PROJECT REPORT

COS40006-Software Deployment and Evolution

GROUP MEMBERS:
Pham Tuan Bach
Tran Hoang Duy Linh
Nguyen Hoang Nguyen

Table of Contents

1. Consultation Phase

1.1 Project Description

This project focuses on building an automated DevOps pipeline using three Azure Virtual Machines, each dedicated to a specific task. The first VM, referred to as JenkinsServer, handles continuous integration by automating the process of building and testing code. The second VM, SonarQubeServer, ensures code quality by performing static analysis to detect errors, vulnerabilities, and other quality issues. Finally, the ProductionServer hosts the deployed application in Docker containers, serves the application via Nginx, and incorporates Prometheus and Grafana for real-time monitoring.

The pipeline starts with changes made to the code repository on GitHub, which triggers the Jenkins pipeline. Jenkins clones the repository, analyzes the code with SonarQube, and deploys error-free code to the ProductionServer. Once deployed, Nginx serves the application while Prometheus gathers performance metrics. Grafana provides dashboards for visualizing these metrics, enabling real-time monitoring of application health and infrastructure performance. This design leverages the scalability of Azure VMs and integrates powerful DevOps tools to streamline the software development lifecycle.

1.2 Project Objectives

The primary objective of this project is to automate software development and deploy lifecycles to enhance efficiency, scalability, and reliability. By automating these processes, the pipeline eliminates the dependency on manual intervention, thereby accelerating deployment, reducing errors, and maintaining consistent quality standards across iterations. The use of Jenkins as the CI server ensures a streamlined build process, while SonarQube integrates robust static code analysis to establish quality gates that prevent the deployment of faulty, insecure, or non-compliant code, thus fostering secure and maintainable software.

Another critical objective is to implement a comprehensive and proactive monitoring solution for the deployed application. Prometheus collects a wide range of real-time metrics, including CPU usage, memory consumption, response times, disk I/O, and network traffic. These metrics are presented in intuitive Grafana dashboards, enabling stakeholders to easily monitor system performance. The monitoring system not only provides actionable insights but also helps in proactively addressing performance bottlenecks and ensuring the stability and reliability of the application in production. Additionally, this setup ensures scalability and adaptability to support larger workloads or additional applications in the future.

1.3 Problem Statement

Software deployment in traditional environments often involves manual processes that are prone to inefficiencies, errors, and delays. As the complexity of software systems grows, these manual interventions can lead to bottlenecks in the deployment cycle, adversely impacting product quality and delivery timelines. Key challenges include difficulty in identifying issues early in the software lifecycle, reliance on error-prone manual reviews, and delayed feedback loops. These challenges are particularly pronounced in agile and continuous delivery frameworks, where rapid iterations and quick feedback are critical.

Furthermore, manual deployment practices make it difficult to maintain consistent quality and security standards. Vulnerabilities or misconfigurations introduced during deployment can remain undetected until they manifest as production incidents. The lack of automation hinders the scalability and responsiveness of DevOps teams, undermining overall organizational agility.

The absence of robust monitoring mechanisms compounds these challenges. Reactive approaches to performance and health monitoring lead to delayed identification of system bottlenecks, resulting in extended downtimes and poor user experiences. Integrating automated monitoring systems within the DevOps ecosystem ensures proactive identification of performance issues and enhances system reliability.

By integrating tools such as Jenkins for continuous integration, SonarQube for automated code quality analysis, and Docker for containerized deployments, this project addresses the inefficiencies inherent in manual deployment. Additionally, Prometheus and Grafana provide a proactive monitoring framework, enabling teams to visualize and act on real-time performance metrics, which is critical for maintaining high service availability. The automation and monitoring solution aligns with industry's best practices and ensures a scalable, reliable deployment pipeline.

2. Preliminary Design Concepts

The preliminary design for this DevOps pipeline project centers around creating a fully automated continuous integration and deployment (CI/CD) pipeline with integrated code quality analysis and monitoring capabilities. The design follows modern DevOps practices and emphasizes automation, quality assurance, and observability.

2.1. System Architecture

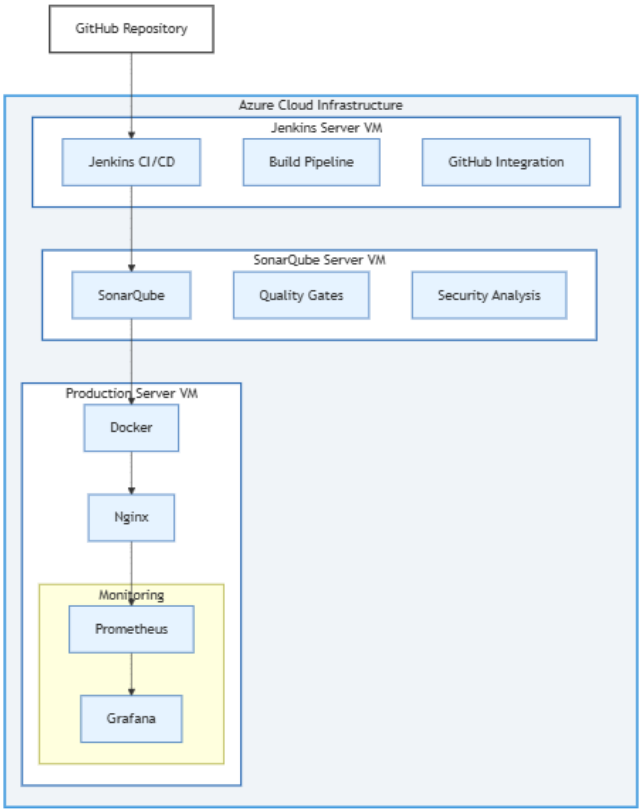


Figure 1: System Architecture design

The system architecture illustrates the three main components hosted on separate Azure Virtual Machines, showing their relationships and primary functions. Each VM serves a specific purpose in the pipeline:

1. Jenkins Server (CI/Build Server)

- Main component for CI/CD pipeline

- Manages source code from GitHub
- Coordinates build processes
- Triggers following pipeline stages

2. SonarQube Server (Test/Quality Server)

- Performs continuous code inspection
- Enforces quality gates
- Conducts security analysis
- Reports code quality metrics

3. Production Server

- Runs containerized applications
- Handles web serving through Nginx
- Collects and visualizes metrics
- Manages application performance

2.2. Design and Development

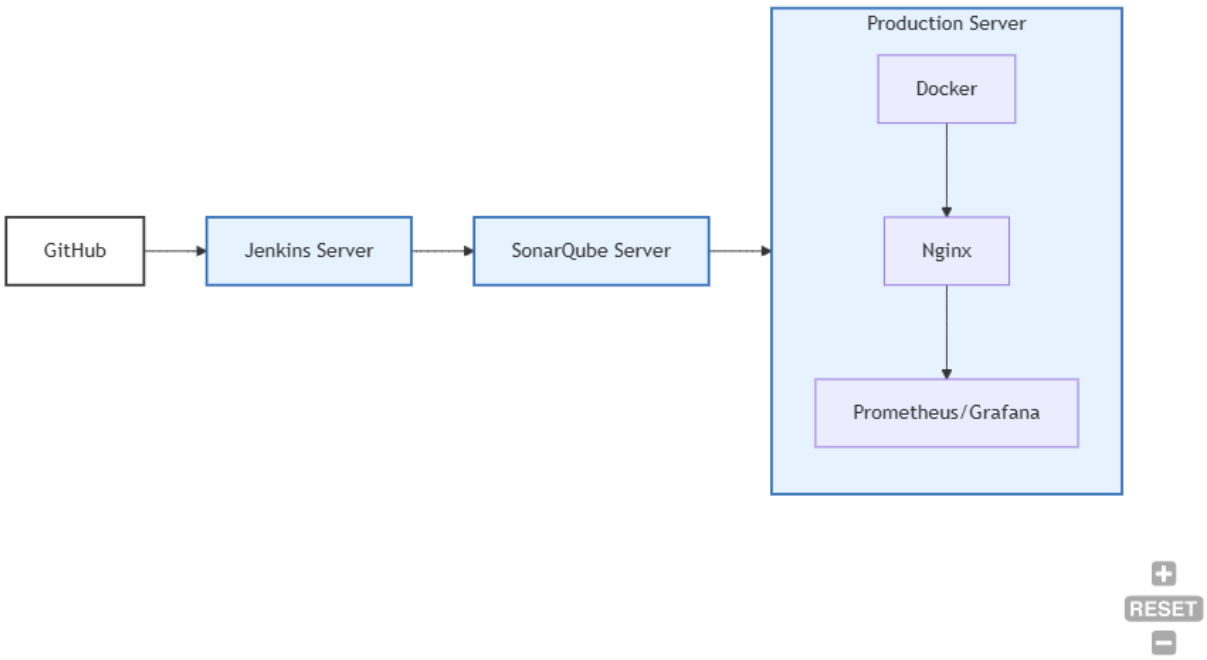


Figure 2: Workflow design

Pipeline Workflow Design

1. Source Code Management

- GitHub repository integration
- Webhook configuration for automatic pipeline triggering
- Branch management strategy for feature development and main releases

2. Continuous Integration Process

- Automated build triggering on code commits
- Jenkins pipeline configuration using Jenkinsfile
- Integration with SonarQube for code analysis

3. Quality Assurance Integration

- SonarQube quality gates configuration

- Code coverage requirements
- Security vulnerability scanning
- Quality metrics dashboard setup

4. Deployment Strategy

- Docker container build process
- Nginx configuration for application serving
- Container orchestration and management
- Zero-downtime deployment approach

5. Monitoring Solution

- Prometheus metrics collection
- Grafana dashboard configuration
- Performance metrics tracking

Workflow explanation

The DevOps pipeline follows a straightforward automated workflow across three Azure Virtual Machines. It starts when developers push code to GitHub, which triggers the Jenkins pipeline on the Jenkins Server VM. Jenkins pulls the code and runs the build process with initial testing. The code then moves to the SonarQube Server VM, where it undergoes quality and security checks. If the code passes these checks, it proceeds with the Production Server VM; if it fails, the pipeline is aborted. On the Production Server, Docker creates a container for the application, which is then deployed through Nginx for public access. Once deployed, Prometheus and Grafana continuously monitor the application's performance and health. This setup ensures that code moves from development to production in a controlled, automated way while maintaining quality and providing monitoring capabilities.

Technical Components

1. Jenkins Configuration

- Pipeline as Code implementation
- Multi-stage pipeline definition
- Integration with Azure VM infrastructure

- Credential management and security setup

2. SonarQube Setup

- Quality profiles configuration
- Custom rule definitions
- Project key management
- Integration with Jenkins pipeline

3. Docker Implementation

- Custom Dockerfile creation
- Multi-stage builds for optimization
- Container security considerations
- Image versioning strategy

4. Nginx Configuration

- Reverse proxy setup

5. Monitoring Stack

- Prometheus data collection endpoints
- Custom metrics definition
- Grafana visualization panels

3. Compatibility of Design

3.1. Specifications

The table below lists all the tools and technologies used in the pipeline, along with their roles, configurations, and integration details.

Component	**Specifications**
JenkinsServer (CI Server)	- **OS:** Ubuntu 24.04 on Azure VM

Component	Specifications
	- **Version:** Jenkins 2.479.1 - **Plugins:** Git Plugin, SonarQube Scanner, Docker Plugin - **Purpose:** Automates code integration, builds, and deployment - **Integration:** Triggered by GitHub webhooks, integrates with SonarQube for quality checks
SonarQubeServer (Code Analysis)	- **OS:** Ubuntu 24.04 on Azure VM - **Version:** SonarQube 9.9 - **Purpose:** Conducts static code analysis to identify bugs, vulnerabilities, and code smells - **Integration:** Works with Jenkins for automated quality analysis
ProductionServer (Hosting)	- **OS:** Ubuntu 24.04 on Azure VM - **Tools:** Docker 27.3.1, Docker Compose 1.29.2, Nginx 1.21.0 - **Purpose:** Hosts containerized applications via Nginx - **Features:** Supports SSL encryption for secure communication
Prometheus (Monitoring)	- **Version:** Prometheus 3.0 - **Purpose:** Collects time-series data on infrastructure and application performance - **Integration:** Monitors server resources and Dockerized applications
Grafana (Visualization)	- **Version:** Grafana 9.3.1 - **Purpose:** Visualizes performance metrics from Prometheus-**Features:** Customizable dashboards, supports real-time data monitoring and alerting systems

3.2. Data Display

The data display for the project was facilitated using Grafana dashboards, which provided real-time visualization of system and container metrics. These dashboards, integrated with Prometheus, ensured comprehensive monitoring of the servers and Docker containers. The following metrics were displayed:

3.2.1. System Metrics

- **Uptime:** Displayed the duration for which each server and container had been running, helping track system reliability.
- **Disk Space:** Showed used and available disk space across servers, ensuring that storage constraints could be anticipated and managed effectively.
- **Memory Usage:** Visualized the total and available memory on each server, providing insights into resource consumption.
- **Swap Usage:** Monitored the amount of swap memory being used, helping detect potential memory pressure on the system.
- **Load Average:** Tracked the system load over different intervals, highlighting the overall utilization and responsiveness of the servers.
- **CPU Usage:** Displayed the percentage of CPU resources consumed by each server, helping monitor processing loads.
- **Disk I/O:** Provided insights into the read and write operations on the disk, useful for identifying storage bottlenecks.
- **Network Traffic:** Monitored incoming and outgoing network traffic, helping detect unusual patterns or excessive bandwidth usage.

3.2.2. Container Metrics

- **Container Health:** Tracked the status of all active Docker containers, including uptime and performance.
- **Received Network Traffic per Container:** Displayed the amount of data received by each container, enabling granular monitoring of network usage.
- **Sent Network Traffic per Container:** Showed the volume of data transmitted by each container, identifying containers with high network demand.

- **CPU Usage per Container:** Visualized the percentage of CPU resources consumed by individual containers, allowing efficient resource allocation.
- **Memory Usage per Container:** Tracked memory consumption for each container, helping detect containers with high memory demands.
- **Memory Swap per Container:** Monitored the swap memory used by containers, indicating if additional memory resources were required.

