**Task 3: Continuous Integration and Continuous Delivery (CI/CD) Pipeline**

**Introduction:**

A pipeline refers to a set of processes which are automated with the help of tools that enable continuous integration, delivery and deployment of applications. It is an essential component of DevOps methodology. The stages in software lifecycle are automated allowing faster and reliable delivery of a software system. The main significance of CI/CD pipeline is that it enforce consistency in development and deployment processes which are highly adaptable and scalable.

**CI/CD pipeline explanation:**

As mentioned below, Git and GitHub were used as the source code management(SCM) tool for CI(continuous integration). Jenkins was used as an automation server which played crucial role in CI/CD pipeline. Jenkins consisted of a JenkinsFile which included set of actions to be performed in the pipeline. Whenever there was a change in code, Jenkins detected it using Webhooks in GitHub and started triggering the jenkins pipeline. Webhooks were a way to receive automated notifications and updates that occured within a GitHub repository. Next step in the pipeline was building the application which was carried out using Maven. Maven’s role in the pipeline was to manage the build process and generate an executable Jar file. Going further, Sonarqube was used as a static code analysis tool that was integrated in CI/CD pipeline. The scanner in Sonarqube was used to for the testing purpose and analyse various code quality issues, security vulnerabilities and maintainability problems, and generate a report which can be viewed on the interface. Next step was using Docker to generate docker images which includes executable packages consisting of code, system tools, libraries, etc. These docker image was then pushed to DockerHub. The final step in CI/CD pipeline was to use Kubernetes for the deployment of the application.



Git/GitHub Jenkins Maven Sonarqube Docker Kubernetes

**The aims of this assessment was:**

To create a pipeline which includes source code repository, automated build and testing and deployment to a staging environment

**Method:**

1. **Git/GitHub (Source code repository – Source Code Management):**

**What is Git/GitHub and how it is used in pipeline:**

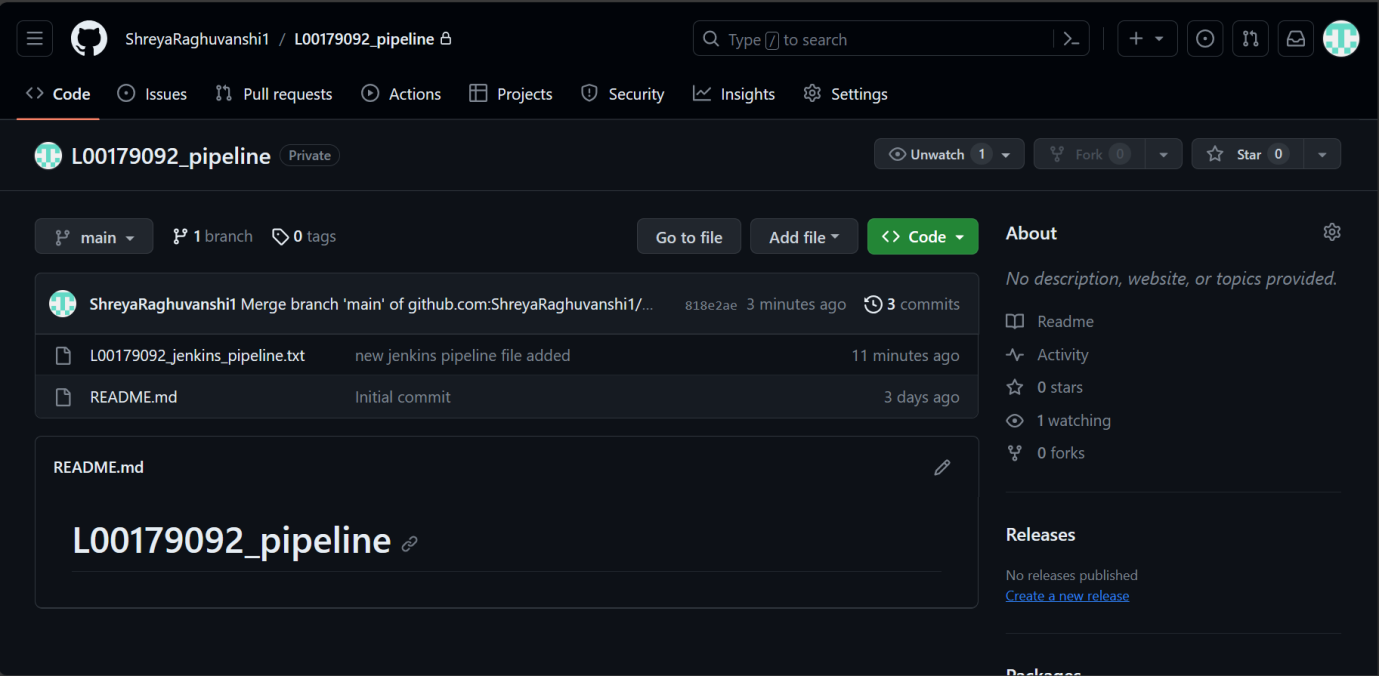
GitHub is a hosting service for Git repositories that enables sharing, reviewing, and collaborating on code. Git/ GitHub was used as a source code management tool. In a CI/CD pipeline, Git/GitHub integration allows for automated processes, such as triggering builds upon code changes (CI) ensuring rapid, consistent, and reliable software delivery.

**Reasons for choosing Git/GitHub:**

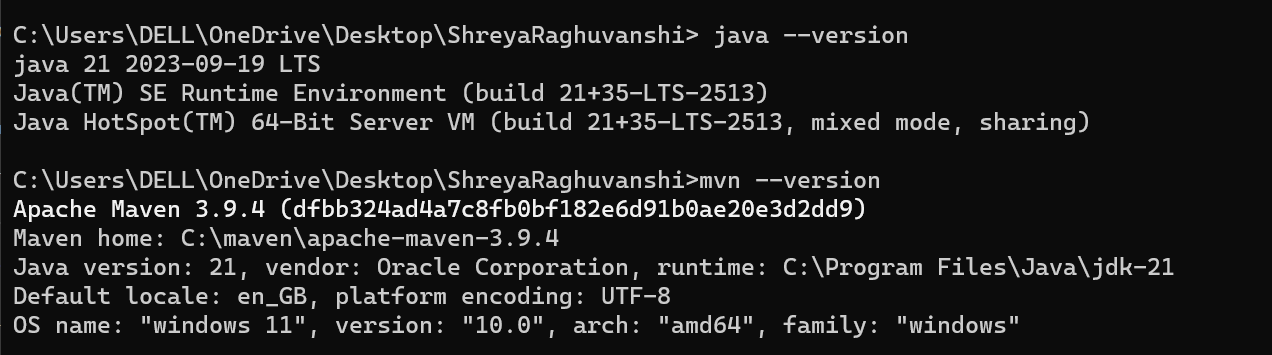
The reason for choosing Git/GitHub as Source code repository was it serves as a reliable version control system which allows teams to collaborate and track changes made to the code. CI/CD pipelines was integrated with GitHub repository which enabled automatic triggers for build and deployment processes whenever changes are pushed to the repository. Webhooks in GitHub was another reason why GitHub was chosen. Whenever there is a push in GitHub repository was it triggers a message to Jenkins and Jenkins then starts building the pipeline. GitHub integrated seamlessly with Jenkins enabling automated build, test, and deployment pipelines.

