# Project: Medicine Tracker

## Project Overview

**Medicine Tracker Pro** is a full-stack application designed to help users manage their medication schedules effectively. With user-friendly interfaces and backend logic, it ensures timely alerts, medicine tracking, and history logging. This project automates its development lifecycle using Jenkins, ensuring rapid, secure, and high-quality releases.

The CI/CD pipeline includes stages for static code analysis, Docker containerization, vulnerability scanning, and multi-target deployment using Docker and Railway.

## Pipeline Configuration Overview

This CI/CD pipeline automates the process of building, testing, securing, and deploying the application. It starts with code checkout, followed by test execution and coverage generation. Static analysis is performed using SonarQube and Semgrep, while security checks include secret scanning with Gitleaks and shell script linting. A Docker image is built, scanned for vulnerabilities using Trivy and Grype, and its SBOM is generated using Syft. After verifying dependencies, the image is pushed to a registry and deployed to Railway. A final health check ensures the application is running correctly post-deployment.



## 

Pipline configuration overview

## Tools and Technologies Used

* **Git/GitHub** - Version control system for source code management
* **Node.js & npm** - Backend runtime and package manager
* **React + Vite + TypeScript + Tailwind CSS** - Frontend tech stack
* **SQLite** - Database used by the backend
* **Jenkins** - CI/CD automation server
* **SonarQube** - Static code analysis and quality gate
* **OWASP Dependency-Check** - Detects vulnerabilities in dependencies
* **Docker** - Containerization of backend/frontend
* **Trivy** - Security scanner for Docker images
* **Railway** - Cloud deployment platform for frontend and backend

## Pipeline Stages and Descriptions

1. **Tools Installation**

* **Purpose:** Ensures that tools like Node.js, Docker, and Trivy are installed and configured on the Jenkins server.
* **Jenkins plugins used:** Git, Docker, SonarQube Scanner, OWASP Dependency-Check, and Trivy integration.

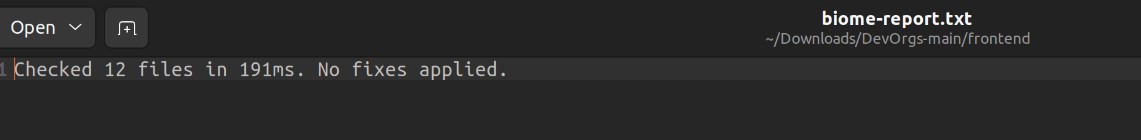
## Git-CheckOut

**Purpose:** Clone the repository from GitHub.

* **Tool:** Git
* **Action:** git ‘https://github.com/Aakashjn/MedicineTrackerPro.git'
* **Description:** This stage fetches the latest code from the specified GitHub repository. This ensures that the pipeline is always working with the most current version of the codebase.

## Run Tests and Generate Coverage

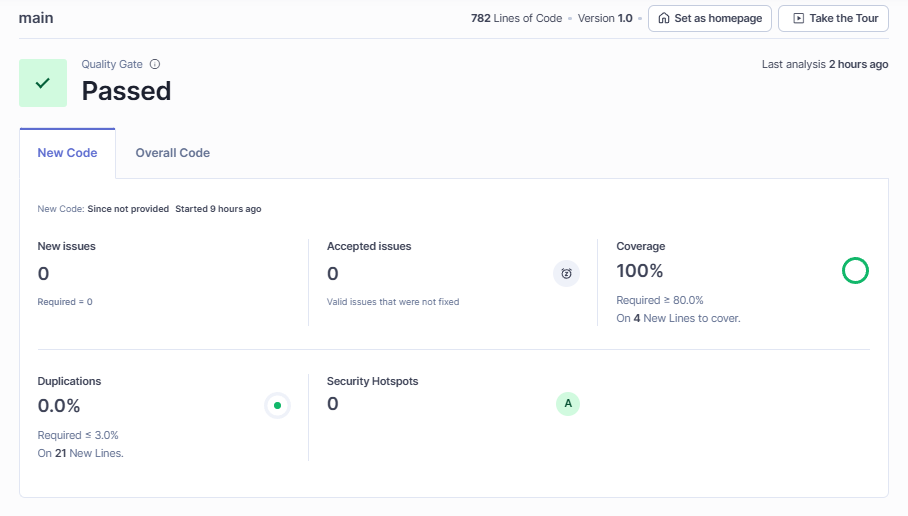
Automated tests are executed using the application's configured test runner (e.g., Jest, Mocha, or others for Node.js). This stage validates the correctness of individual components and the logic of the application. Simultaneously, code coverage tools (e.g., Istanbul or NYC) analyze which parts of the codebase are exercised by the tests, generating a coverage report. This information feeds into SonarQube and can also influence quality gates.