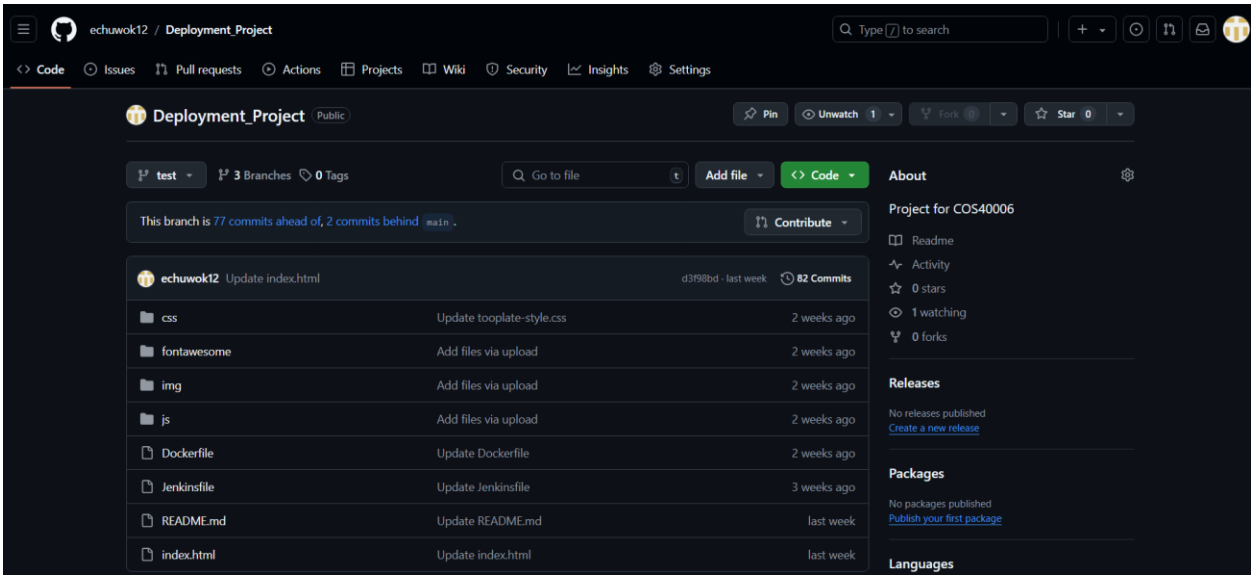
3.2.3. Dashboard Features

- **Customizable Time Filters:** Enabled users to view metrics over specific time periods, such as the last hour, day, or week, providing flexibility in analysis.
- **Real-Time Updates:** Data was refreshed in real-time to provide up-to-date monitoring of the system and containers.
- **Alerts and Notifications (optional):** Alerts were configured for critical metrics like high CPU usage, low available memory, or excessive disk I/O, ensuring timely issue resolution.

3.3. Code explanation

3.3.1. Github repository setup:

- The Github page will store 3 main components: the website code, the Dockerfile file to create a suitable environment in the Production server, and the Jenkinsfile ile to automate all the pipeline tasks.



- The project can be accessed via the link: [echuwok12/Deployment_Project at test]

3.3.2. Jenkin Server setup:

- Open Azure. Set up a server in Azure Virtual Machine with the following settings:
  - OS: Linux
  - VM architecture: x64
  - Size: Standard B1ms
  - vCPUs: 1
  - RAM: 2Gb
  - Source image: ubuntu
  - Public IP: 20.2.218.210

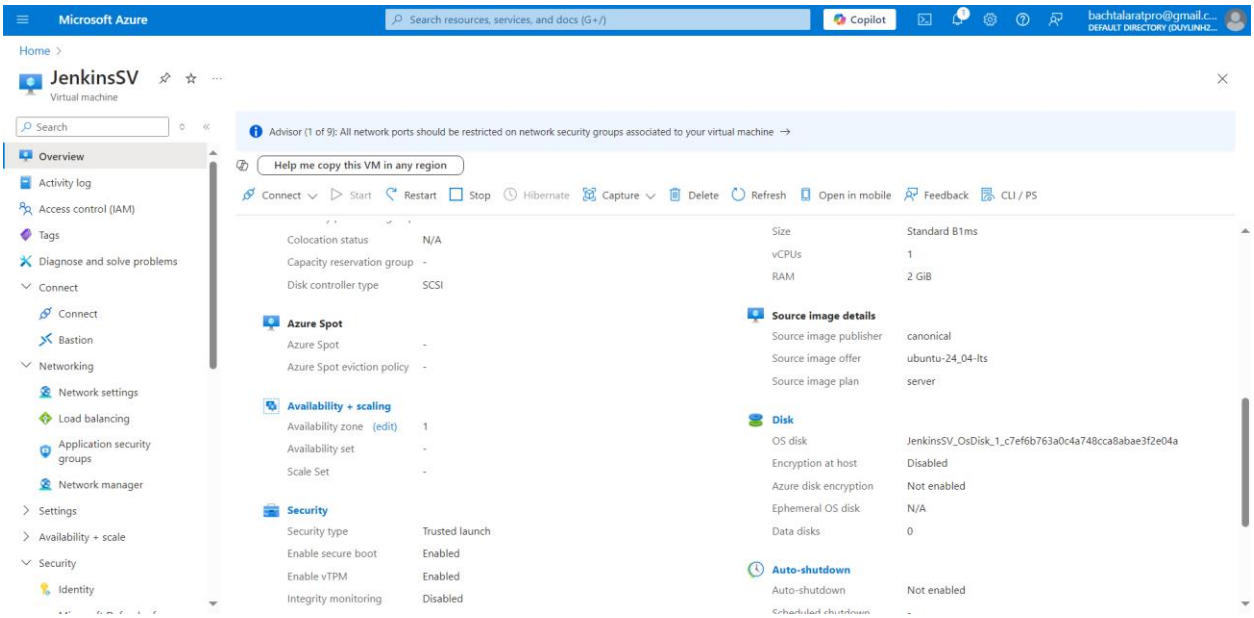


Figure 1: JenkinsSV set up

- Add an 8080 inbound port for Jenkin security rule

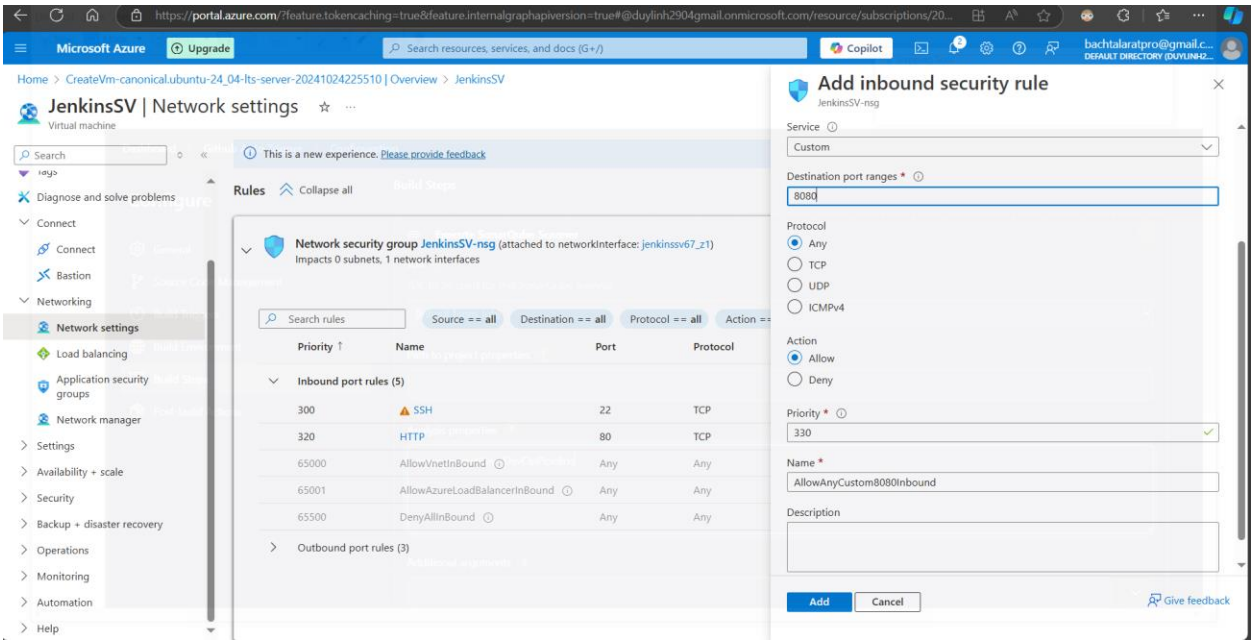


Figure 1.2: Port for Jenkin

- Go to 20.2.218.210:8080 to access the Jenkin page. First get the password from the link and paste it into the box.

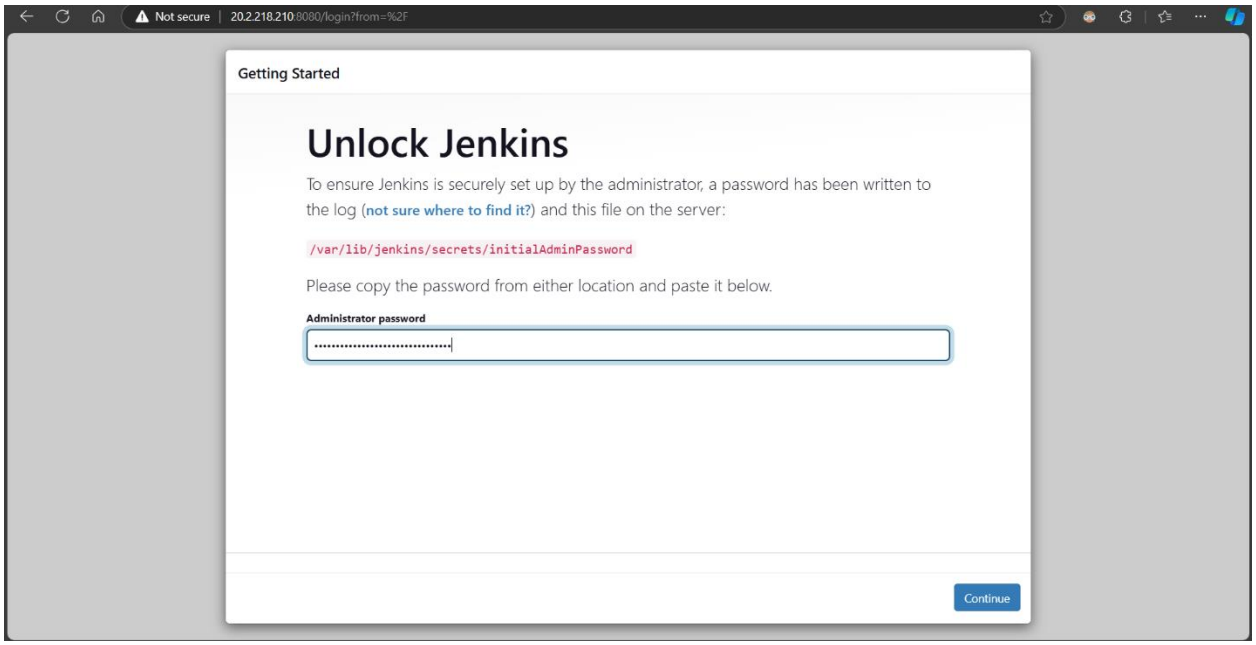


Figure 1.3: Unlock Jenkin

- Create an admin account to manage Jenkin

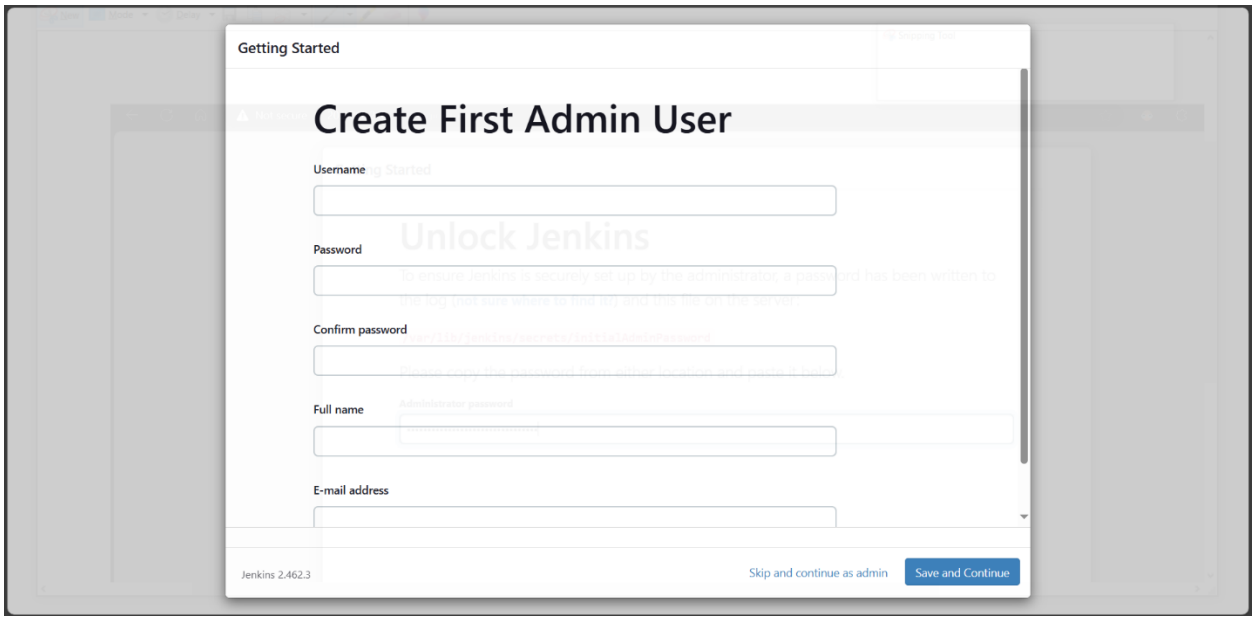


Figure 1.4: Create an admin account

- Access the Jenkin page

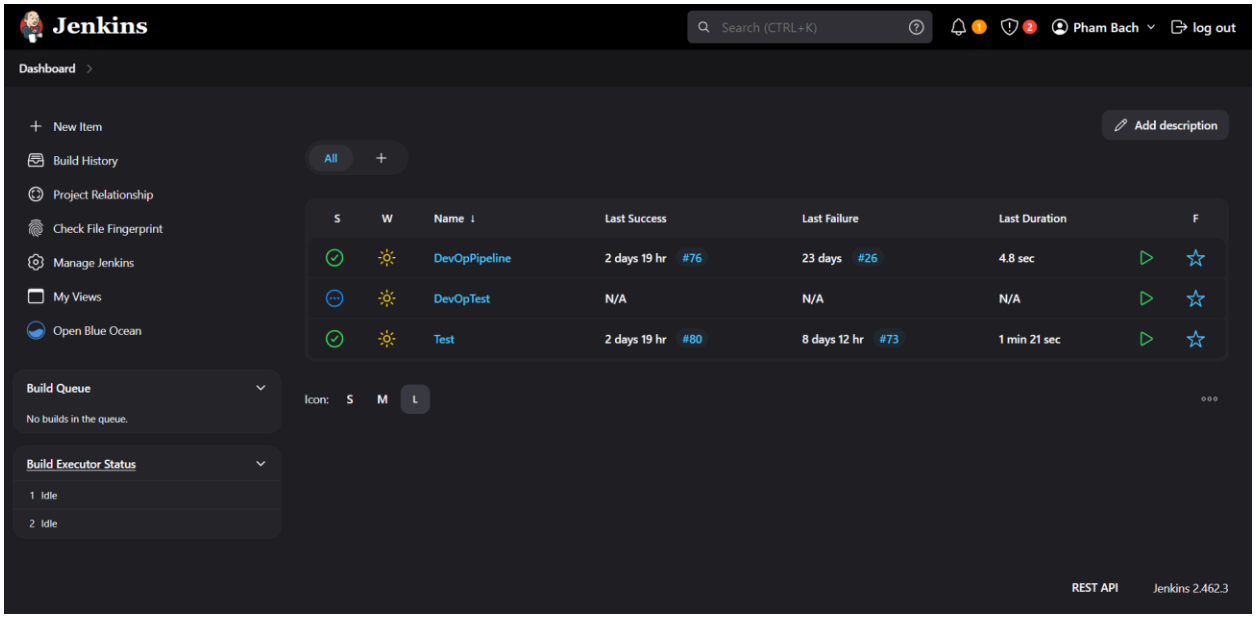


Figure 1.5: Jenkin page

3.3.3. Test Server setup

- Create an Azure VM instance to host the Sonarqube for testing purposes with the following settings:

- Computer name: SonarQubeServer
- Operating system: Linux
- VM architecture: x64
- Size: Standard B2s
- vCPUs: 2
- RAM: 4Gb
- Source image: ubuntu
- Public IP: 20.255.48.4

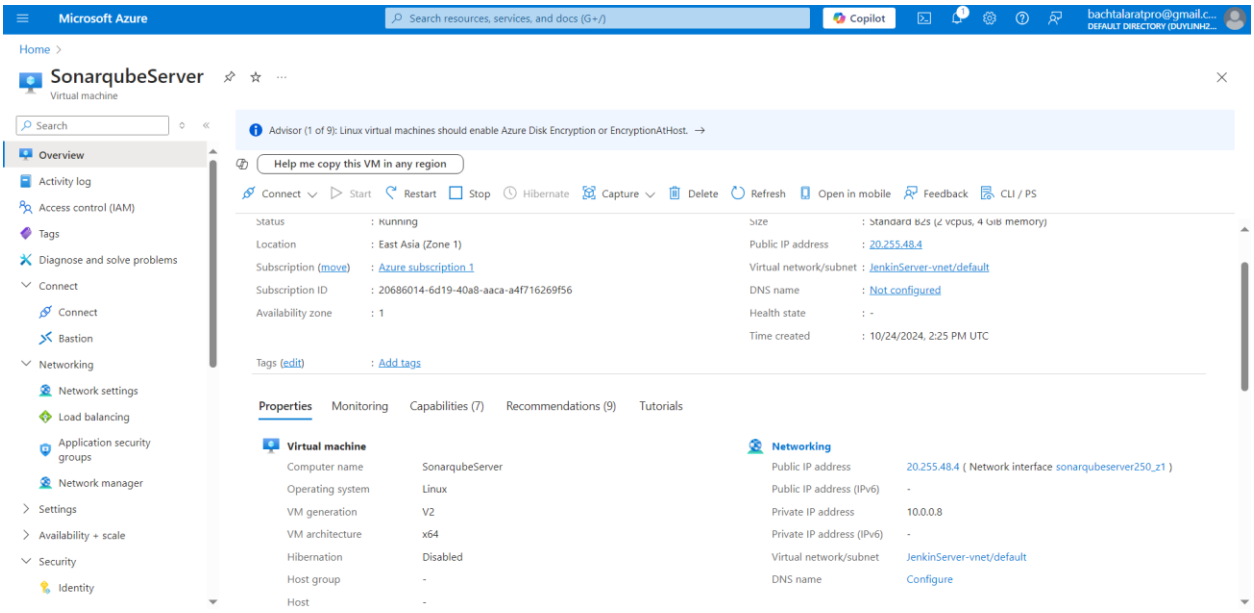


Figure 2.1: SonaQube server setup

- Go to Sonarsource.com and copy the link to download the community edition. The downloaded version is 10.7.0.96327.

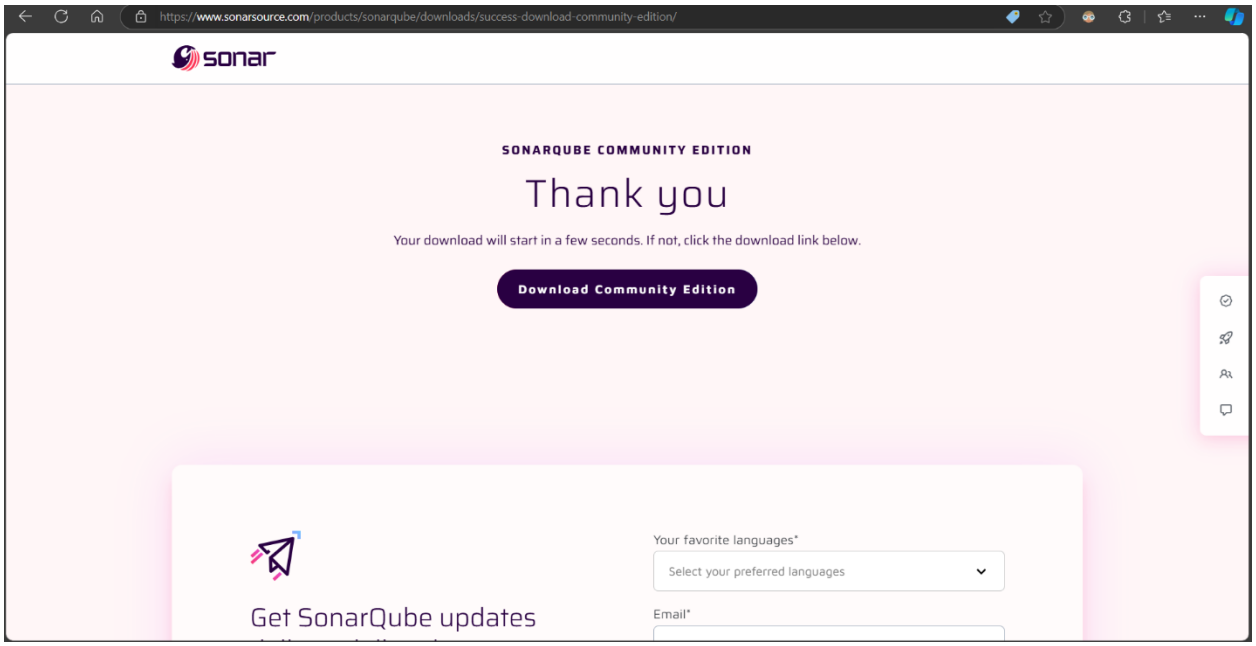


Figure 2.2: SonarQube download page

- Create a 9000 port for inbound security rule to host the SonarQube service.

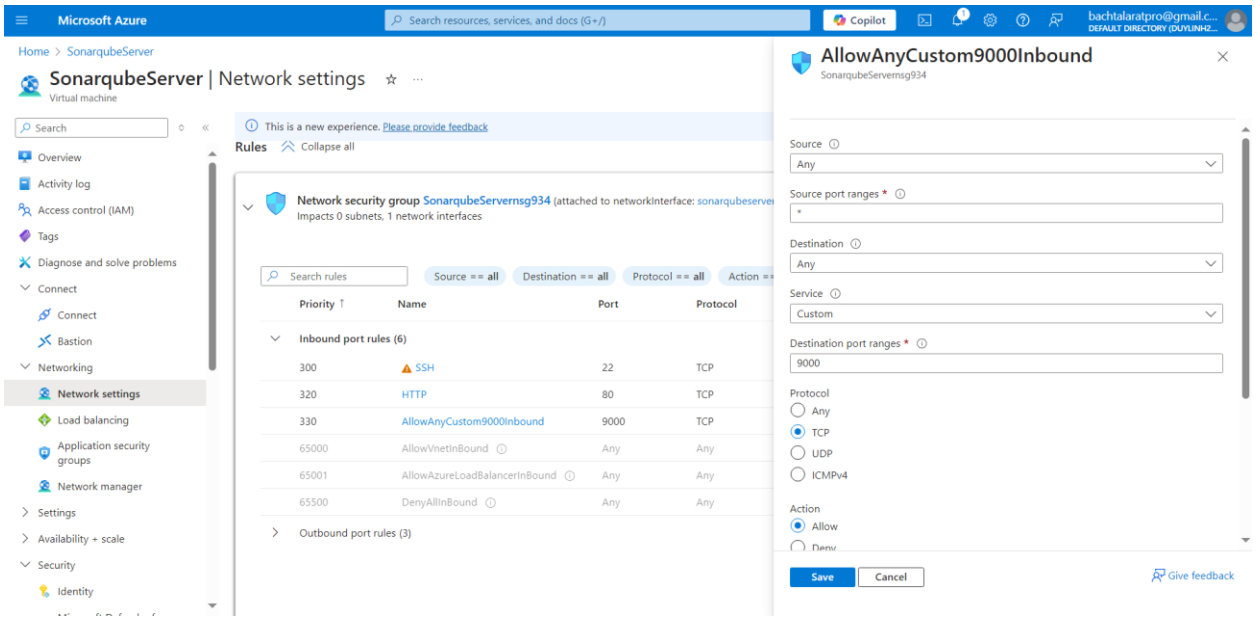


Figure 2.3: Adding port for SonarQube

- Connect to the server by “SSH using Azure CLI”. Download the community edition with the command wget <download link>.

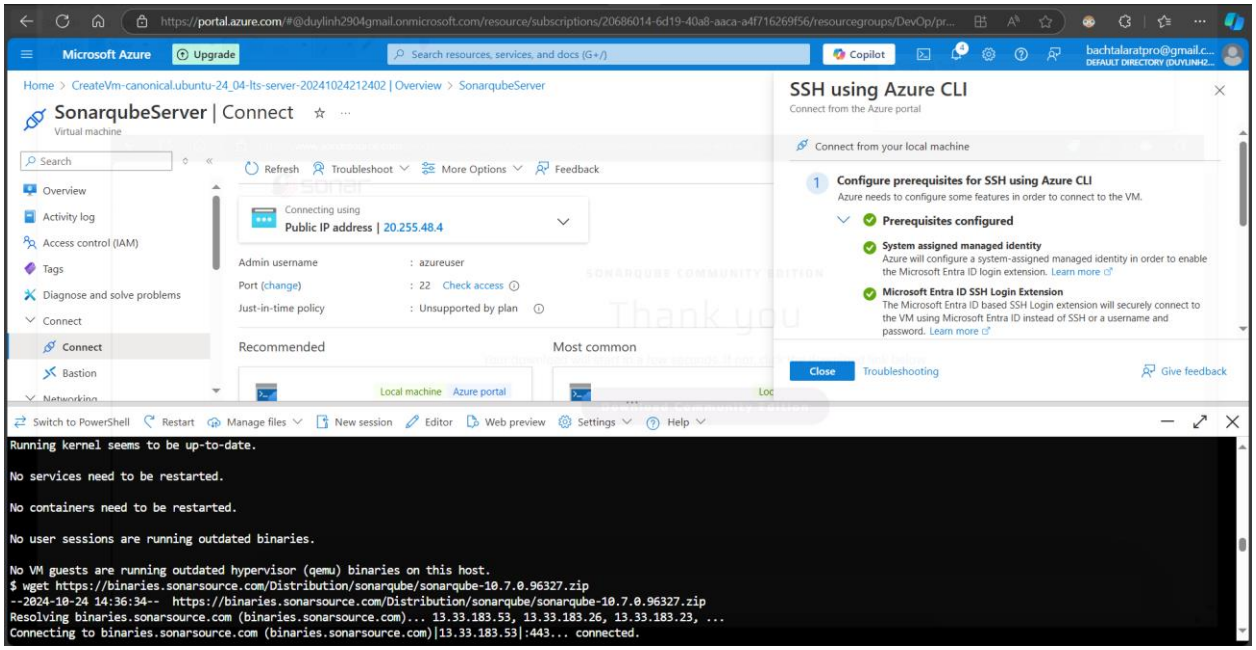


Figure 2.4: Download the SonarQube package

- Unzip the downloaded file.