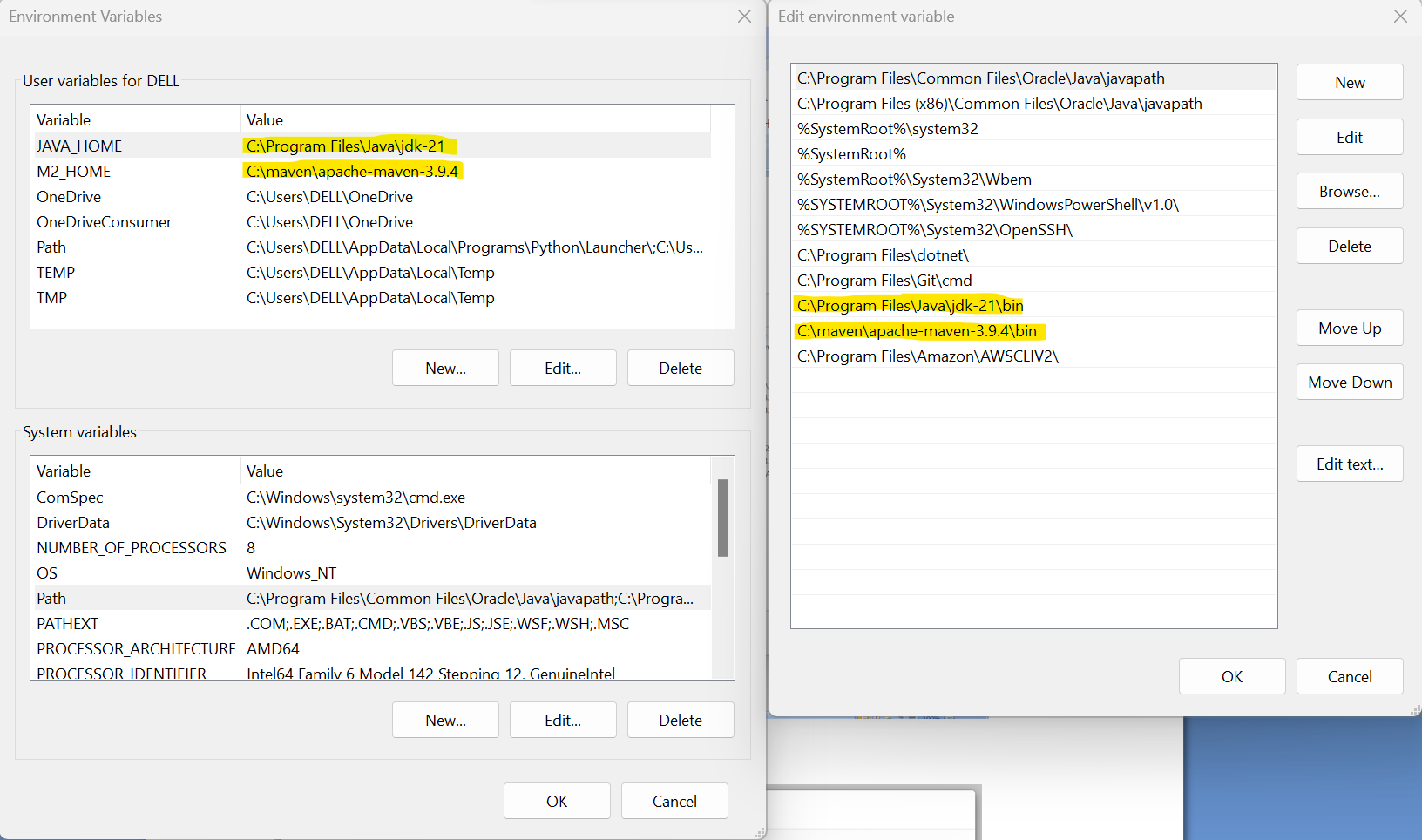
**Steps for Git/GitHub configuration with Jenkins:**

1. A new private repository ‘[L00179092\_pipeline](https://github.com/ShreyaRaghuvanshi1/L00179092_pipeline)’ was created and a file ‘L00179092\_jenkins\_pipeline’ was pushed into the GitHub. Initially this file was empty.



1. Java JDK 21 and Maven was installed. Path for both was added in the environment variables of the local system.





**Jenkins (Automation server, Continuous integration tool):**

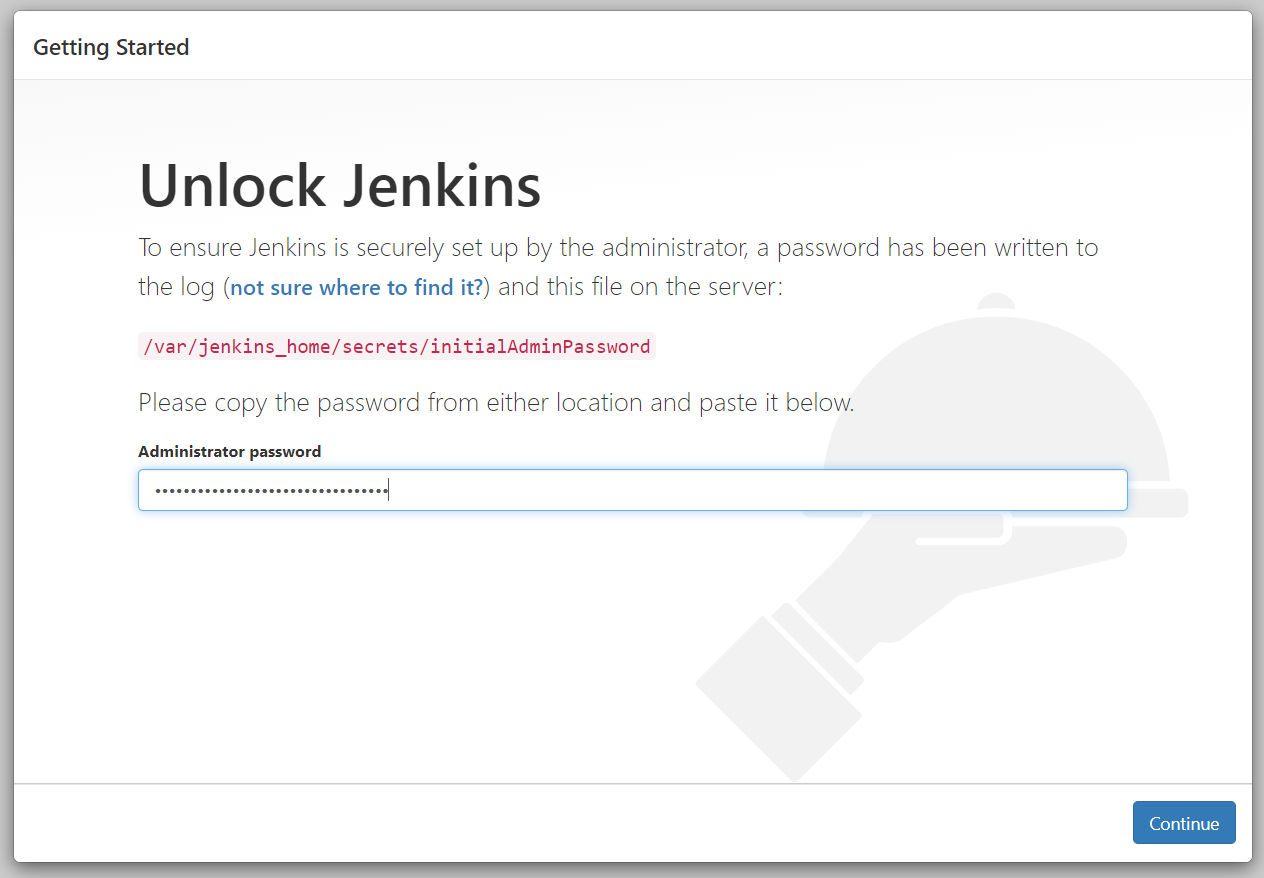
**What is Jenkins and how it is used in pipeline:**