## SonarQube Analysis

SonarQube performs static analysis on the codebase to identify:

* Bugs
* Code smells
* Security vulnerabilities
* Duplications  
  It also consumes the test coverage report from the previous stage to assess how well-tested the code is. Based on pre-configured quality gates, it determines whether the code meets the required standards. If the gate fails (e.g., coverage is too low or a critical bug is found), the pipeline can be halted here.



## GitLeaks Secrets Scan

This stage uses Gitleaks to scan for sensitive data leaks such as API keys, credentials, or tokens hardcoded in the codebase or accidentally committed into Git history. Gitleaks checks against known regex signatures of secrets and flags any violations. This helps prevent accidental exposure of critical infrastructure credentials.

## Shell Script Linting

All shell scripts used in the project are linted using shellcheck, which analyzes for:

* Syntax errors
* Deprecated syntax
* Insecure command usage (e.g., eval, unquoted variables)
* Portability issues  
  This improves script reliability and makes them safer and more maintainable across environments.

## Docker Build

In this stage, the backend and/or frontend application is packaged into a Docker container using a Dockerfile. It captures all dependencies, runtime environments, and configurations, creating a portable image. This step is critical for achieving consistency across development, staging, and production environments.



# Syft SBOM generation

Syft is used to generate a Software Bill of Materials (SBOM) from the Docker image. The SBOM is a comprehensive inventory of all software components, libraries, and packages in the image. This data is important for compliance, software supply chain security, and vulnerability tracking, especially in regulated industries.

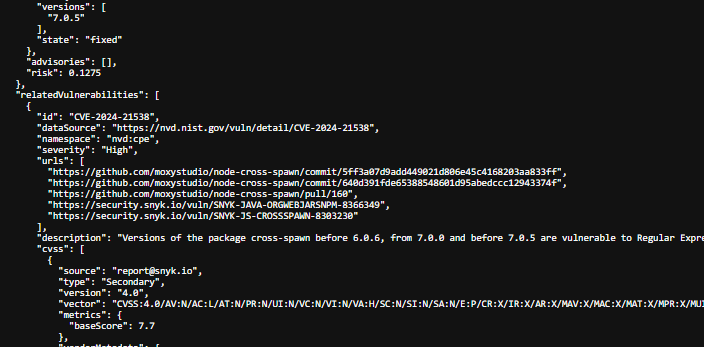


## Trivy Image Scanner

Trivy is integrated into the Jenkins pipeline to perform container image vulnerability scanning. It is a simple and comprehensive security scanner that detects vulnerabilities in operating system packages and application dependencies. In this stage, Trivy runs as a Docker container and scans the newly built Docker image for known vulnerabilities with high and critical severity. This ensures that only secure images are deployed, reducing the risk of introducing security flaws into the production environment.

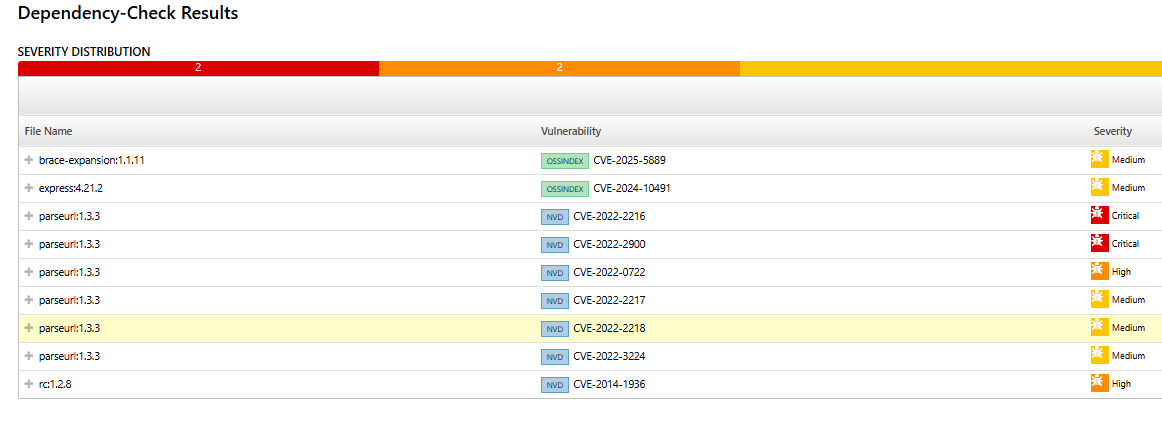
Grype Image Scanner

Grype is integrated into the Jenkins pipeline to perform in-depth container image vulnerability scanning. It is a powerful and reliable open-source vulnerability scanner developed by Anchore, designed to detect known vulnerabilities in the packages and dependencies of container images. During this stage, Grype analyzes the newly built Docker image using a continuously updated vulnerability database, identifying issues with high and critical severity levels. By incorporating Grype into the CI/CD process, the pipeline ensures that insecure images are flagged early, maintaining a strong security posture before deployment to production.