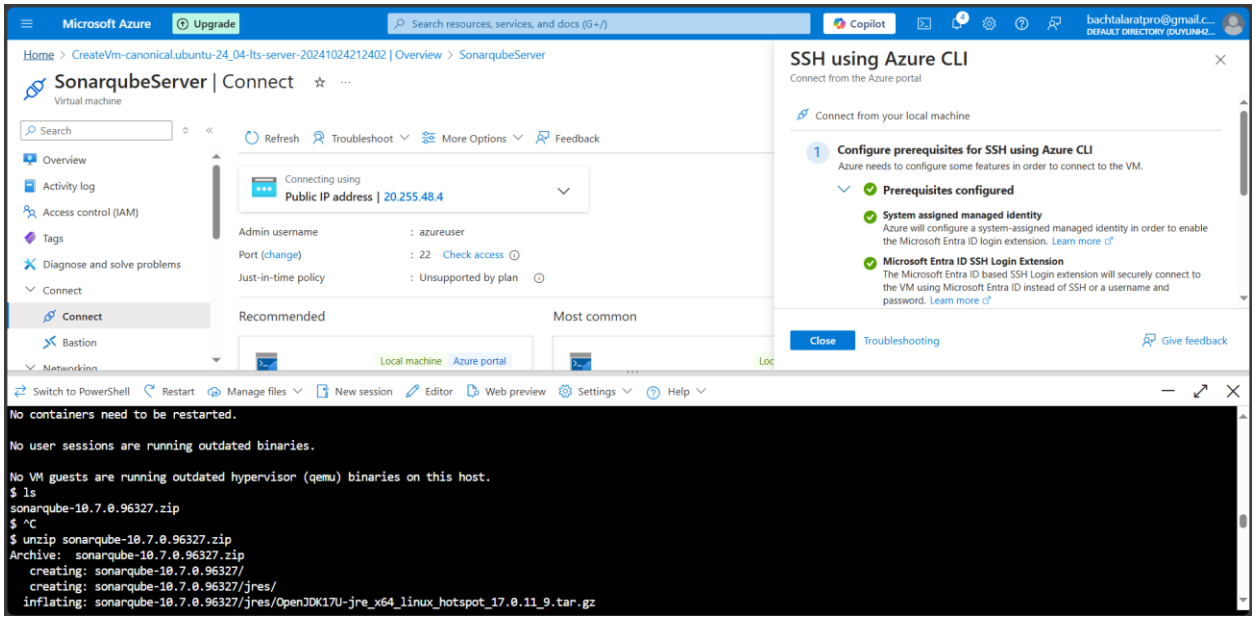


Figure 2.5: Unzip the SonarQube package

- Start the SonarQube server by going to /bin/linux-x86-64/ and run the command: *./sonar.sh console*

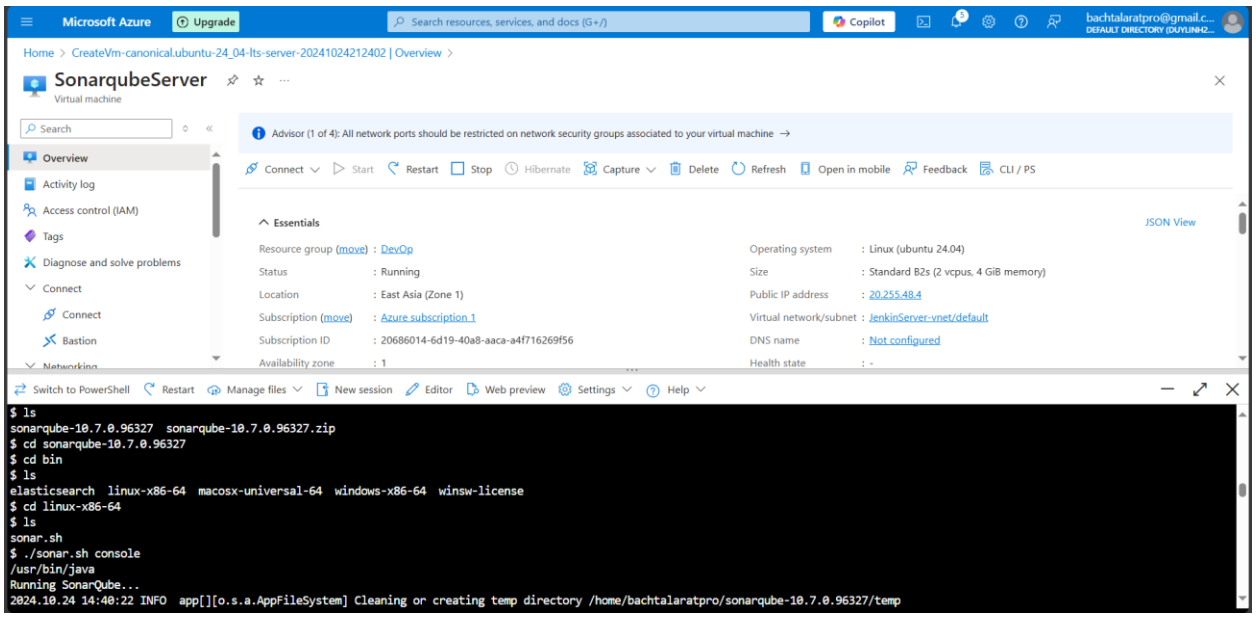


Figure 2.6: Start the SonarQube server

- After the service is up, SonarQube can be checked via 20.255.48.4:9000. From there, login using “admin” as both username and password.

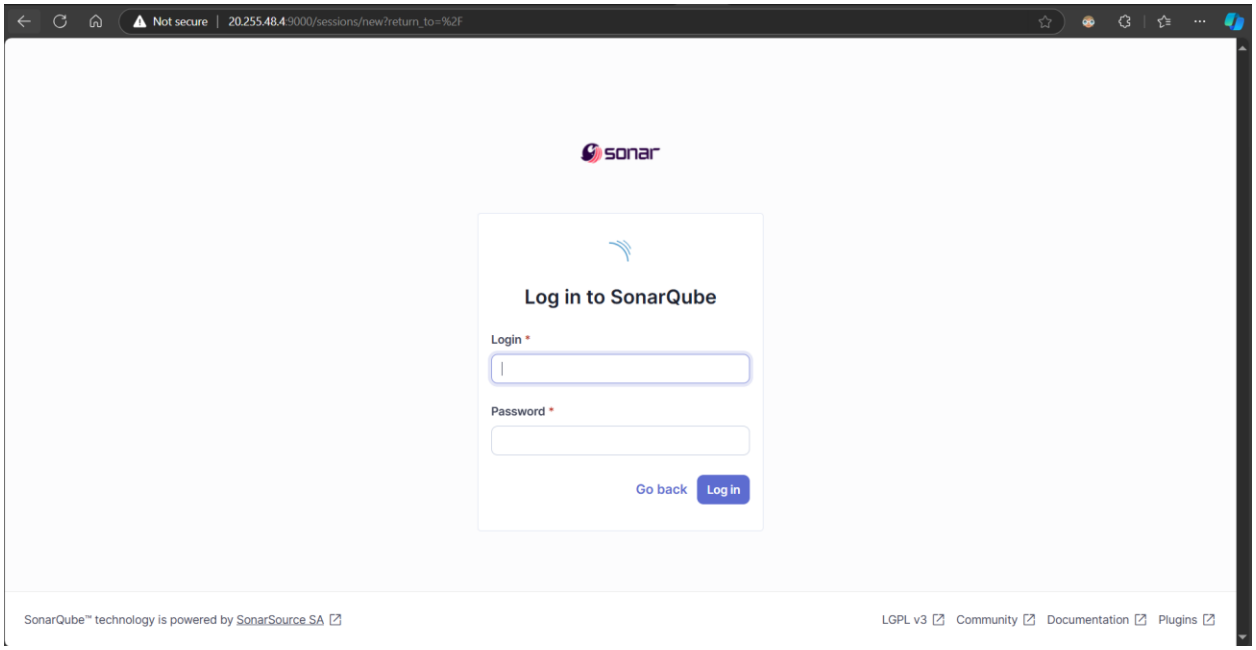


Figure 2.7: SonarQube first login

- After logging in, create a project to run tests for the website. Set the project name and project key as “DevOpPipeline”

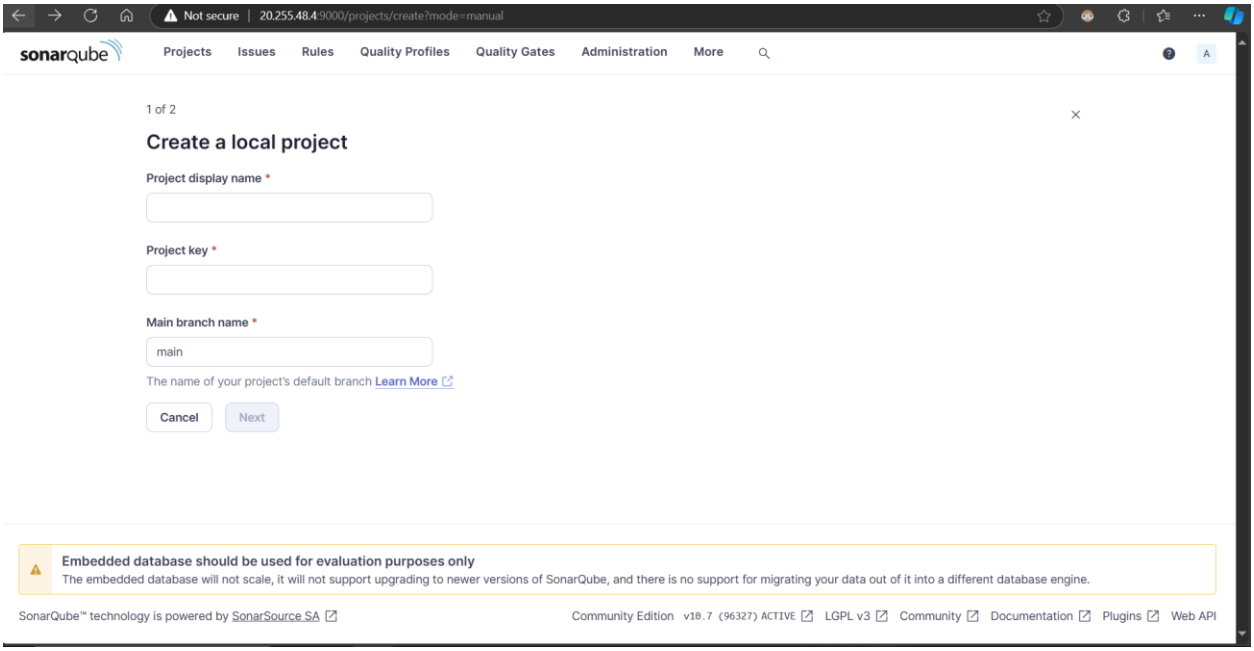


Figure 2.8: Setting up the test project

- Go to Project setting to create a web hook to trigger the test server from Jenkin.

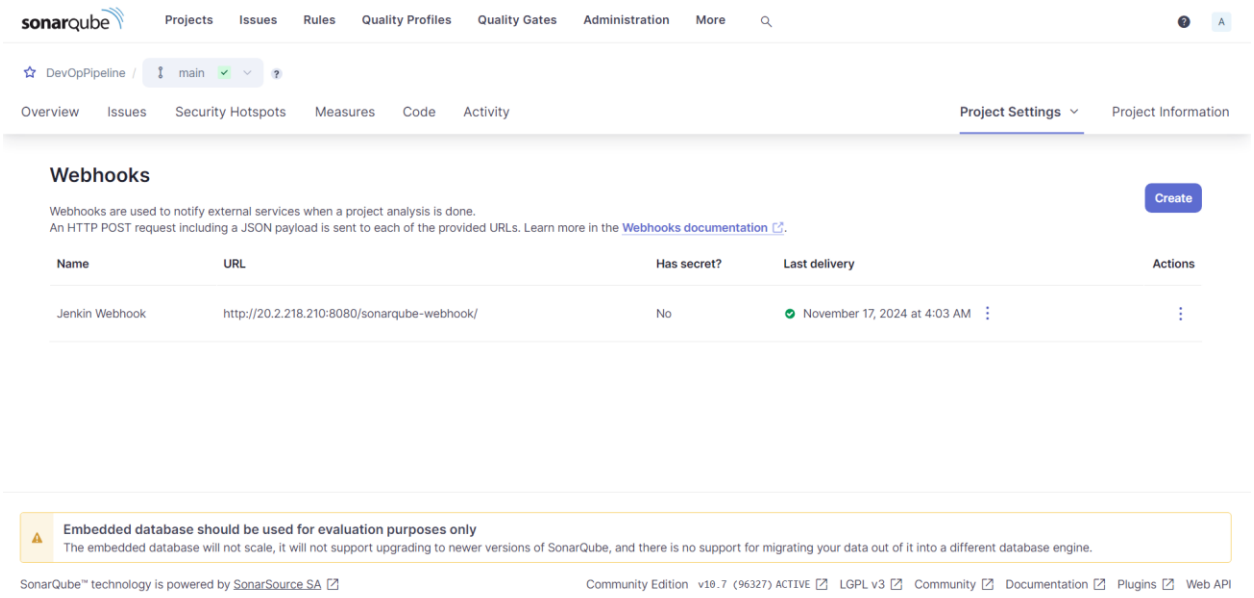


Figure 2.9: Jenkin web hook created

- Now back to Jenkins server, from Dashboard, go to Manage Jenkins/Plugins. Download the SonarQube Scanner to integrate with the Jenkins web hook from the test server.

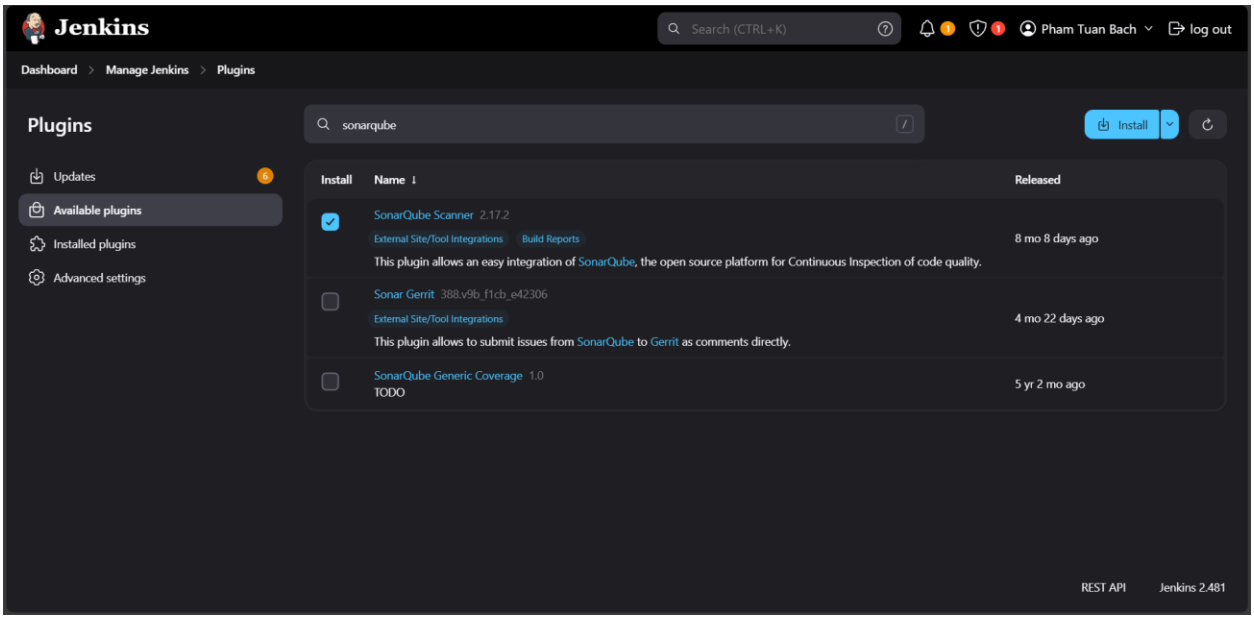


Figure 2.10: SonarQube Scanner downloaded

- Back to the test server, go to Security/Generate Tokens to create a new token to enforce security on the ssh connection to the Jenkins server. Without providing this token, the ssh connection between the test server and the other 2 ones cannot be done.