Jenkins is an open source automation server commonly used to build, deploy, and automate software development processes. It was used primarily for continuous integration (CI) and continuous delivery/continuous deployment (CD) pipelines. Jenkins was used in pipelines by defining stages and steps within a pipeline script. Jenkins provides a rich set of plugins that extend its functionalities, allowing users to integrate with a wide range of tools, version control systems like git, build tools like maven, testing frameworks and cloud services.

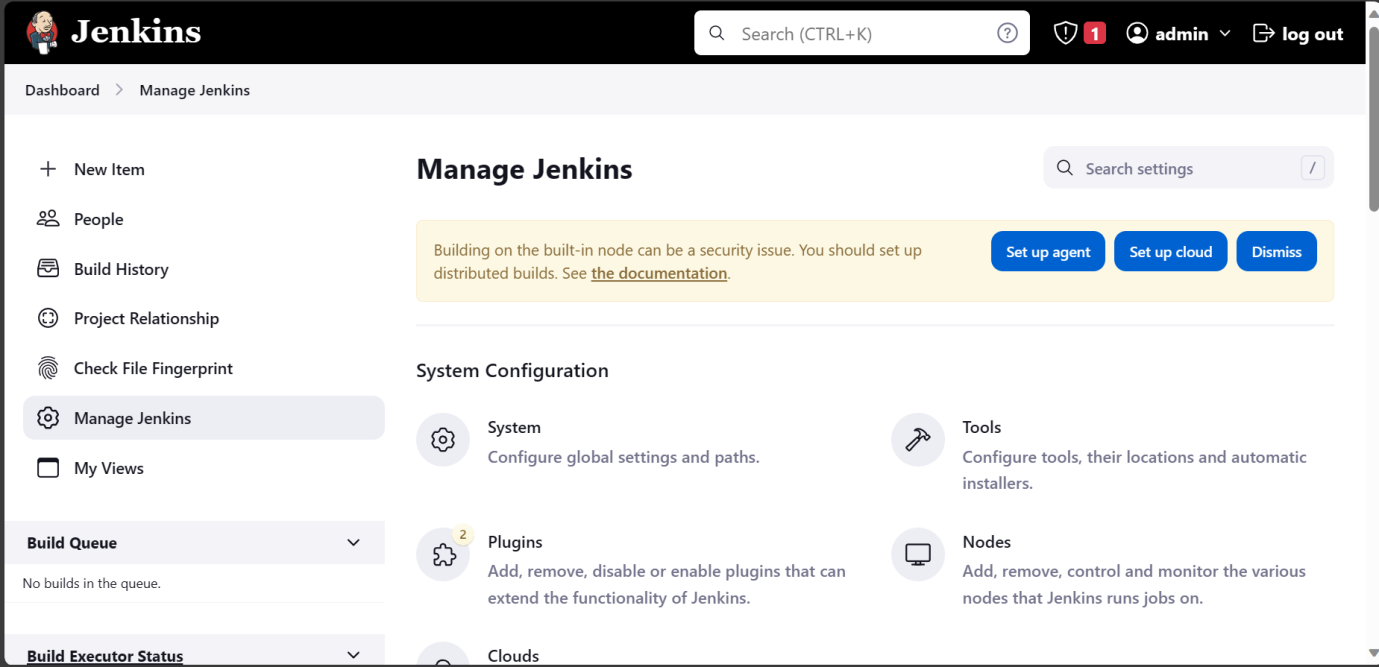
**Reasons for choosing Jenkins:**

Jenkins provides flexibility in creating pipelines. It supports both declarative and scripted pipeline syntax, enabling users to define pipelines as code. Jenkins was written in java but because of its in-build syntax, users with no knowledge in java can also use jenkins. It was also an open source tool. Jenkins has a vast community support so users can interact with forums in-case of an issue. It was also easy to install and configure.

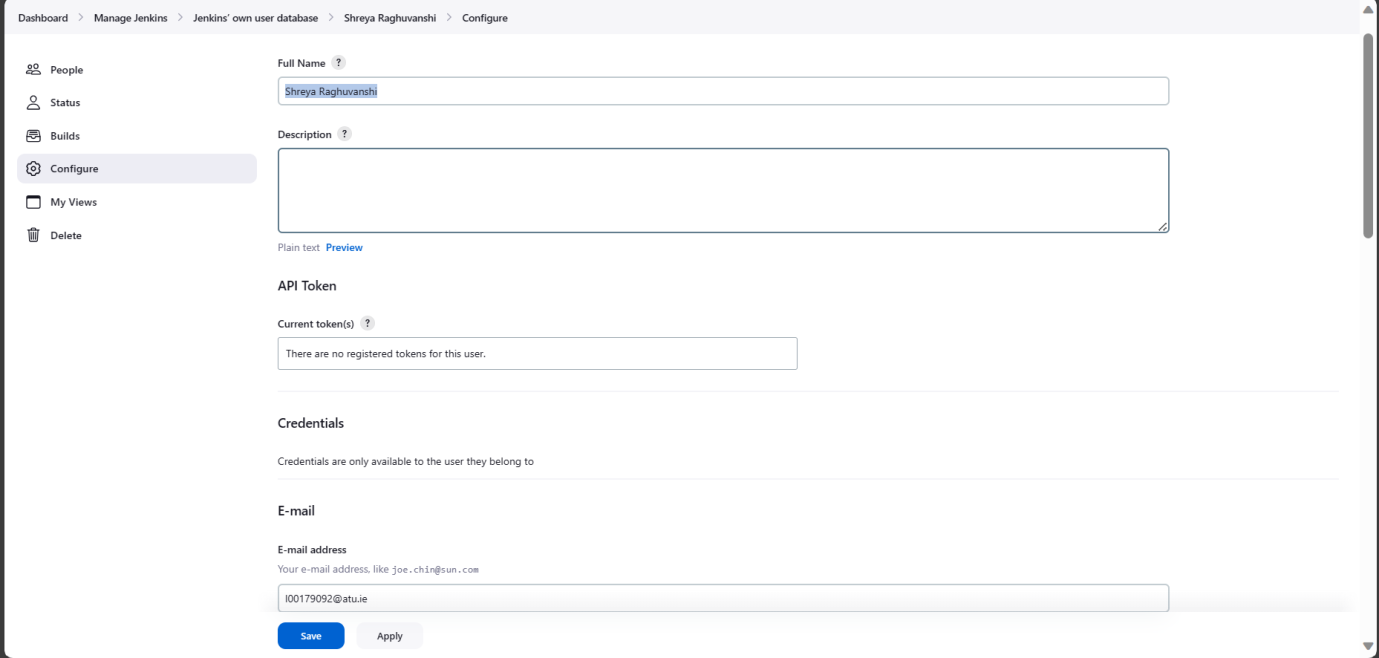
1. After installation of jenkins, it was started at <http://localhost:8080> which was the default URL for Jenkins. The Administrator password was found at /var/jenkins\_home/secrets/initialAdminPassword location