## OWASP Dependency checker

This stage performs a detailed scan of the application’s declared dependencies using OWASP Dependency-Check. It compares them against known CVEs (Common Vulnerabilities and Exposures) in public databases and flags those with known security issues. It provides CVSS scores, CWE references, and often includes remediation advice.



## Deploy to Docker and Railway

The application container is deployed to a local or remote Docker runtime to simulate a production environment. This acts as a final pre-deployment validation to ensure that the image works when started in isolation, including correct environment variable handling, port binding, and service availability.

Using the Railway CLI or integration, the backend and frontend services are deployed to the Railway cloud platform. This platform handles environment provisioning, build execution, and runtime monitoring. The deployment is typically triggered via a webhook or post-image-push event, depending on the setup.

## Health Check

The final stage performs automated health checks on the live application by sending HTTP requests to predefined endpoints (e.g., /health, /api/status). It verifies that the deployed services are running, responsive, and returning expected HTTP status codes (usually 200).

## The Pipeline Complete Code

pipeline {

agent any

tools {

jdk 'jdk21'

}

environment {

SCANNER\_HOME = tool 'Sonar-scanner'

DOCKER\_IMAGE = 'aakashjn/medicinetrackerpro'

}

stages {

stage('Git checkout') {

steps {

git branch: 'main', url: 'https://github.com/Aakashjn/MedicineTrackerPro.git'

}

}

stage('Run Tests and Generate Coverage') {

steps {

dir('backend') {

script {

try {

bat 'npm install'

bat 'npm test'

} catch (err) {

echo "⚠ Test stage failed: ${err.getMessage()}"

currentBuild.result = 'FAILURE'

}

}

}

}

}

stage('Sonarqube Analysis') {

steps {

withSonarQubeEnv('SonarQube') {

withCredentials([string(credentialsId: 'sonarqube-token', variable: 'SONAR\_TOKEN')]) {

bat """

cd backend

${tool('Sonar-scanner')}/bin/sonar-scanner ^

-Dsonar.projectKey=medicinetrackerpro ^

-Dsonar.projectName="Medicine Tracker Pro" ^

-Dsonar.token=%SONAR\_TOKEN% ^

-Dsonar.sources=. ^

-Dsonar.test.inclusions=\*\*/\_\_tests\_\_/\*\*/\*.js ^

-Dsonar.javascript.lcov.reportPaths=coverage/lcov.info

"""

}

}

}

}

stage('Gitleaks Secrets Scan') {

steps {

bat 'gitleaks detect --source=. --report-format=json --report-path=gitleaks-report.json || exit /b 0'

}

post {

always {

archiveArtifacts artifacts: 'gitleaks-report.json', fingerprint: true

}

}

}

stage('Shell Script Linting') {

steps {

bat 'for /r %%f in (\*.sh) do shellcheck "%%f" || exit /b 0'

}

}

stage('Semgrep Static Scan') {

steps {

bat 'semgrep scan --config p/default --json > semgrep-report.json || exit /b 0'

}

post {

always {

archiveArtifacts artifacts: 'semgrep-report.json', fingerprint: true

}

}

}

stage('Docker Build') {

steps {

script {

bat "docker build -t ${DOCKER\_IMAGE}:latest ."

bat "docker tag ${DOCKER\_IMAGE}:latest ${DOCKER\_IMAGE}:${BUILD\_NUMBER}"

bat "docker images ${DOCKER\_IMAGE}"

}

}

}

stage('Syft SBOM') {

steps {

bat "syft ${DOCKER\_IMAGE}:latest -o json > sbom.json || exit /b 0"

}

post {

always {

archiveArtifacts artifacts: 'sbom.json', fingerprint: true

}

}

}

stage('Trivy Docker Scan') {

steps {

bat "E:\\trivy\\trivy.exe image ${DOCKER\_IMAGE}:latest"

}

}

stage('Grype Vulnerability Scan') {

steps {

script {

try {

bat """

echo "Running Grype vulnerability scan..."

grype ${DOCKER\_IMAGE}:latest --output json --file grype-report.json

grype ${DOCKER\_IMAGE}:latest --output table

"""

} catch (Exception e) {

echo "Grype scan completed with findings: ${e.getMessage()}"