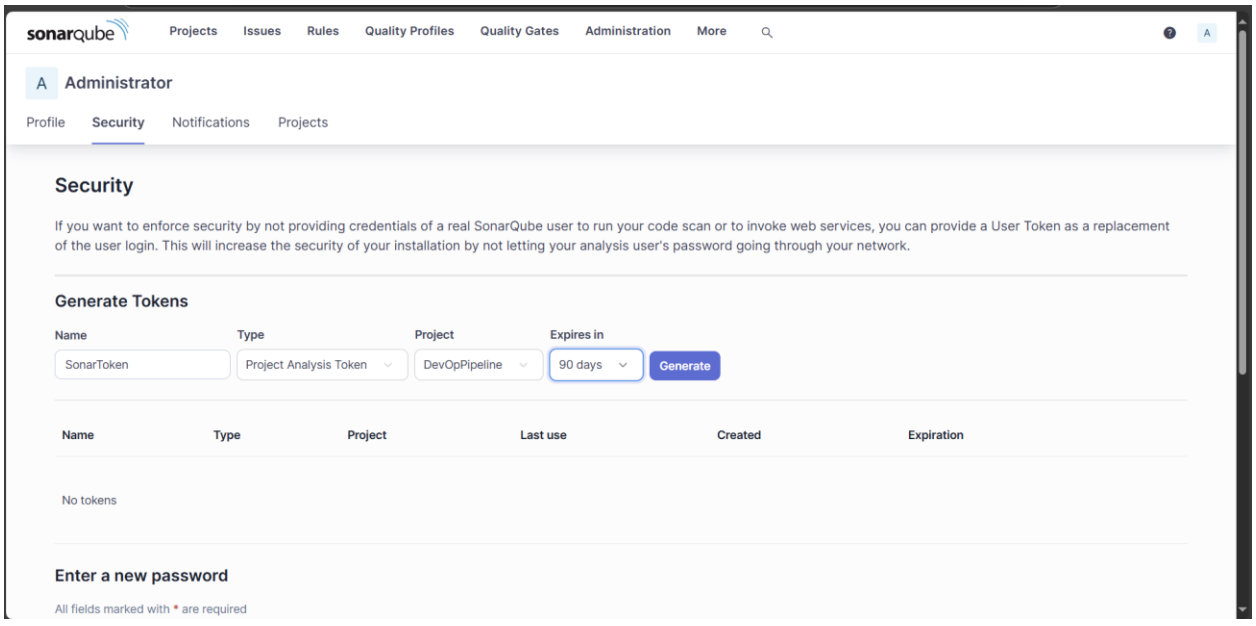


Figure 2.11: Generate a token for secure connection

- Next, go back to Jenkin server. Go to Manage/Configure Tools/. Scroll down to the SonarQube scanner installation section. Provide the info of the test server so that it doesn't require the token every time the pipeline runs.

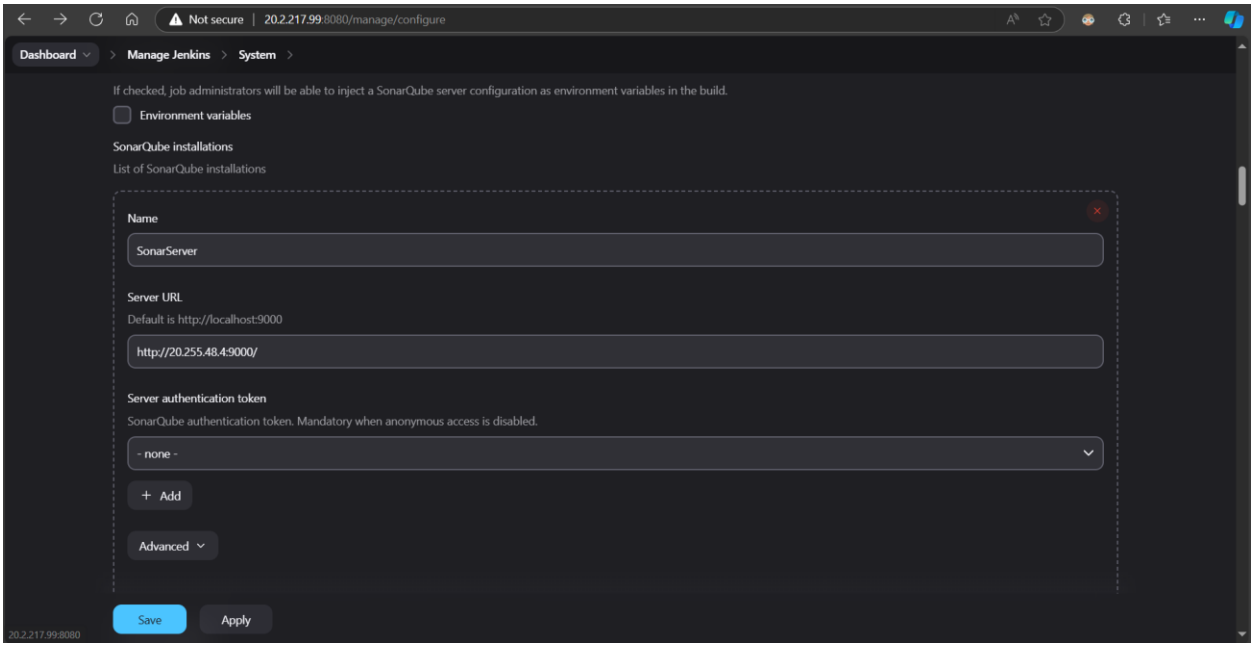


Figure 2.12: Provided connection for SonarQube from/to Jenkin

- The setup for SonarQube server is complete.

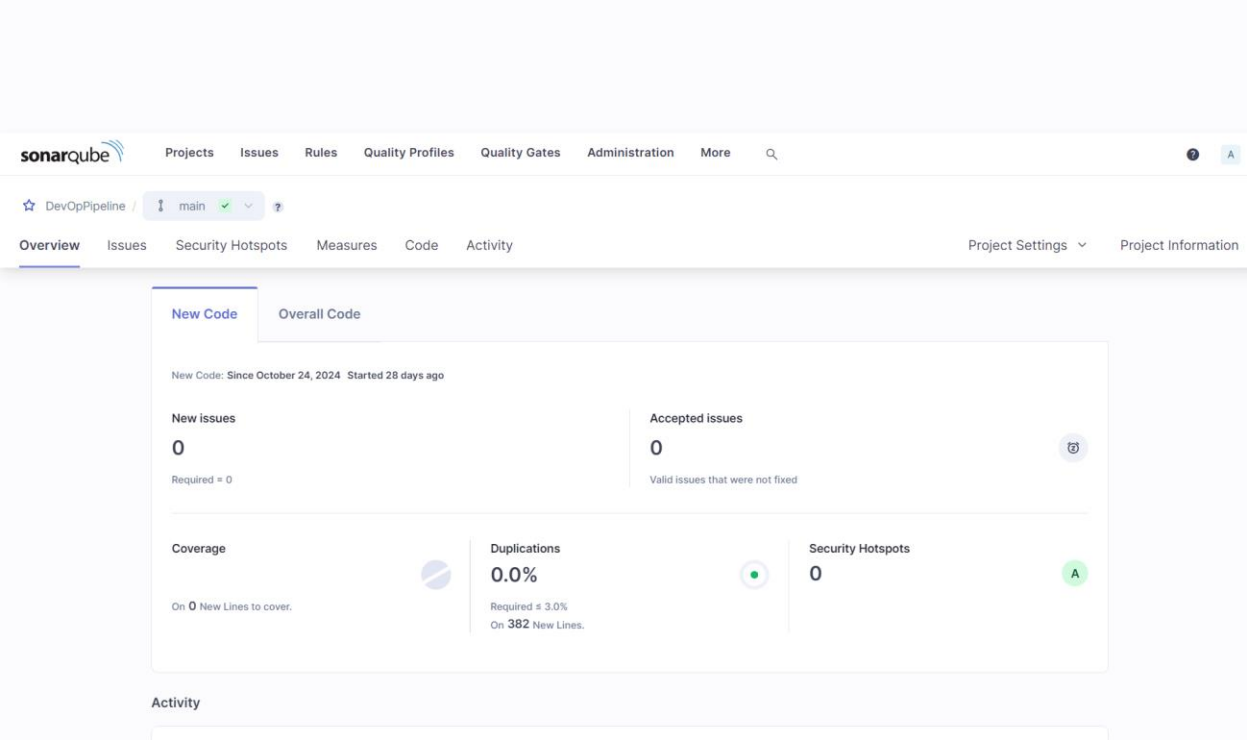


Figure 2.13: SonarQube Project complete setup

3.4. Production server setup

- Create an Azure VM to host the Production server. It has the same system settings as the Jenkins Server.

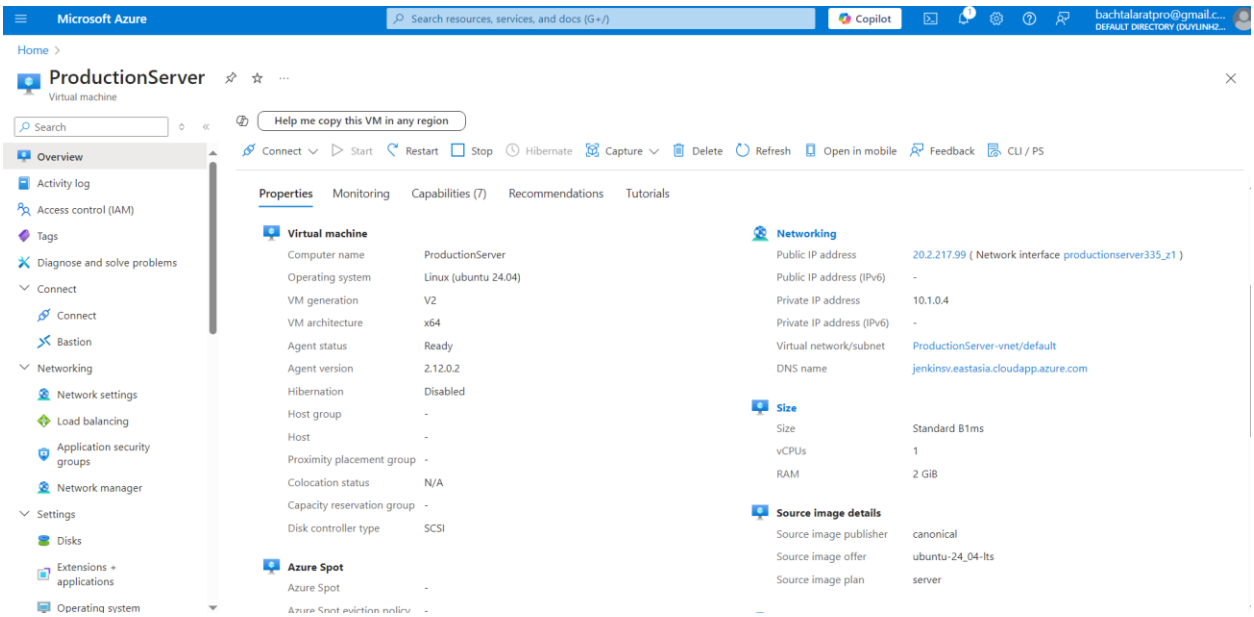


Figure 3.1: Production Sever settings

- Connect to the server terminal by using SSH using Azure CLI
- First, turn on the privileged user mode in order to set up the services easier using the command:

  sudo su

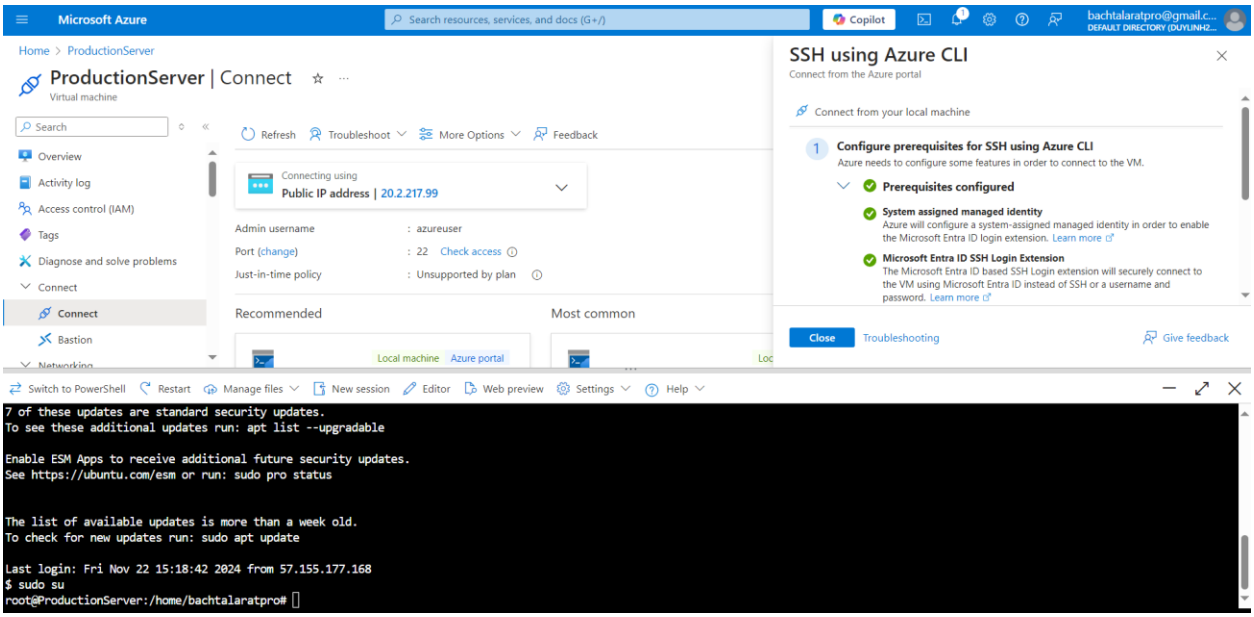


Figure 2.2: Turn on privileged user mode

- Now, install Docker Engine in the production server, using the following command:

  # Add Docker's official GPG key:
  sudo apt-get update
  sudo apt-get install ca-certificates curl
  sudo install -m 0755 -d /etc/apt/keyrings
  sudo curl -fsSL https://download.docker.com/linux/ubuntu/gpg -o /etc/apt/keyrings/docker.asc
  sudo chmod a+r /etc/apt/keyrings/docker.asc

  # Add the repository to Apt sources:
  echo \
  "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.asc]
  https://download.docker.com/linux/ubuntu \
  $(. /etc/os-release && echo "$VERSION_CODENAME") stable" | \
  sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
  sudo apt-get update

- Next, run the following command to install the bundles:

sudo apt-get install docker-ce docker-ce-cli containerd.io docker-buildx-plugin docker-compose-plugin

- Next, use this command to create a container for the Docker image and verify that Ngnix can be run inside the container to host the website transferred from SonarQube server.

  docker exec new-container nginx -t

- Now, to export the metrics performance of the container to Prometheus and Grafana, first create a daemon.json file using the command:

  vi /etc/docker/daemon.json

- Then post the following content to the file:

```
{
  "metrics-addr" : "0.0.0.0:9323",
  "experimental" : true
}
```

- Save the file
- Next, to configure Prometheus to receive the exposed traffic sent from the deamon.json file, create a Prometheus.yml using the command:

  nano prometheus.yml

- Post the following content inside the file:

```
scrape_configs:
  - job_name: prometheus
    scrape_interval: 5s
    static_configs:
    - targets:
      - prometheus:9090
      - node-exporter:9100
      - pushgateway:9091
      - cadvisor:8080

  - job_name: docker
    scrape_interval: 5s
    static_configs:
    - targets:
      - 20.2.217.99:9323
```

  This setup enables Prometheus to collect data from all critical monitoring components and Docker.