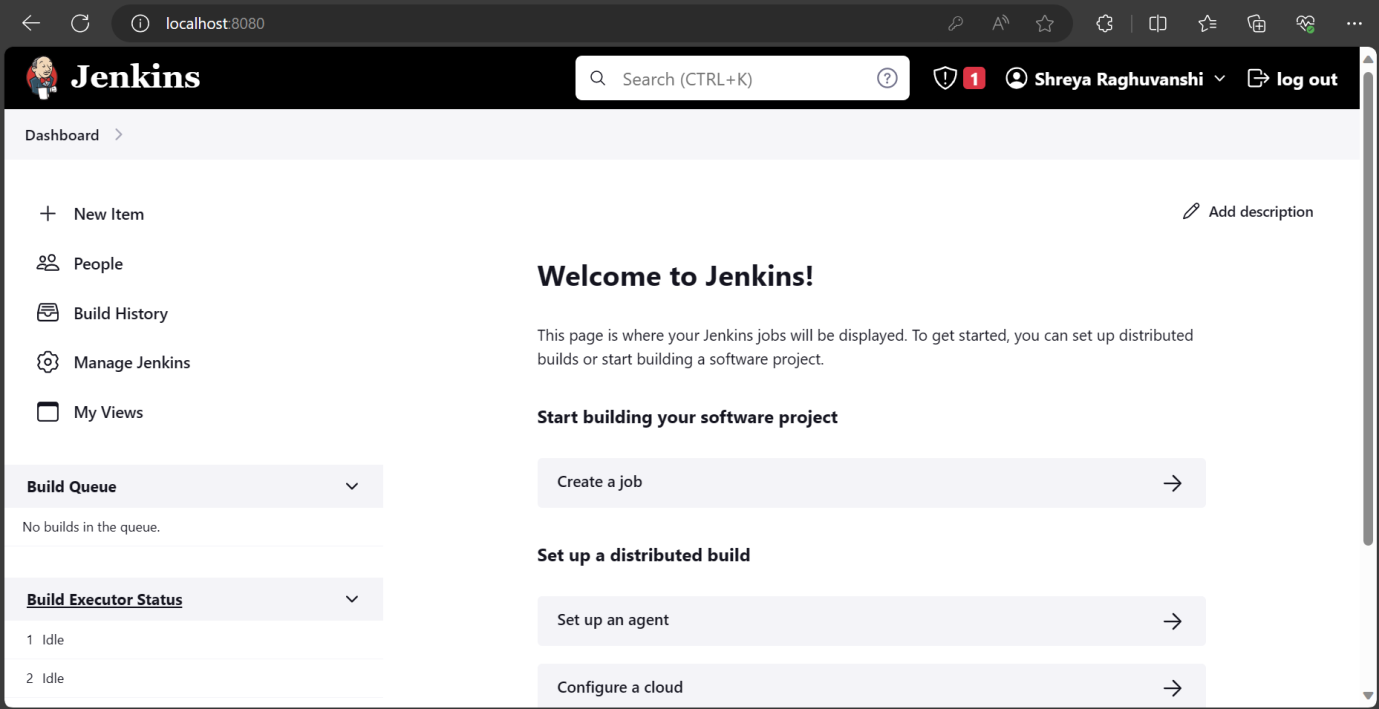
1. After that, jenkins prompted to install suggested plungins. Once all plugins were downloaded, jenkins dashboard was opened as admin.



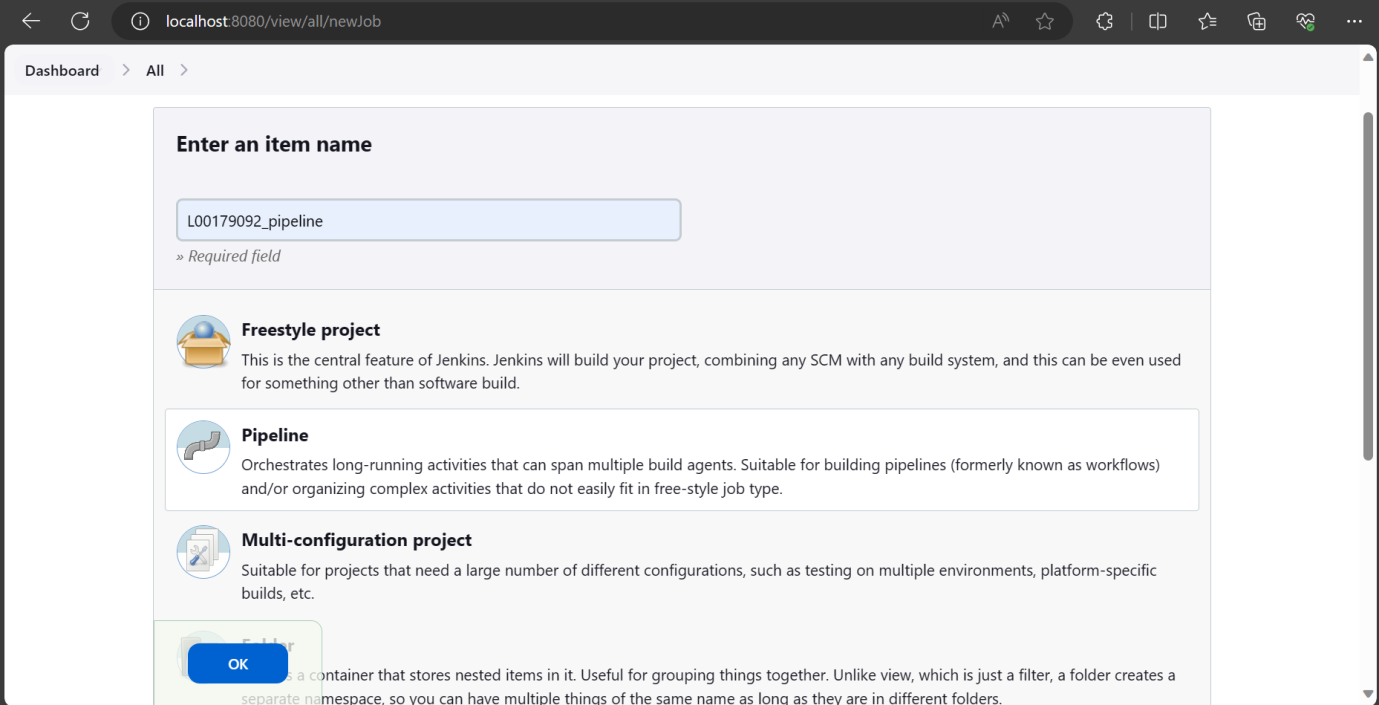
1. A new user ‘Shreya Raghuvanshi’ was created by clicking on manage jenkins -> users -> create user. Then full name, email address and password of choice were entered.