}

}

}

post {

always {

archiveArtifacts artifacts: 'grype-report.json', fingerprint: true

}

}

}

stage('Docker Push') {

steps {

script {

withDockerRegistry(credentialsId: 'Docker', url: 'https://index.docker.io/v1/') {

bat "docker push ${DOCKER\_IMAGE}:latest"

bat "docker push ${DOCKER\_IMAGE}:${BUILD\_NUMBER}"

}

}

}

}

stage('Dependency Check') {

steps {

dependencyCheck additionalArguments: '''

--format HTML

--format XML

--suppression suppression.xml

--enableRetired

''', odcInstallation: 'dependency-check'

dependencyCheckPublisher pattern: 'dependency-check-report.xml'

}

post {

always {

archiveArtifacts artifacts: 'dependency-check-report.\*', fingerprint: true

}

}

}

stage('Docker Deploy') {

steps {

script {

bat "docker rm -f medicinetrackerpro || echo \"No existing container to remove\""

bat "docker run -d --name medicinetrackerpro -p 4000:4000 ${DOCKER\_IMAGE}:latest"

}

}

}

stage('Deploy to Railway') {

steps {

script {

echo "Current branch: ${env.BRANCH\_NAME}"

echo "Git branch: ${env.GIT\_BRANCH}"

def isMainBranch = (env.BRANCH\_NAME == 'main' ||

env.GIT\_BRANCH == 'main' ||

env.GIT\_BRANCH == 'origin/main')

if (isMainBranch) {

withCredentials([string(credentialsId: 'railway-token', variable: 'RAILWAY\_TOKEN')]) {

bat """

echo "✅ Deploying to Railway from main branch..."

railway login --token %RAILWAY\_TOKEN%

railway up --detach --service medicinetrackerpro --project %RAILWAY\_PROJECT\_ID%

"""

}

} else {

echo "⏭ Skipping Railway deployment - not on main branch"

echo "Current branch: ${env.BRANCH\_NAME ?: env.GIT\_BRANCH}"

}

}

}

}

stage('Health Check') {

steps {

script {

def isMainBranch = (env.BRANCH\_NAME == 'main' ||

env.GIT\_BRANCH == 'main' ||

env.GIT\_BRANCH == 'origin/main')

if (isMainBranch) {

echo "Waiting for Railway deployment to complete..."

sleep(60)

try {

def response = bat(

script: 'curl -s -o nul -w "%%{http\_code}" https://medicinetrackerpro-production.up.railway.app/health',

returnStdout: true

).trim()

if (response != '200') {

echo "⚠ Health check failed. HTTP status: ${response}"

error "Health check failed for Railway deployment."

} else {

echo "✅ Application deployed successfully and health check passed!"

}

} catch (Exception e) {

echo "⚠ Health check failed: ${e.getMessage()}"

error "Health check failed for Railway deployment."

}

} else {

echo "⏭ Skipping health check - not on main branch"

}

}

}

}

}

post {

always {

cleanWs()

}

success {

echo "🎉 Pipeline completed successfully!"

echo "🌐 Application URL: https://medicinetrackerpro-production.up.railway.app"

}

failure {

echo "❌ Pipeline failed. Check logs for details."