- Save the file.
- Now create another file called docker-compose.yml using the command:
  nano docker-compose.yml

- Post the following content inside the file:

```
version: '3'
services:
  prometheus:
    image: prom/prometheus:latest
    container_name: prometheus
    ports:
      - 9090:9090
    command:
      - --config.file=/etc/prometheus/prometheus.yml
    volumes:
      - ./prometheus.yml:/etc/prometheus/prometheus.yml:ro
    depends_on:
      - cadvisor
  cadvisor:
    image: google/cadvisor:latest
    container_name: cadvisor
    ports:
      - 8080:8080
    volumes:
      - /:/rootfs:ro
      - /var/run:/var/run:rw
      - /sys:/sys:ro
      - /var/lib/docker/:/var/lib/docker:ro
  pushgateway:
    image: prom/pushgateway
    container_name: pushgateway
    ports:
      - 9091:9091
  node-exporter:
    image: prom/node-exporter:latest
    container_name: node-exporter
    restart: unless-stopped
    expose:
      - 9100
  grafana:
    image: grafana/grafana
    container_name: grafana
    ports:
      - 3000:3000
    environment:
      - GF_SECURITY_ADMIN_PASSWORD=123456
    depends_on:
      - prometheus
      - cadvisor
```

This Docker Compose file sets up a monitoring stack with Prometheus, Grafana, cAdvisor, Pushgateway, and Node Exporter. Prometheus collects and stores metrics from cAdvisor (container stats), Node Exporter (hardware/OS metrics), and Pushgateway (custom job metrics). Grafana visualizes the data with customizable dashboards. Each service is configured with appropriate Docker images, ports, and volumes for functionality. Prometheus uses a prometheus.yml file to define scrape

targets, intervals, and job configurations. Dependencies ensure services like Grafana start only after Prometheus and cAdvisor are running.

- Save the file
- Now download Docker Compose in order to run these files using the command:
  apt install docker-compose
  sudo apt update

- Now in the same directory, run the command to start the services used in the two above files:
  docker-compose up -d

- If the service configuration was correct, there would be successful notifications as seen in the terminal:

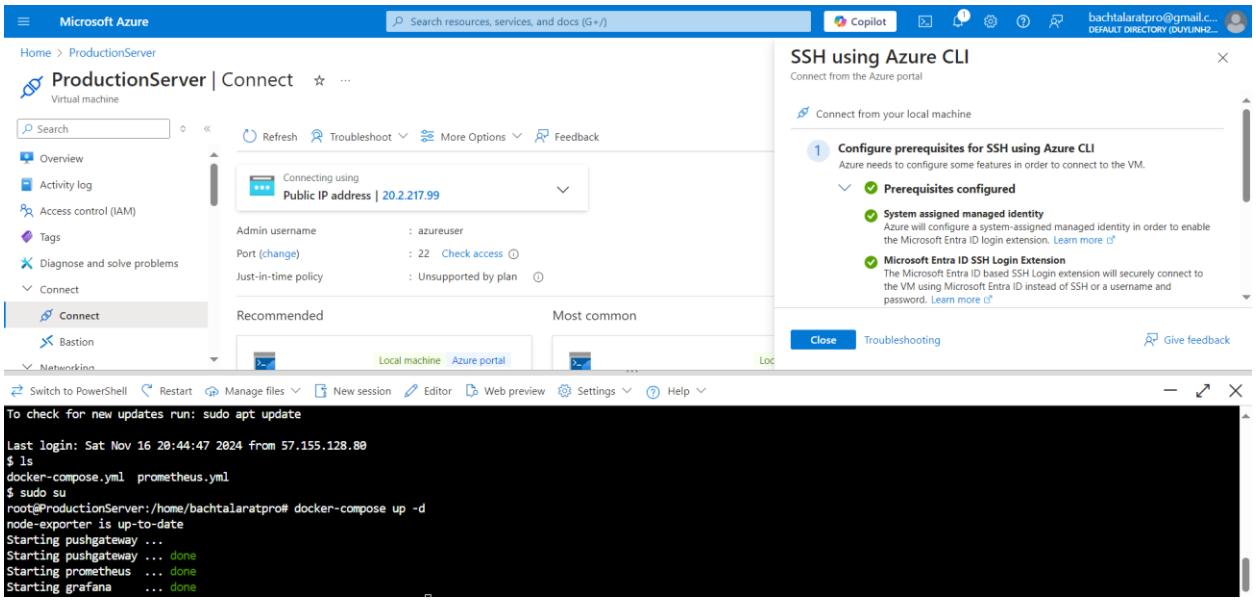


Figure 3.3: All services start successfully

- Next, add all the ports that are going to be used for the services in the inbound security rule: 9323, 3000, 9090, 8080.

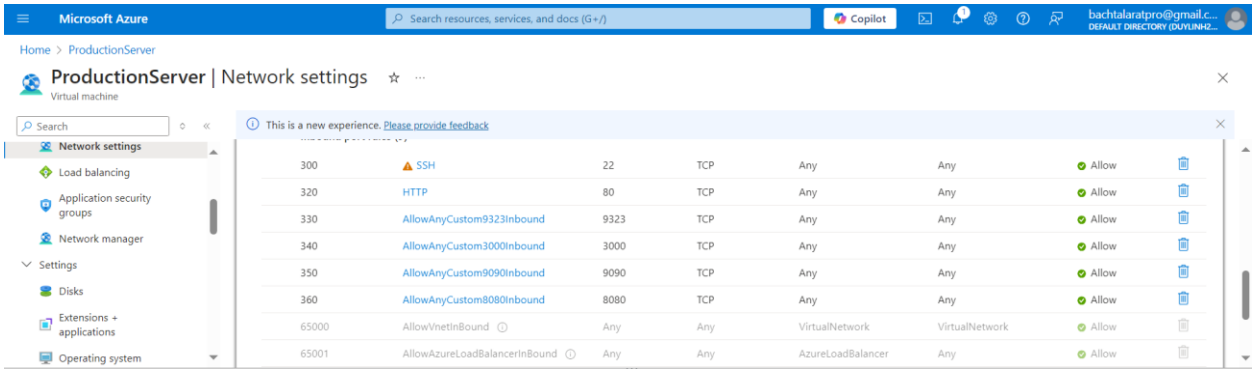


Figure 3.4: Add port for inbound security rule

- First, access the link http://20.2.217.99:9090/ to access Prometheus. From the interface below, click on Status/Targets to check whether all the needed metrics have been exported successfully to Prometheus

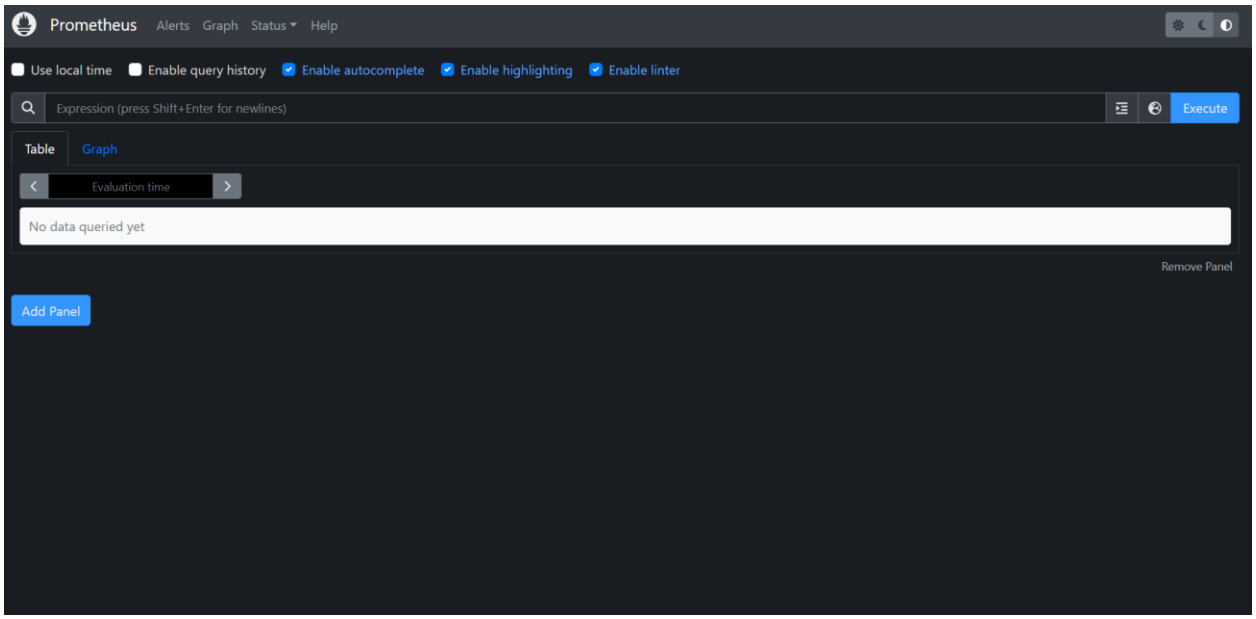


Figure 3.5: Prometheus main interface

- From the image below, all the metrics can be seen to be exported successfully.

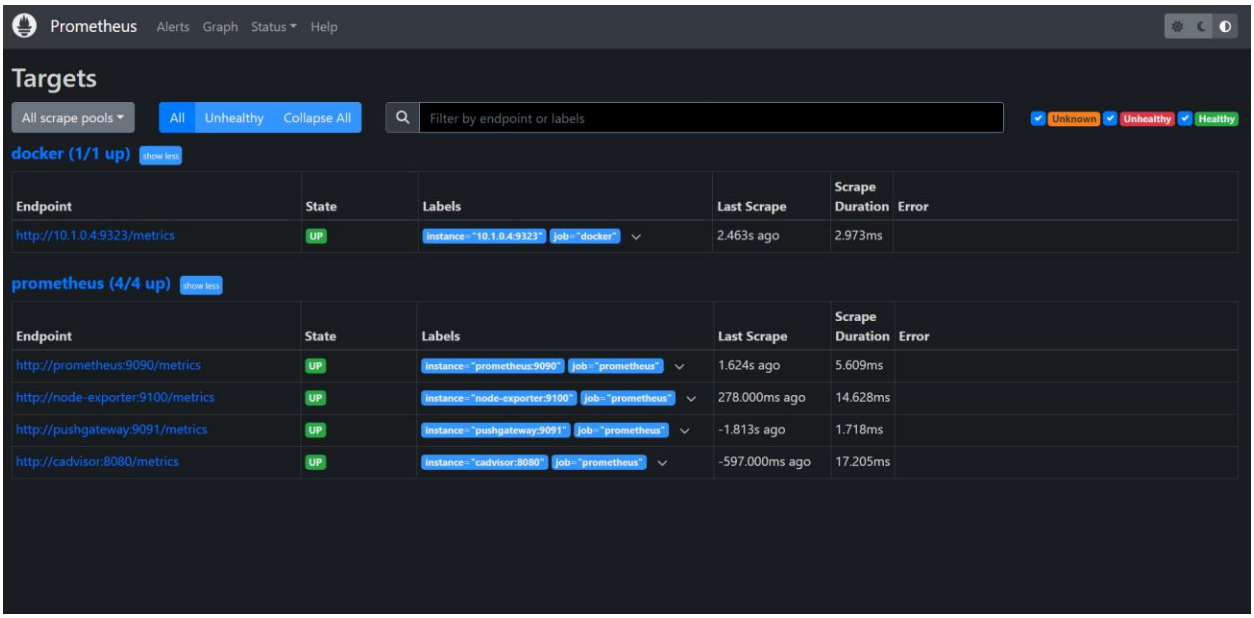


Figure 3.6: All metric exposed successfully

- Now open a new tab, access the link for http://20.2.217.99:3000/ to open Grafana. Click on Dashboard.