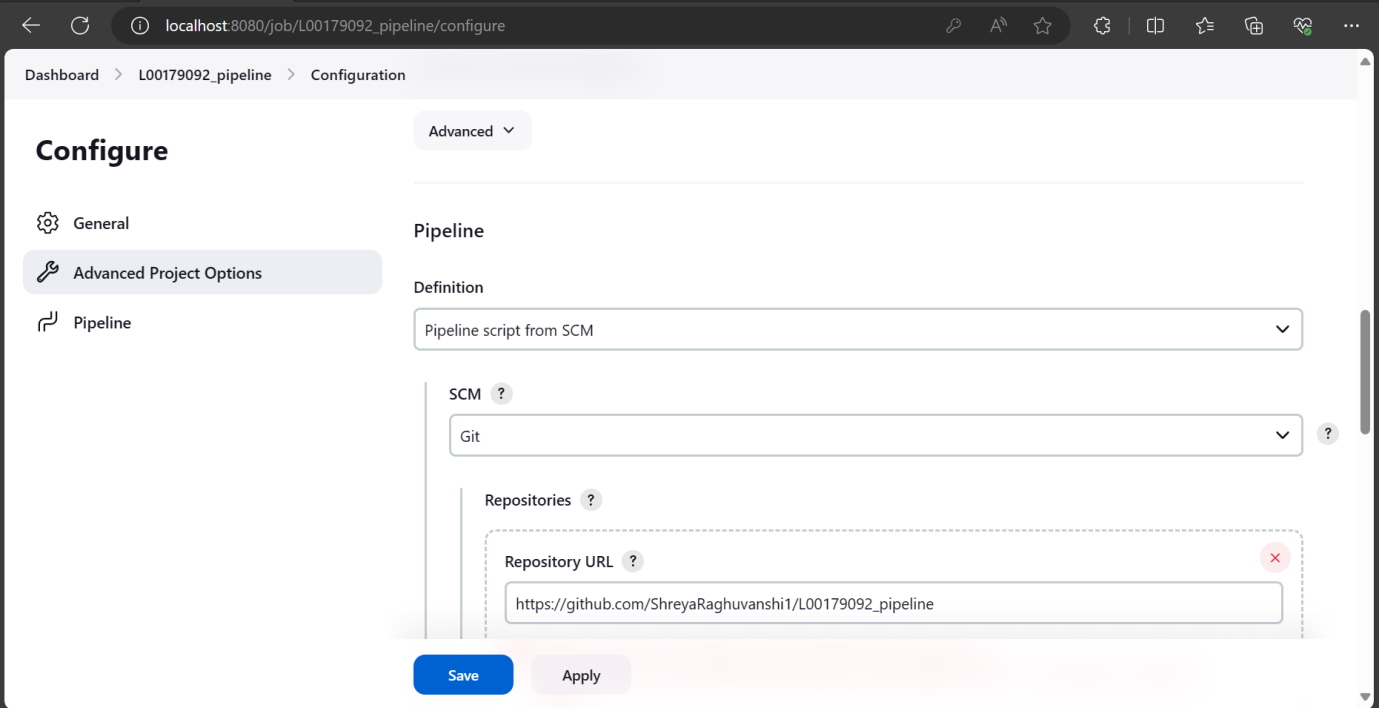
1. After clicking on Apply and then Save, Jenkins was logged in with the user ‘Shreya Raghuvanshi’



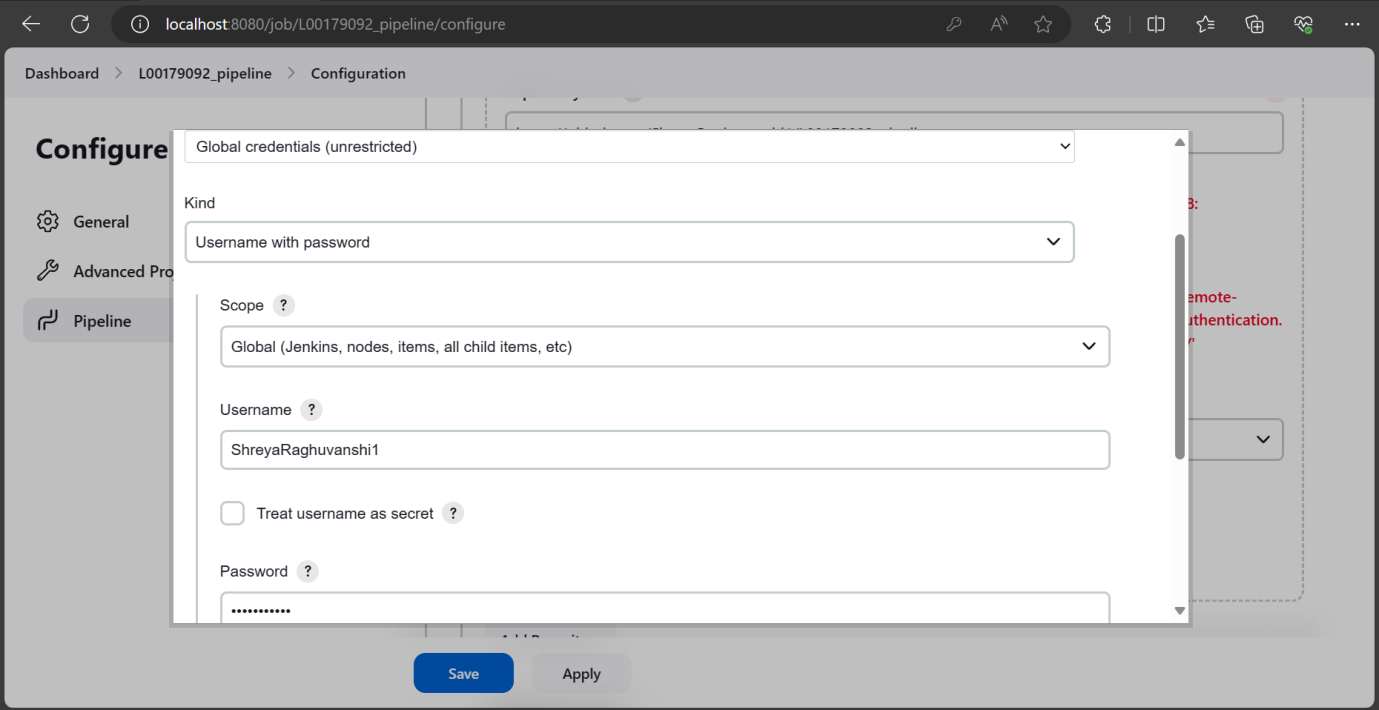
1. A pipeline was created using ‘new item’ button on Jenkins dashboard. ‘L00179092\_pipeline’ name was specified and ‘pipeline’ was selected from the below options:



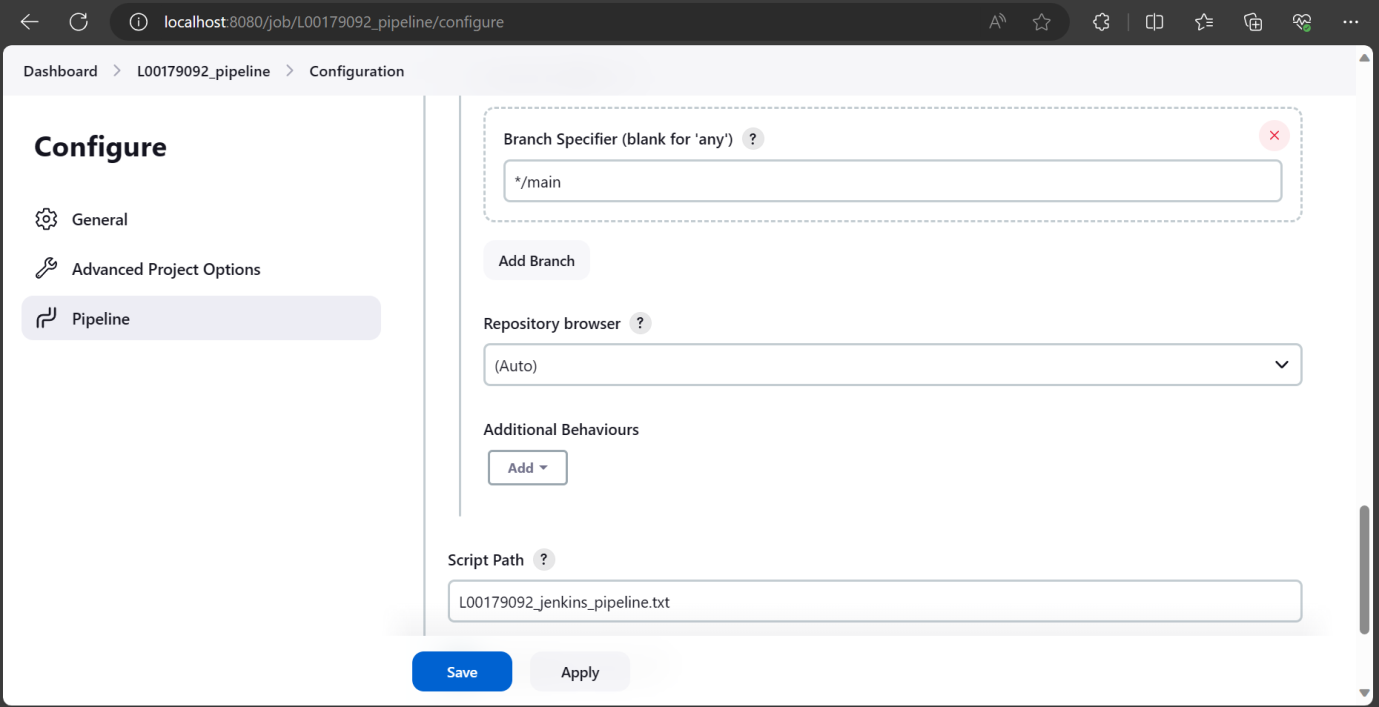
1. Pipeline option was chosen from the left margin. In the Definition, ‘pipeline script from SCM’ was selected as jenkins pipeline code was present in GitHub SCM. Next step was to choose ‘Git’ in the SCM option and insert URL of the GitHub repository in the textbox.



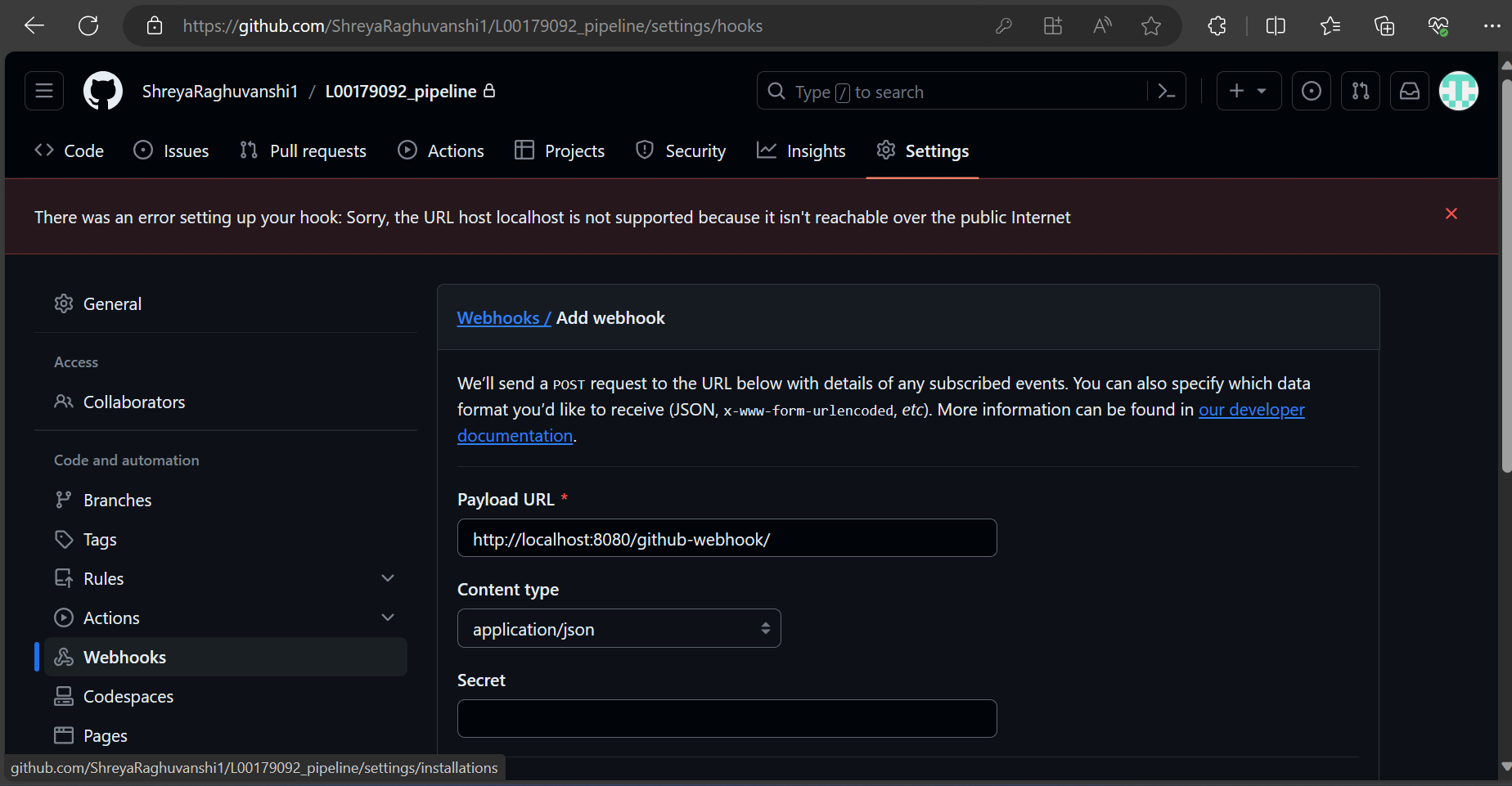
1. As it was a private repository, username and password of our GitHub account was specified so that Jenkins can connect to GitHub using those credentials.