}

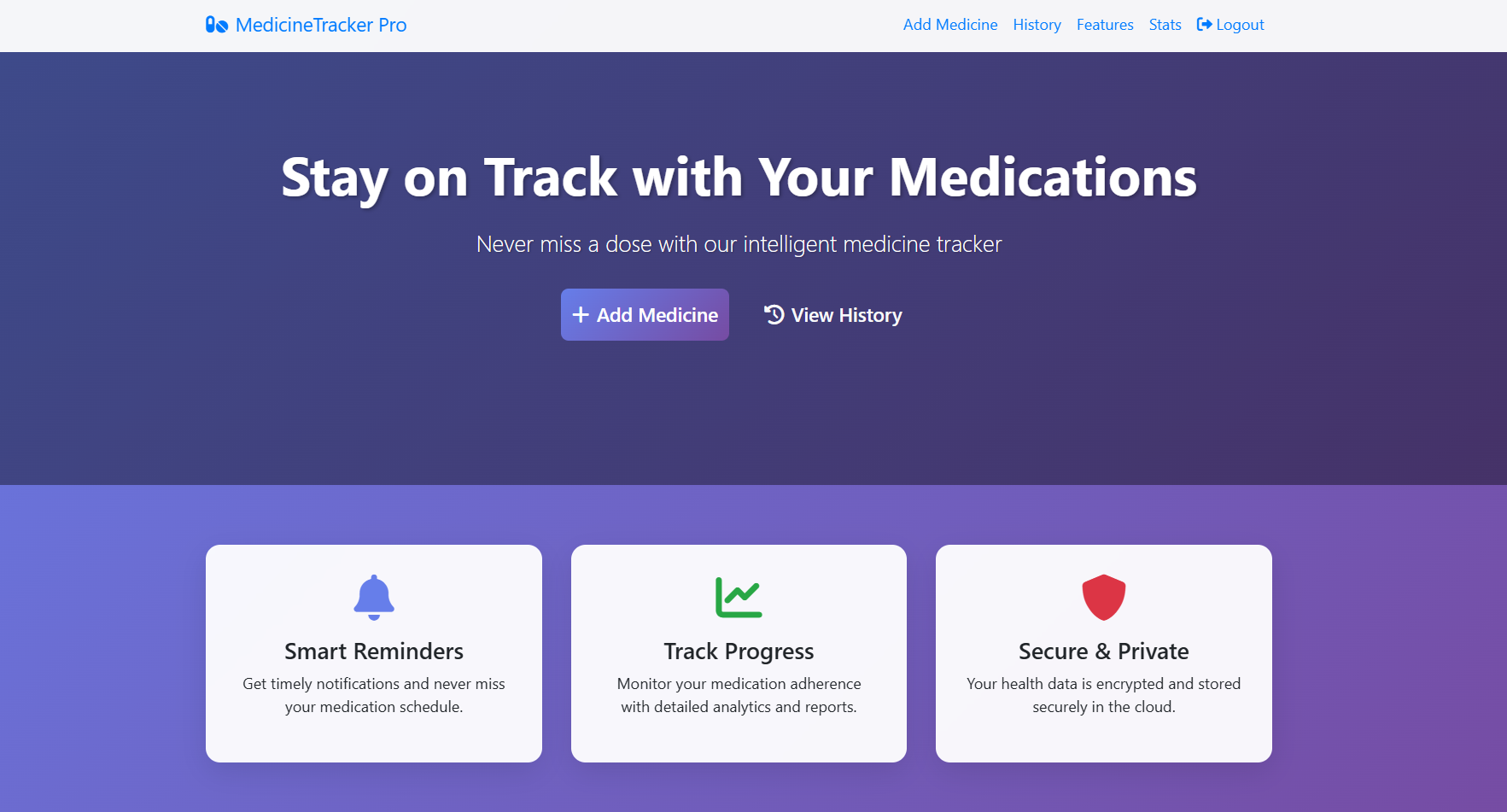
}

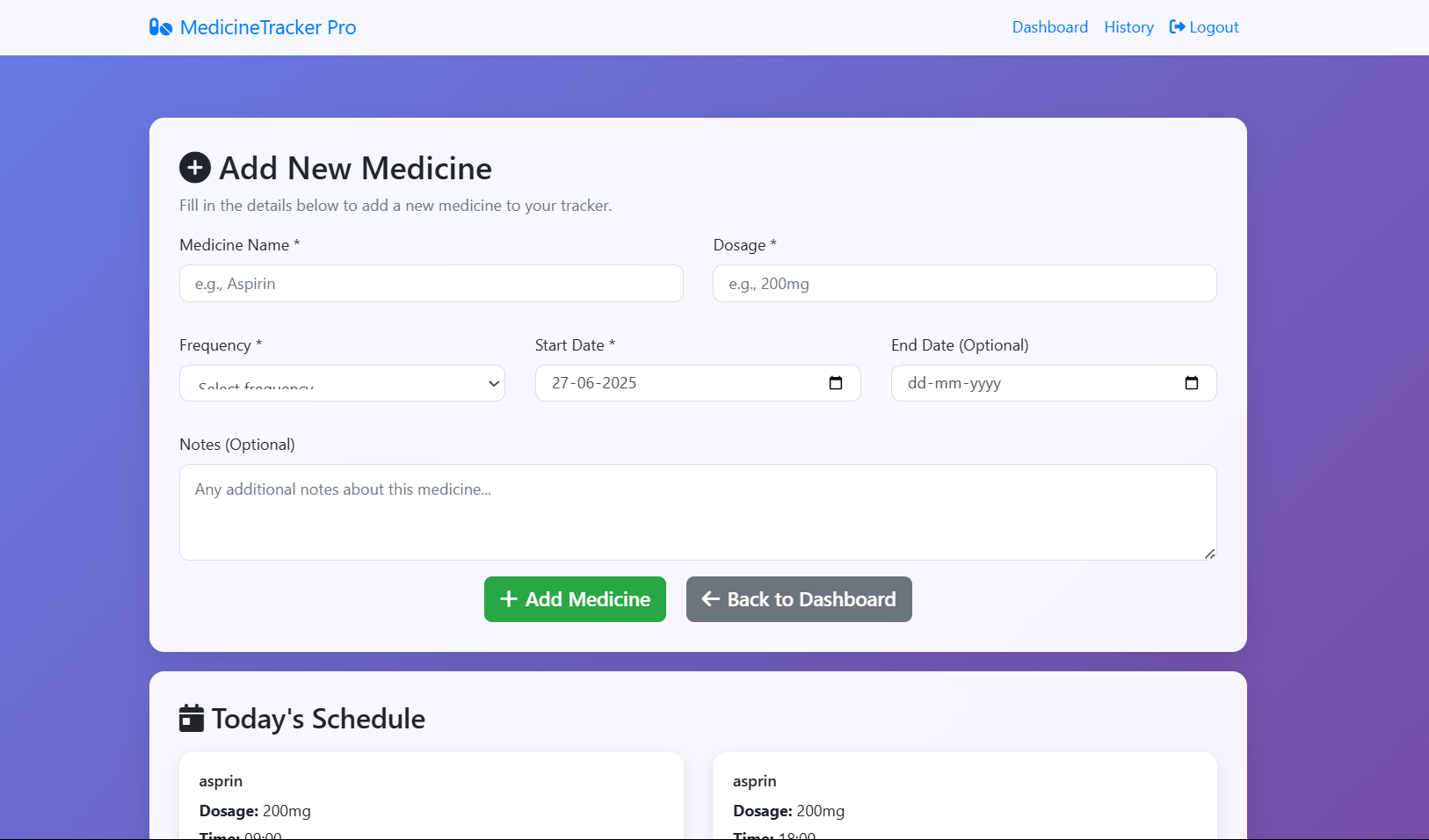