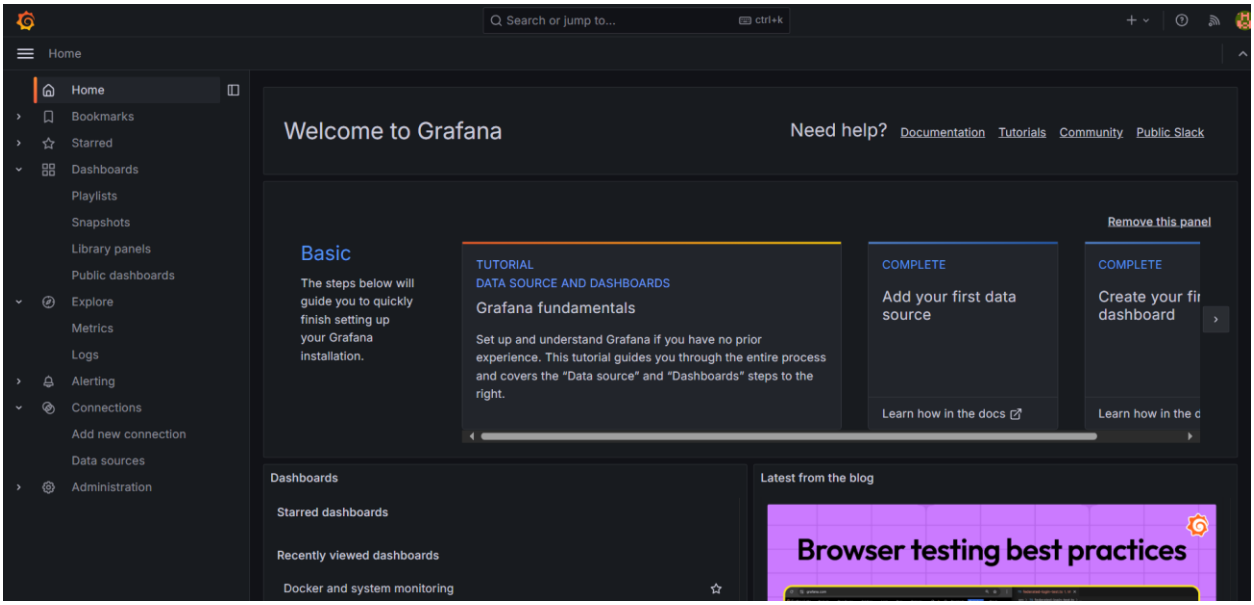


Figure 3.7: Grafana interface

- Open the New dropdown menu, choose New Dashboard/Import Dashboard

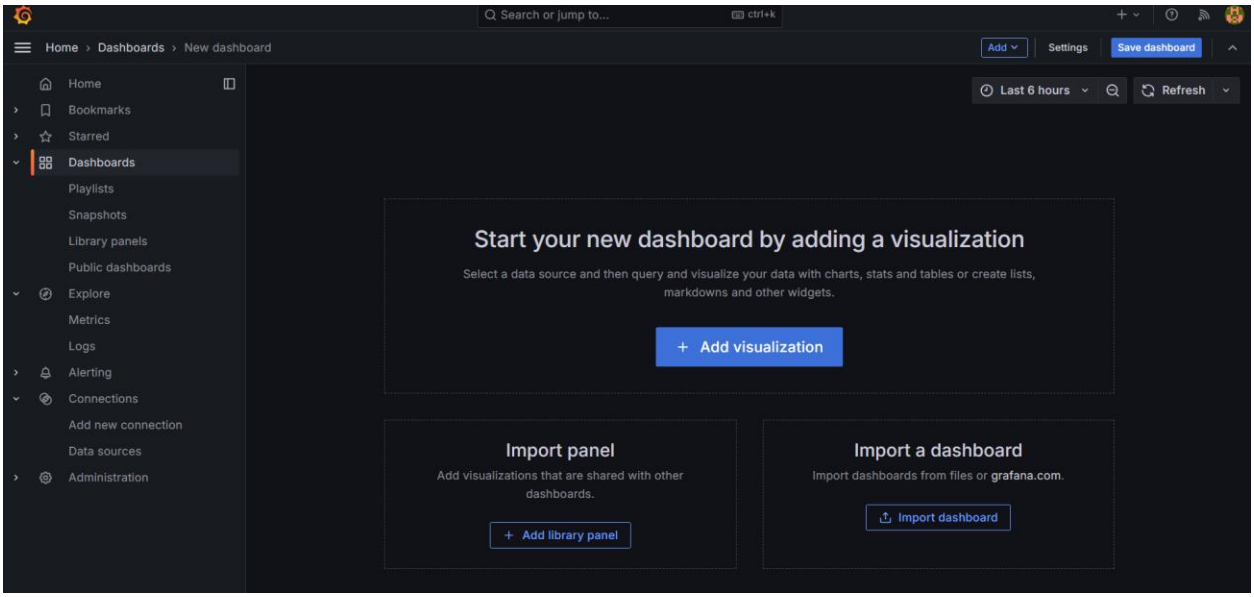


Figure 3.8: Grafana dashboard import

- Next, choose the underlined link to find the suitable dashboard for the metrics. In the search panel, search for 'Docker and system monitoring". Click on "Copy ID to clipboard" to use later.

Figure 3.9: Monitoring template

- Back to the import dashboard page, paste the ID to the text box and click Load.

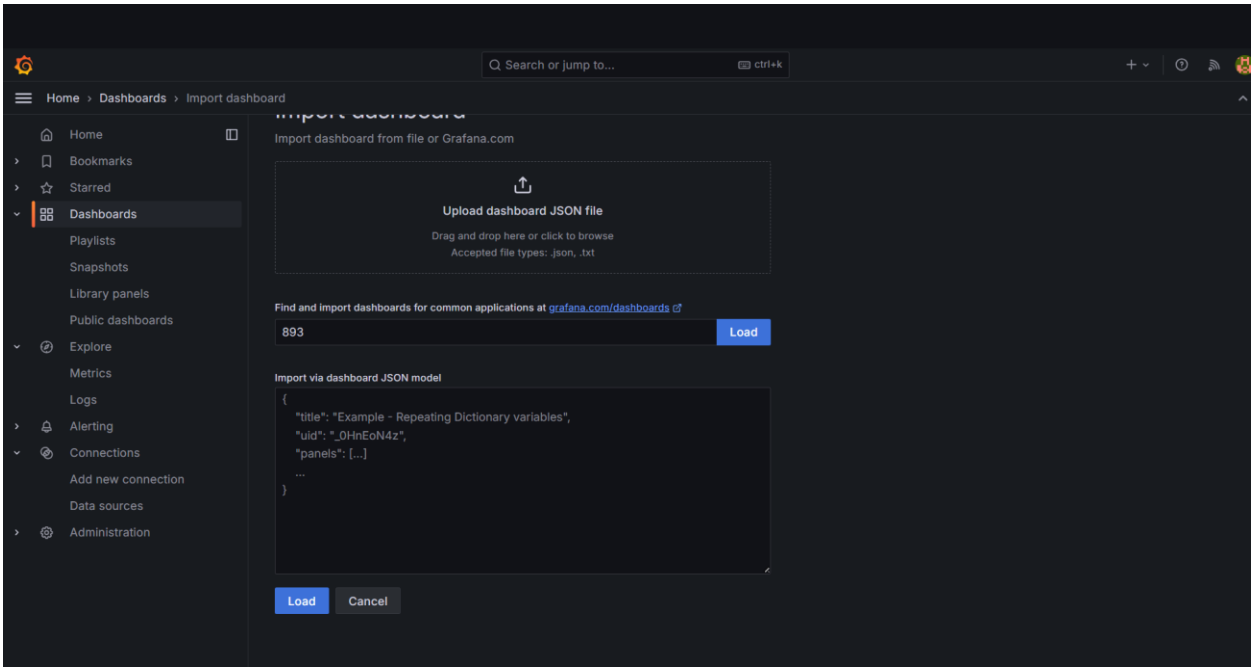


Figure 3.10: Dashboard configuration step

- Click on Prometheus to choose the default data source.

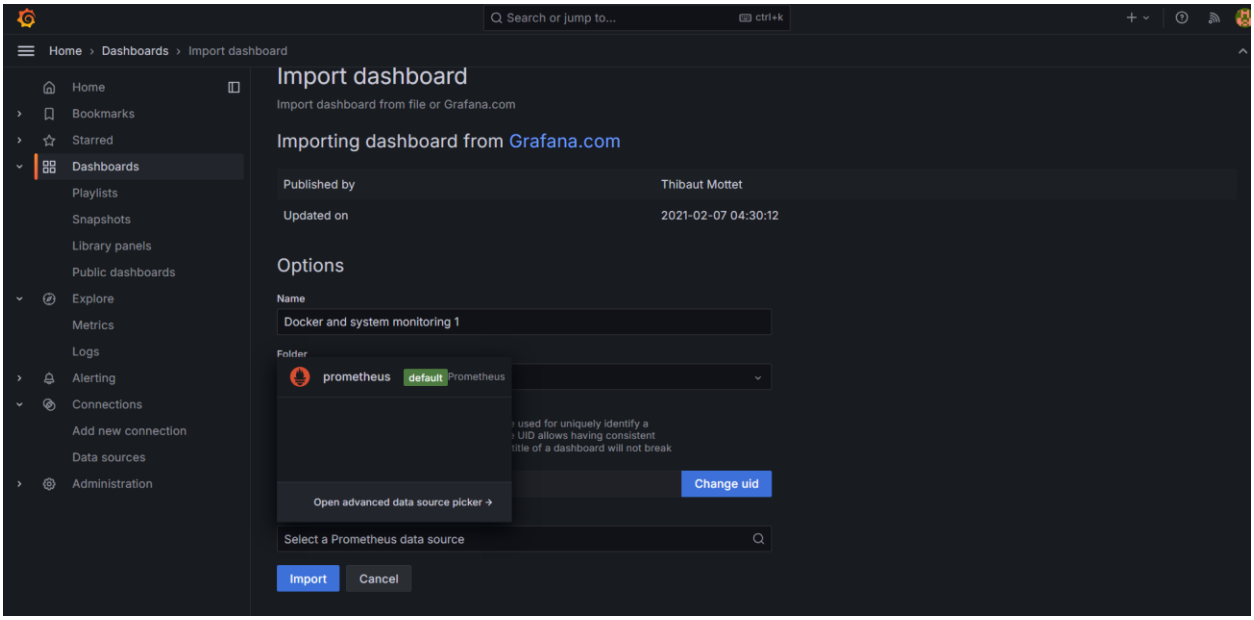


Figure 3.11: Dashboard configuration step

- After clicking Import, the dashboard will appear. That's it for performance monitoring for the production server.

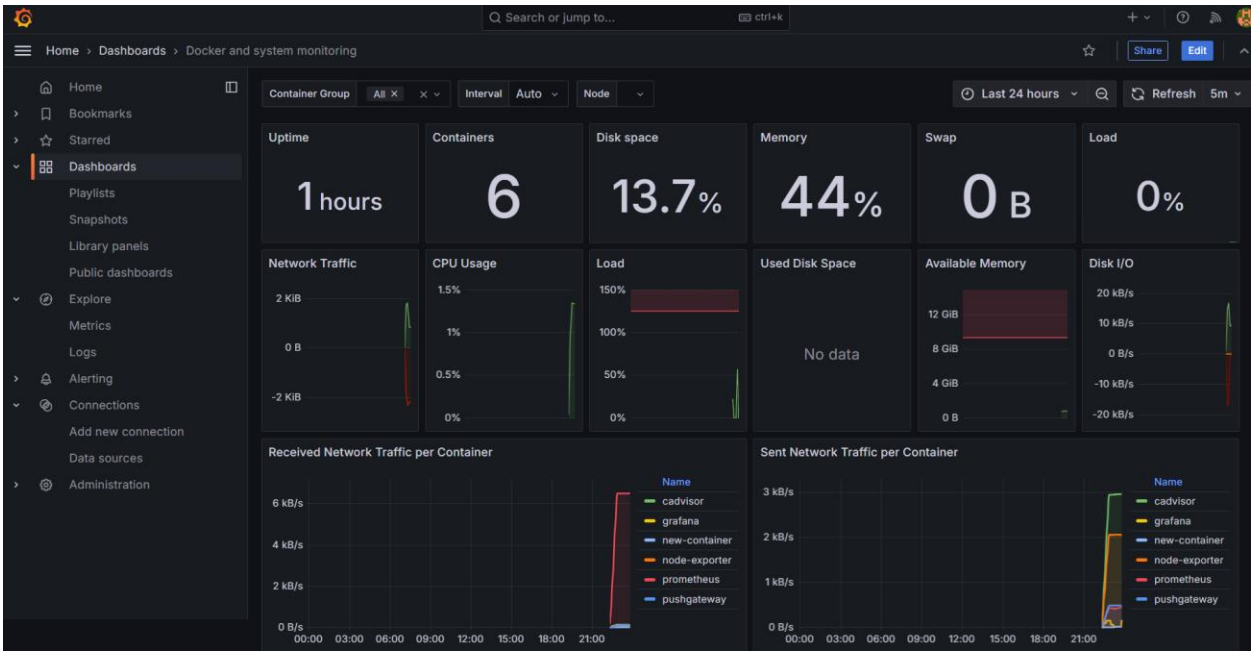


Figure 3.12: Dashboard

3.5. Pipeline setup:

- Now go back to Jenkins server at http://20.2.218.210:8080/ to create a pipeline. First click on New Item.

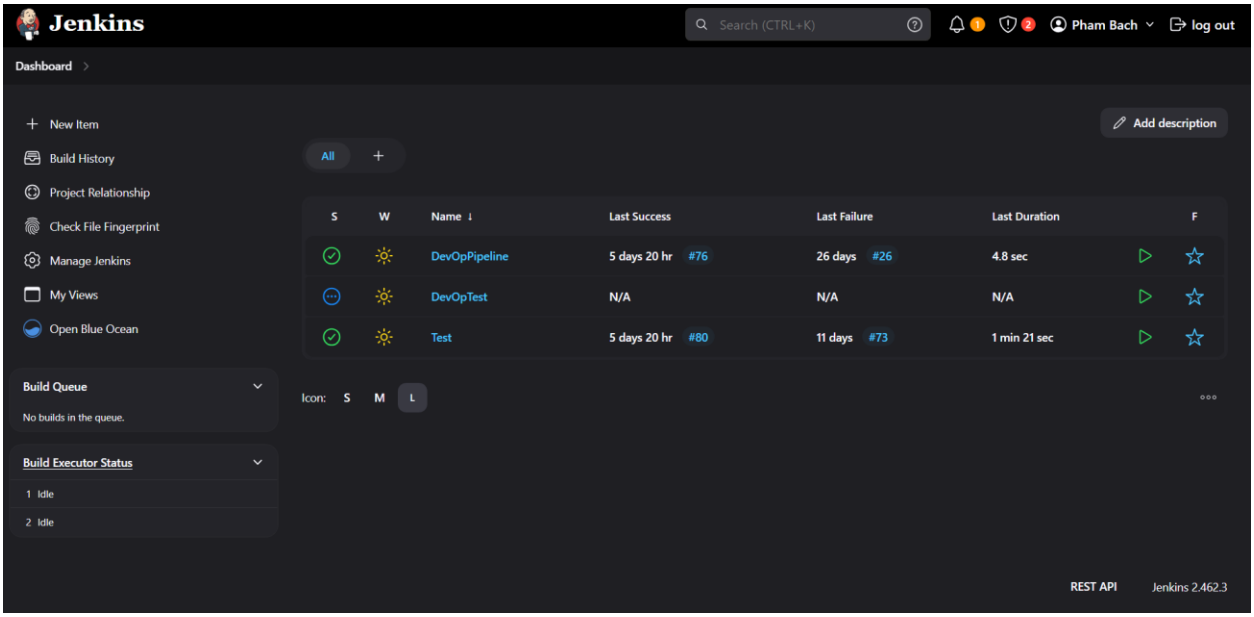


Figure 4.1: Jenkins interface

- Set the name of the pipeline to Test, choose the item type as pipeline. Then click on OK.