1. Branch was converted from Master(default setting) to Main and the file name of jenkins pipeline which was present in the GitHub was inserted.



1. Next, Webhook was used so that any changes (push) which were made to GitHub repository would reflect in Jenkins. For Webhook, Webhook was enabled on GitHub by choosing the pipeline and clicking on settings. As seen below, an error occured as Jenkins was using localhost:8080 which GitHub does not support.

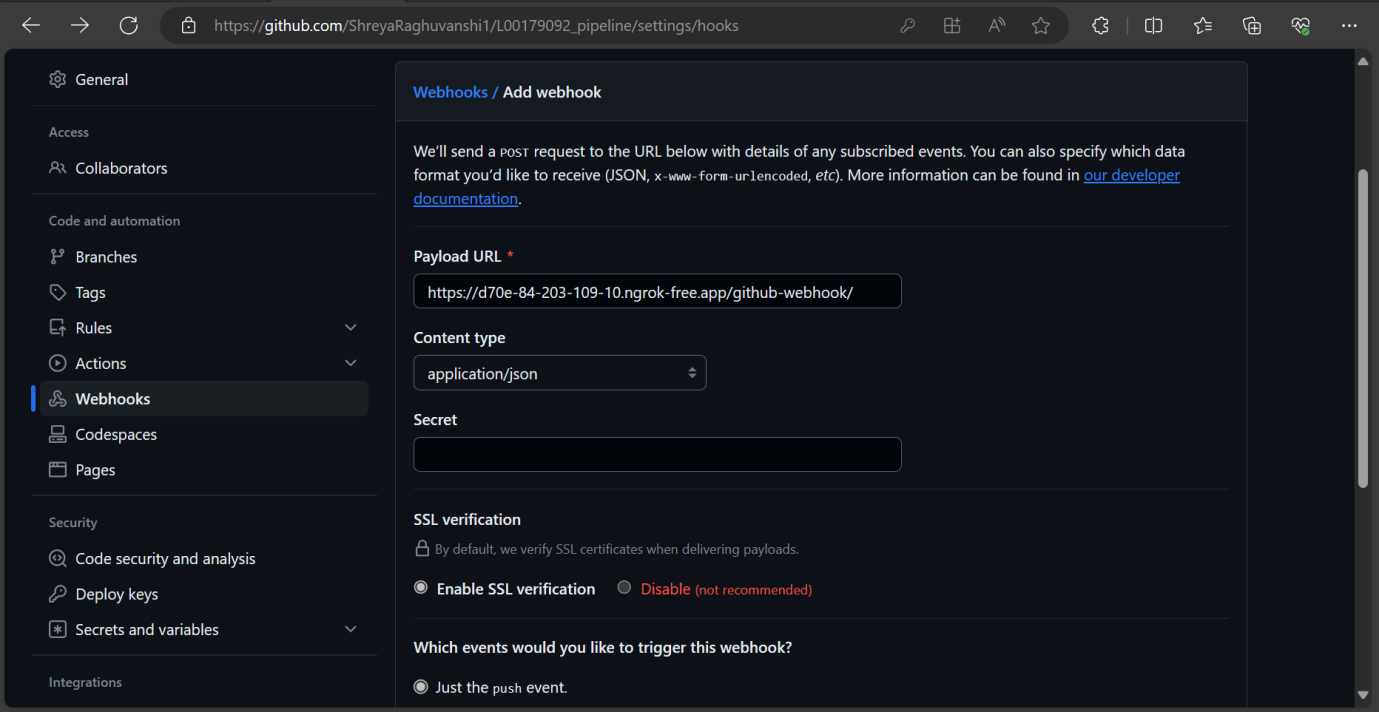


1. To solve the above error, ngrok was used which converts private IP to public IP. A windows version of ngrok was downloaded in the system. After downloading ngrok, agent was authenticated using ngrok config add-authtoken <autotoken>
2. Using the command ngrok http 8080 on CMD the below output was received. The below URL was pasted in the browser.