}

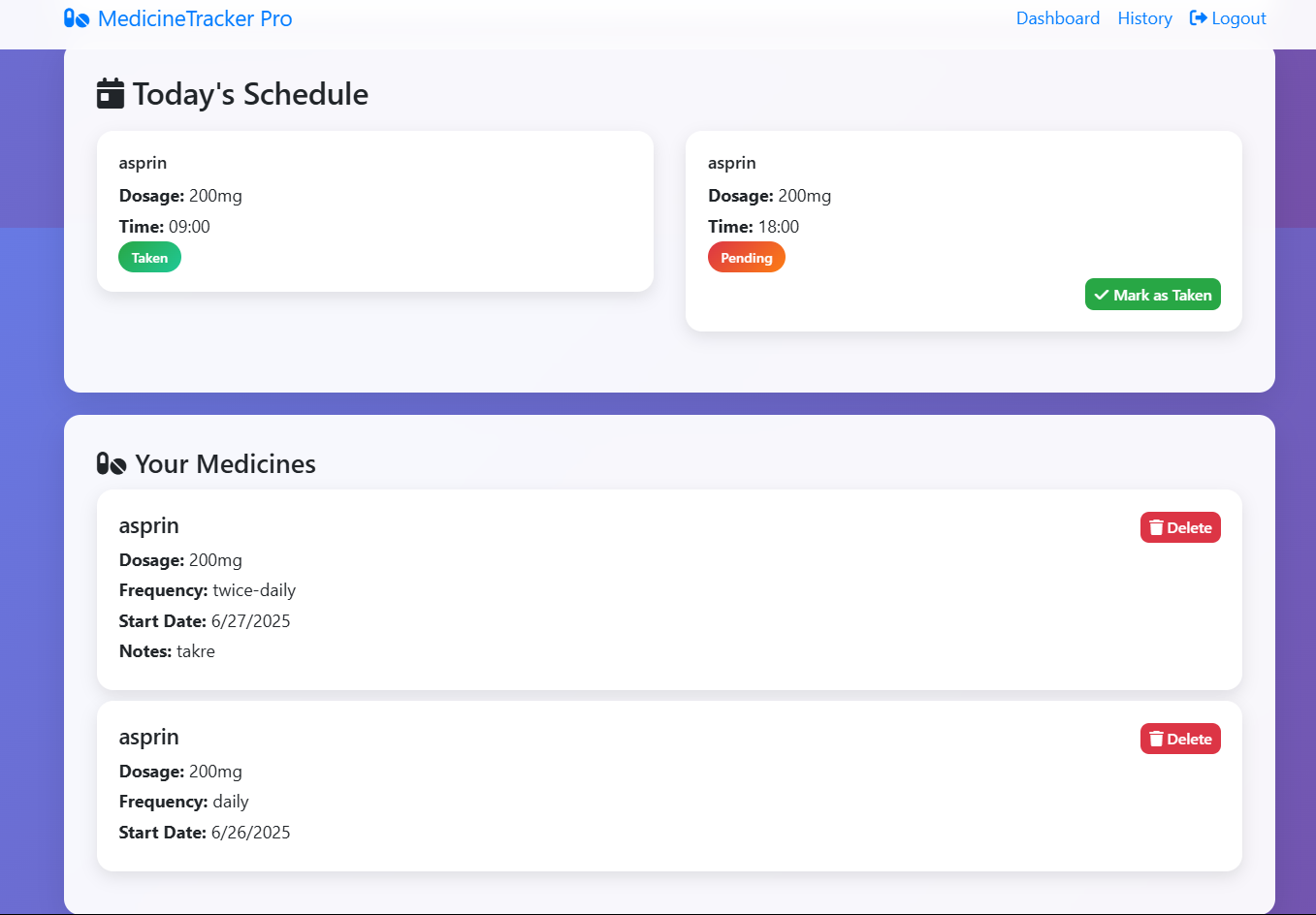
## Final Result