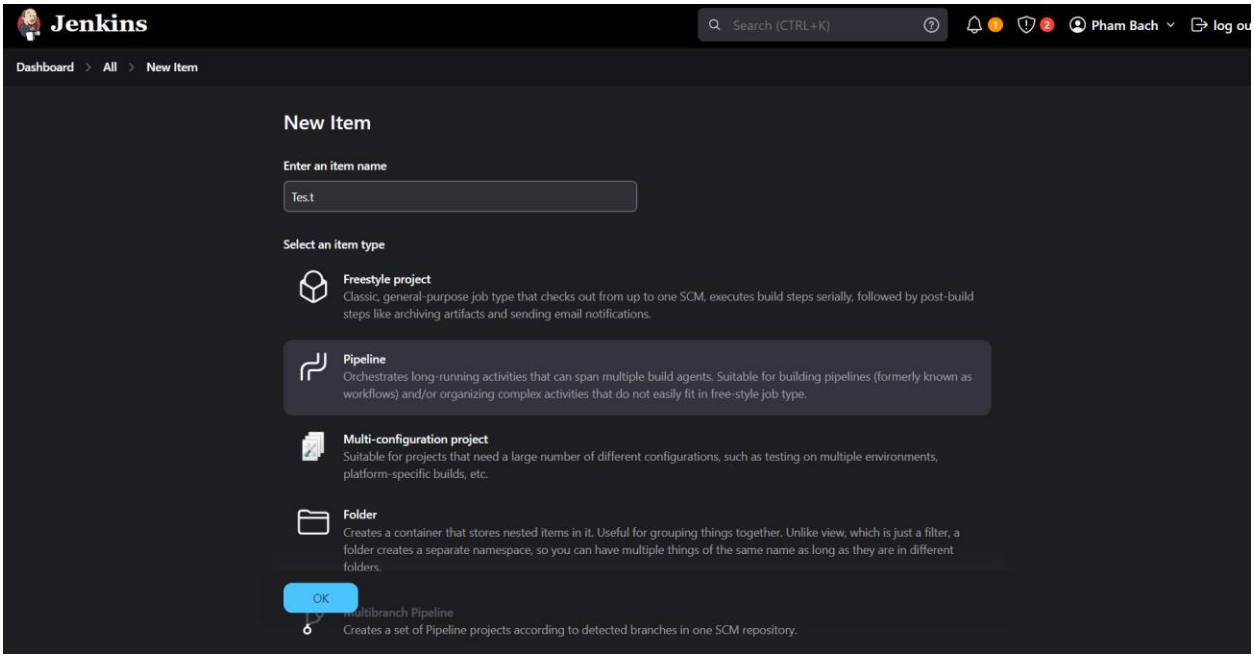


Figure 4.2: Pipeline configuration

- Inside the project, first tick on Github Project and provide the link to the Github repository.

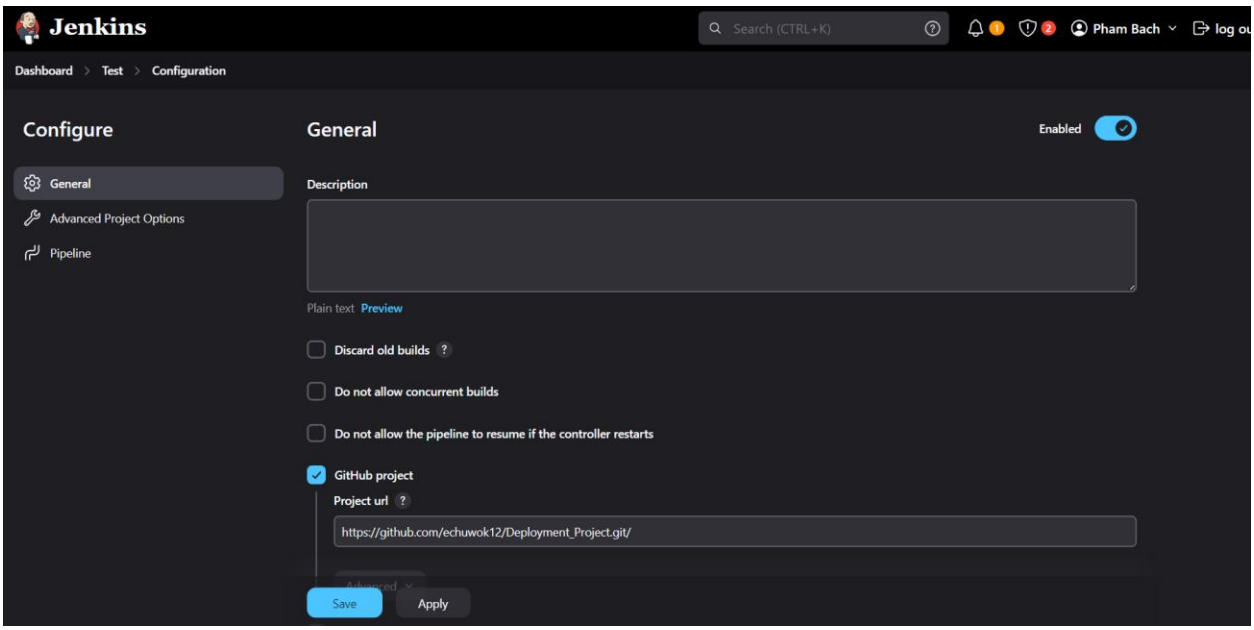


Figure 4.3: Pipeline configuration

- Next, in the Build Trigger section, tick on “Github Hook Trigger for GITScm polling”. This allows the pipeline to pull the code form Github whenever a change is committed on Github.

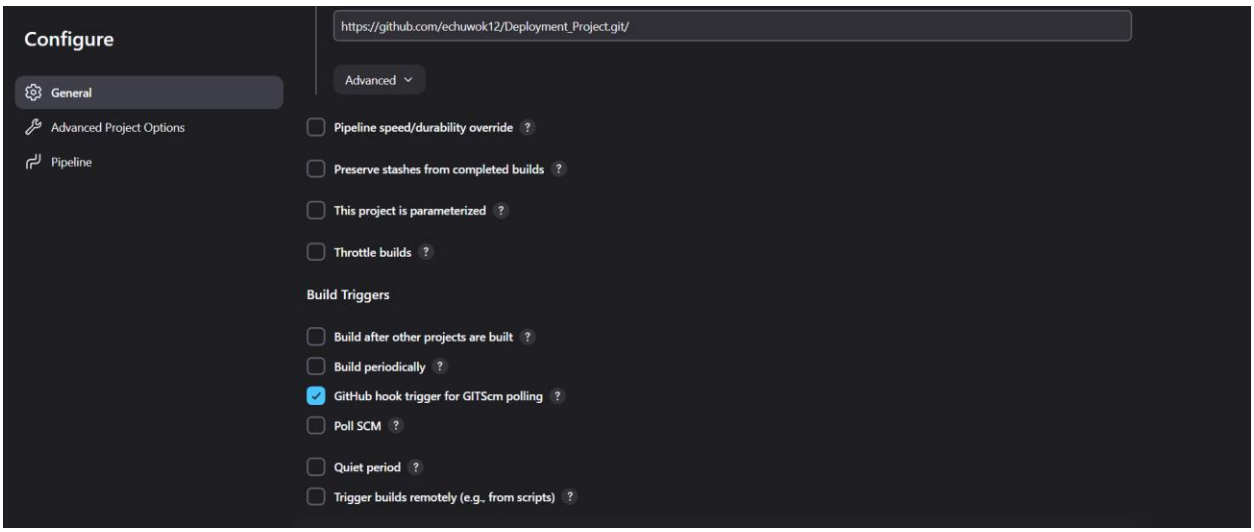


Figure 4.4: Pipeline configuration

- For the Pipeline section, do the same as in the provided image.

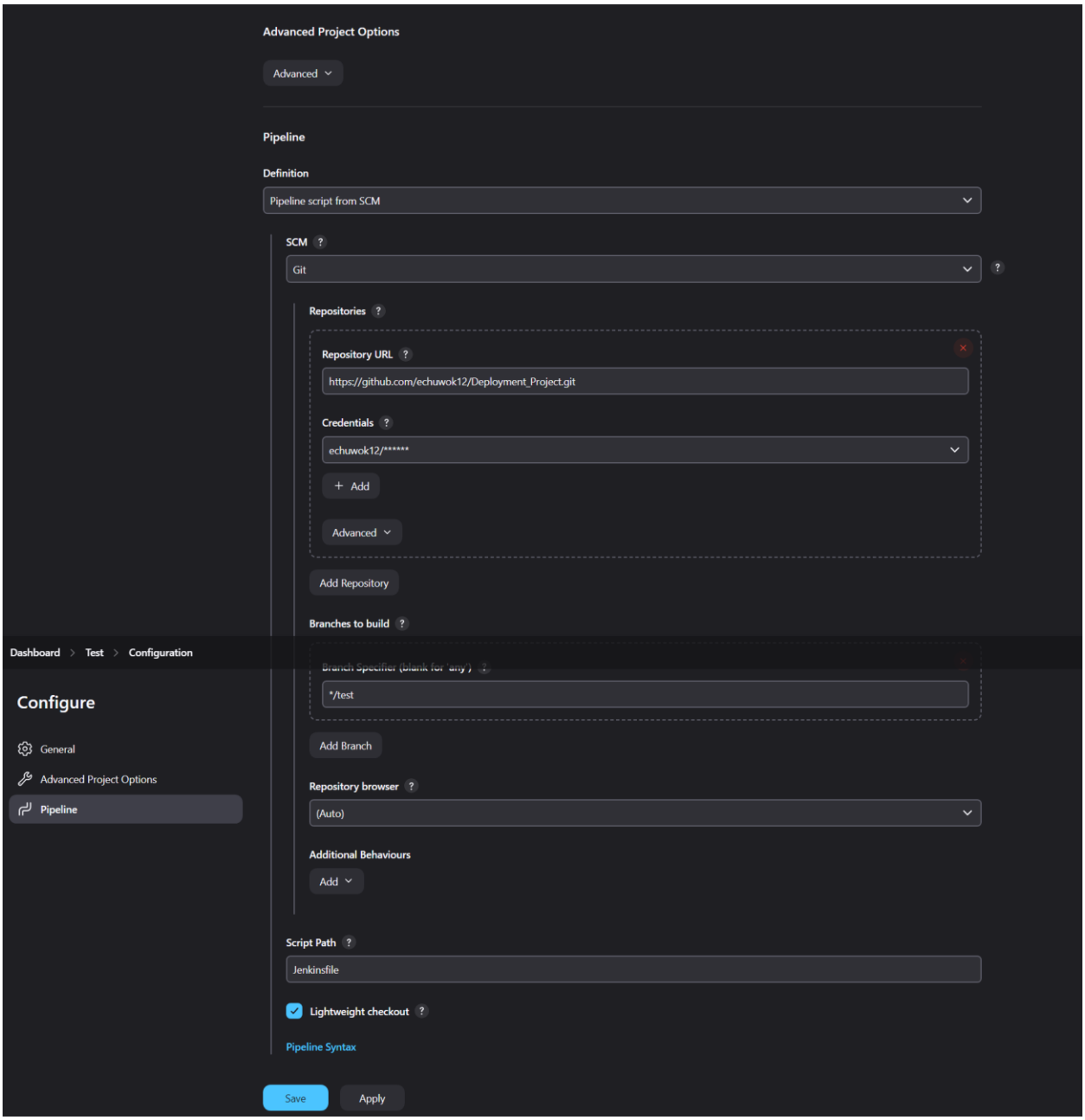


Figure 4.5: Pipeline configuration

- Finally, click on Save and the pipeline is ready to use. To use this pipeline, make any changes to the code on Github, this will trigger the pipeline to run. A video is provided to demonstrate the operation of the pipeline at each level in the assignment submission.

4. Project Outcomes

4.1 Automation and Integration

- The pipeline was triggered automatically by changes pushed to the GitHub repository through configured webhooks.
- Jenkins orchestrated all pipeline stages, including code retrieval, testing, and deployment, without manual intervention.

4.2 Quality Code Enforcement

- SonarQube was used to analyze the code for bugs, vulnerabilities, and code smells.
- Quality gates were configured to ensure only compliant code proceeded to the deployment stage.
- Developers received detailed feedback on flagged issues, facilitating timely corrections.

4.3 Streamlined Deployment

- The ProductionServer hosted the application in a Dockerized environment for consistent and efficient deployment.
- Nginx was configured as a reverse proxy with SSL encryption, ensuring secure communication and optimal performance.
- The deployment process was fully automated, reducing setup time and effort.

4.4 Monitoring and Metrics Visualization

- Prometheus collected real-time metrics such as CPU usage, memory consumption, response times, and error rates.
- Grafana dashboards displayed key metrics, offering an intuitive overview of system health and performance.
- Alerts were configured for early detection of potential issues, enabling proactive resolutions.

4.5 Lessons Learned

The project emphasized the importance of clear planning and a modular approach to tool integration. By adopting this method, we ensured smooth communication and functionality between Jenkins, SonarQube, Docker, Prometheus, and Grafana, allowing each tool to perform its specific function without conflict. Configuring GitHub webhooks, SSL for Nginx, and Prometheus exporters presented initial challenges, but these were addressed through thorough testing, detailed documentation, and research, which helped fine-tune the system. Additionally, troubleshooting issues such as network configurations and resource constraints during deployment further reinforced the need for thorough validation and stress testing. The project underscored the value of automation in reducing manual errors

and the critical role of monitoring in maintaining system reliability and performance, providing real-time visibility into system health, which is crucial for proactive issue resolution and ensuring minimal downtime in production environments.

4.6 Final Results

The completed pipeline successfully automated the build, test, and deployment processes for sample applications, maintaining strict quality standards. Monitoring dashboards provide real-time insights into application performance and system health under various load conditions. The pipeline's design proved scalable, accommodating potential future enhancements like expanding CI/CD support for additional projects or adding quality gates. Overall, the project demonstrated how a well-implemented DevOps pipeline can enhance productivity, ensure code reliability, and optimize system performance.