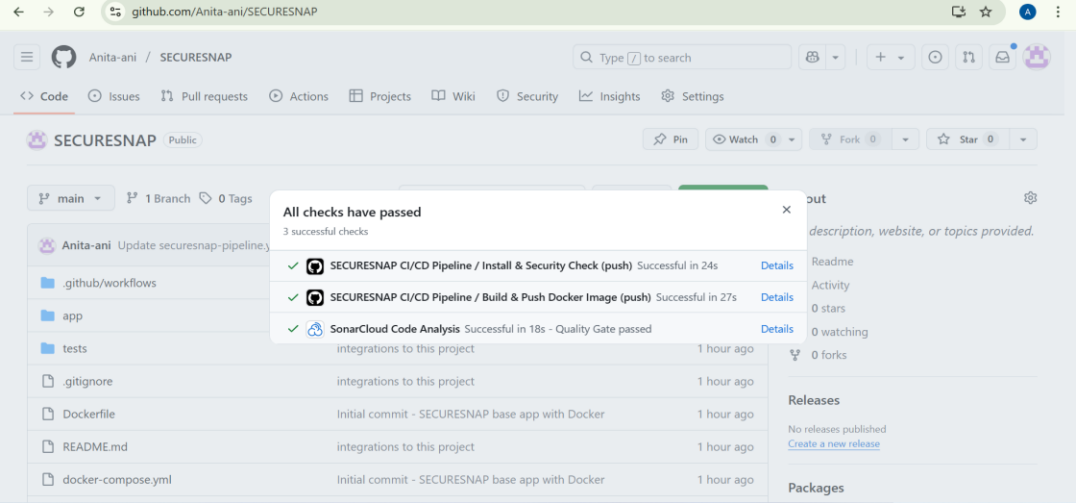
Jenkins not triggering on push	Webhook misconfigured or wrong Jenkins URL	Verify webhook URL ends with `/github-webhook/`
Git clone fails	Missing credentials	Add GitHub token or SSH key to Jenkins credentials
Prometheus shows Jenkins job as down	Wrong target URL or port	Ensure Jenkins exposes `/prometheus/` endpoint
Grafana dashboard empty	Wrong PromQL queries or missing data	Verify Prometheus is scraping metrics correctly

How to Run the Pipeline

1. **Make a Code Change in GitHub**
    a. Edit any file (e.g. `main.py`), commit and push
2. **GitHub Webhook Fires**
    a. GitHub sends payload to Jenkins webhook endpoint
3. **Jenkins Receives Hook**
    a. If configured correctly, Jenkins fetches the repo and starts the pipeline
4. Build/Tests Run Inside Jenkins Pipeline
    a. Optionally include `Docker build`, `pytest`, `lint`, `security scan`, etc.
5. Prometheus Monitors Build Metrics
    a. If Jenkins Prometheus plugin is enabled, metrics are exposed and scraped
6. Grafana Dashboard Displays Build Trends
    a. Grafana visualizes builds, duration, failures, success rate, etc.

What Is Achieved with This Architecture?

By implementing this **GitHub Actions → SonarQube → Docker → AWS EC2 → Jenkins CI/CD → Prometheus + Grafana Monitoring** setup, you achieve the following:



1. Automated CI/CD Pipeline

- **Continuous Integration**: Every code push automatically triggers a build and test cycle.
- **Continuous Delivery/Deployment**: You can extend this to deploy to staging or production automatically (e.g... via Helm, Docker, or Kubernetes).
- **Benefits**:
    - Faster feedback
    - Fewer manual steps
    - Early bug detection
    - Consistent releases

2. Built In Security with SAST (Static Application Security Testing)

Run SAST tools like `bandit`, `semgrep`, or `trivy` during Jenkins pipeline stages.

What it Catches:

- Hardcoded secrets
- Insecure functions or APIs
- Dependency vulnerabilities

Benefits:

- Shift-left security
- Immediate feedback to developers
- Prevents vulnerable code from reaching production

Sample Jenkinsfile SAST Stage:

```
groovy
Copy code
stage('SAST') {
  steps {
    sh 'bandit -r . || true'
  }
}
```

3. Quality Gates

- Thresholds that code must pass (e.g... test coverage, linting score, SAST results).
- Use tools like SonarQube, `pylint`, `pytest-cov` integrated into the pipeline.
- **Example Rule**: Block merge if test coverage < 80% or lint score < 8/10.

Benefits:

- Prevents low quality or insecure code
- Maintains project standards
- Enforces coding discipline across teams

4. GitHub Webhook & GitHub Actions Compatibility

Webhook: Enables GitHub to notify Jenkins of push events, triggering builds.
Actions: You can optionally integrate GitHub Actions for:

- Pre-commit tests
- Linting and secrets scan before Jenkins build
- SAST checks on pull requests

Hybrid Example: GitHub Actions runs `truffleHog` for secrets detection → Jenkins runs build/test/deploy pipeline.

5. Full Observability and Monitoring

Prometheus + Grafana: You get metrics like:

- Build duration trends
- Success/failure rates
- Queue latency

Jenkins Prometheus Plugin: Exposes `/prometheus` metrics endpoint for scraping

Benefits:

- Performance bottleneck detection
- Audit trails for compliance
- Reliability metrics for teams

6. Modular, Scalable, and Secure DevSecOps Pipeline

This architecture promotes a **modular design** that supports:

Feature	Description
Secrets Management	Use Jenkins Credentials Plugin or HashiCorp Vault
Test Automation	`pytest`, `unittest`, `nose`, etc. in CI
Reusable Templates	Declarative pipelines or shared GitHub Actions
Multi Cloud Ready	Can extend pipeline to deploy to AWS, Azure, or GCP

This project demonstrates a secure and observable CI/CD pipeline with GitHub Actions, Jenkins, and monitoring using Prometheus and Grafana , all aligned with modern **DevSecOps practices** like **SAST**, **quality gates**, and **pipeline security**.

Whether for personal projects, team adoption, or enterprise implementation, this setup is flexible, scalable, and secure.

Thanks for reviewing this setup

Feel free to fork , contribute or reach out on www.linkedin.com/in/anita-nnamdi