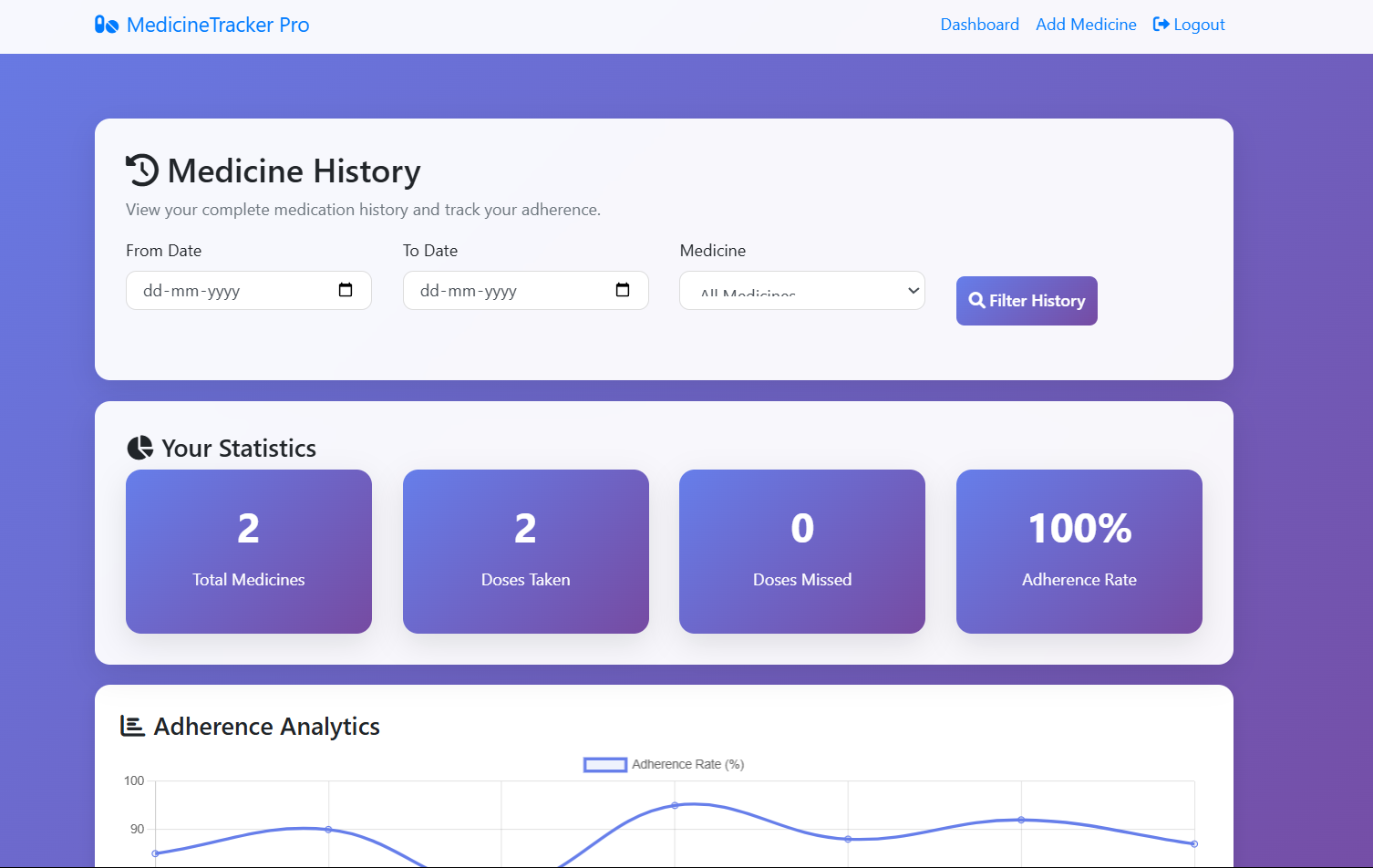
i) Deployed website snapshots:

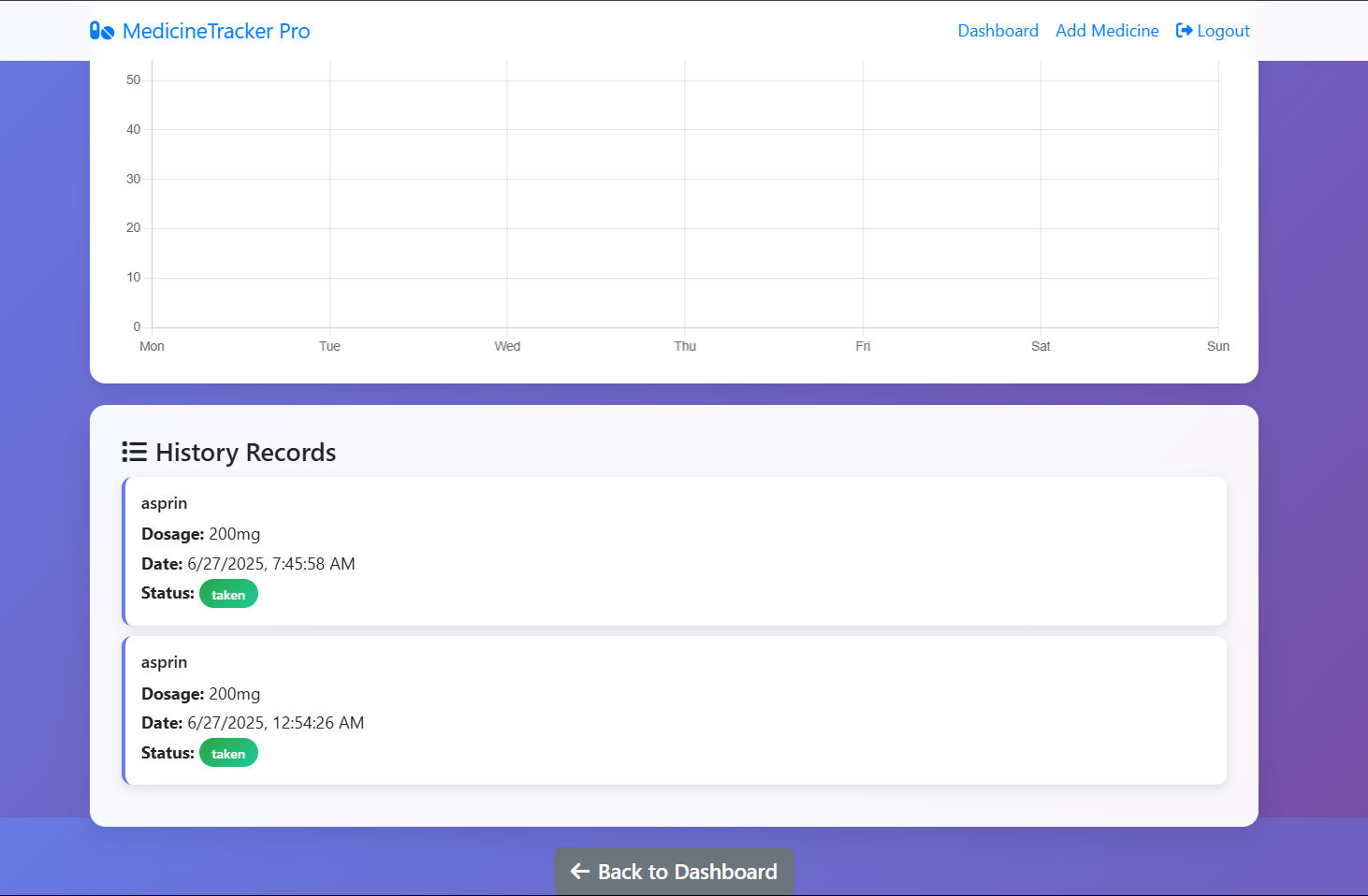
## 











## Summary

The CI/CD pipeline is designed to automate the complete lifecycle of the application—from code retrieval to deployment and verification. It begins with fetching the latest code from GitHub and executing automated tests with coverage analysis. Code quality and security are enforced through static analysis using SonarQube, Semgrep, and secret detection via Gitleaks. Shell scripts are linted for safety, and the application is containerized using Docker. The resulting image is scanned for vulnerabilities using Trivy and Grype, and its Software Bill of Materials (SBOM) is generated with Syft. Dependencies are further checked against known CVEs using OWASP Dependency-Check. After passing all checks, the Docker image is pushed to a registry and deployed to Railway. A final health check ensures that the live application is running correctly. This robust pipeline ensures code reliability, security, and consistent deployment to production.