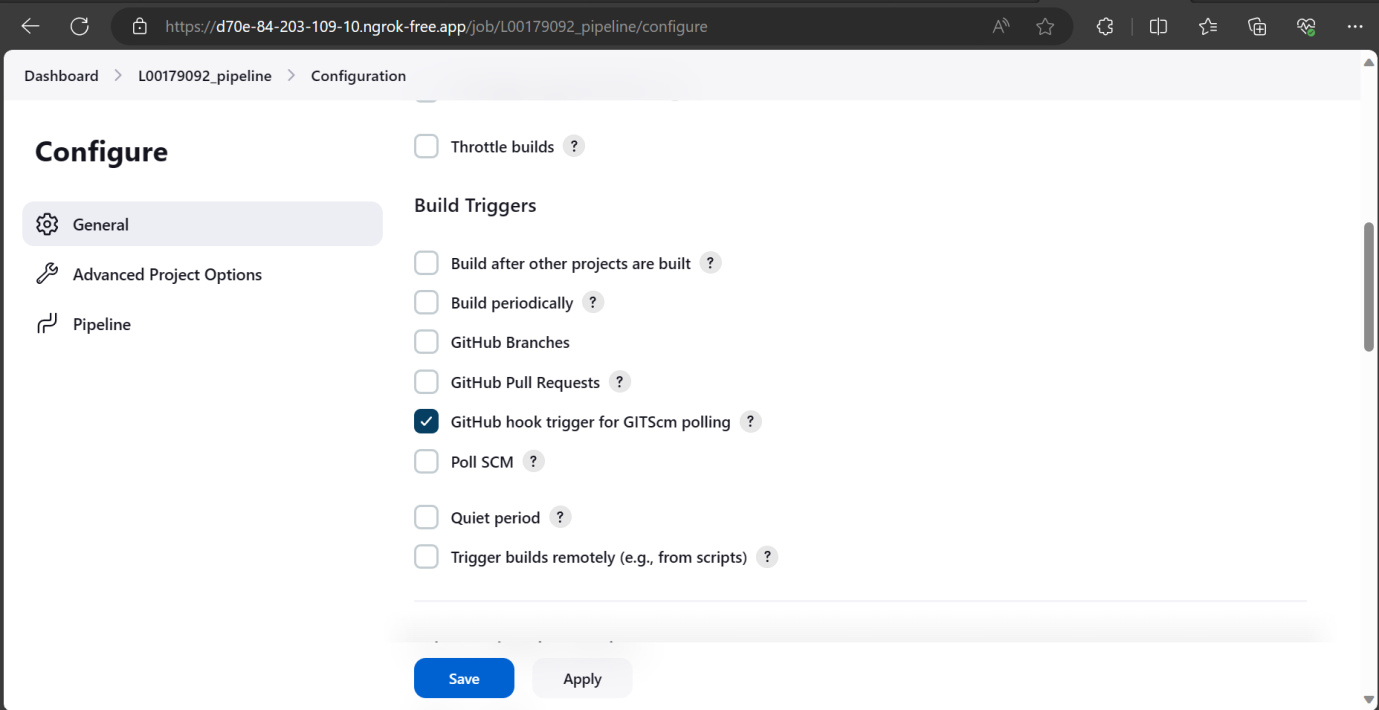




1. Next step was to paste that URL from ngrok to github webhooks for configuring connection between GitHub and Jenkins.



1. Then, build triggers were configured in github by clicking on pipeline -> configure and scrolling down to build triggers. Here, ‘GitHub hook trigger for GITScm polling’ was enabled.



If any changes is pushed in github, Jenkins will automatically get a notifications from github and it will start with the build process.

**Following was the final script which was pasted in the ‘L00179092\_jenkins\_pipeline’ file on GitHub after Git was set up:**

pipeline {

agent any

stages {

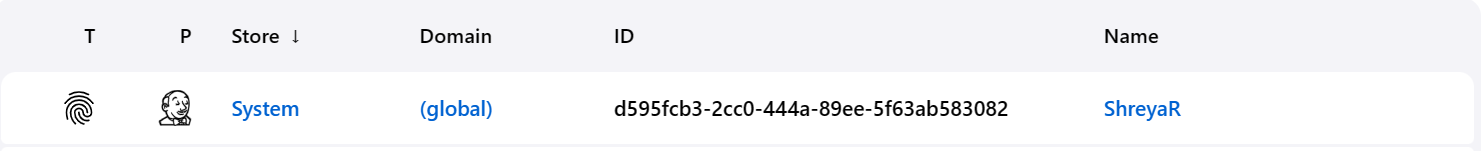
stage('Checkout') {

steps {

git branch: 'main', credentialsId:'d595fcb3-2cc0-444a-89ee-5f63ab583082', url: 'git@github.com:ShreyaRaghuvanshi1/L00179092\_pipeline.git'

} } }

Below is the credentials for Git present on Jenkins



The build was successful.

Output:



1. **SonarQube (Static Code Analysis – Automated testing):**

**What is Sonarqube and how it is used in pipeline:**

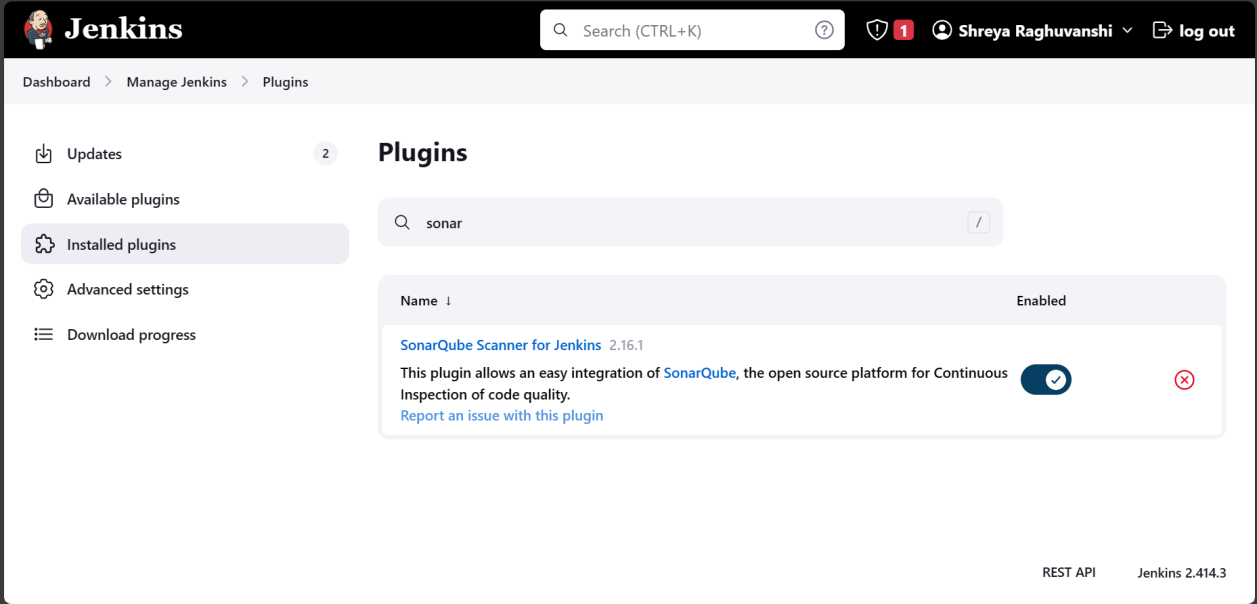
SonarQube is a platform which is free to use and designed for automated reviews and code quality testing. Sonarqube was used in pipeline to manage code quality throughout the software development process and it analyzes code for bugs and vulnerabilities. When integrated into a CI/CD pipeline, SonarQube acts as a quality gate, automatically analyzing code changes, providing feedback, and preventing poor-quality code.

**Reasons for choosing Sonarqube:**

The reason of using Sonarqube was that it supports diverse programming languages. It also had a strong and active community where updates are performed continuously. It was comparatively easy to integrate Sonarqube in CI/CD pipeline and it was also customizable making it an effective choice. Sonarqube is said to be user-friendly making it easy to use for naive users.

**Steps for SonarQube configuration with Jenkins:**

1. Next step was to install Sonarqube plugin on Jenkins.

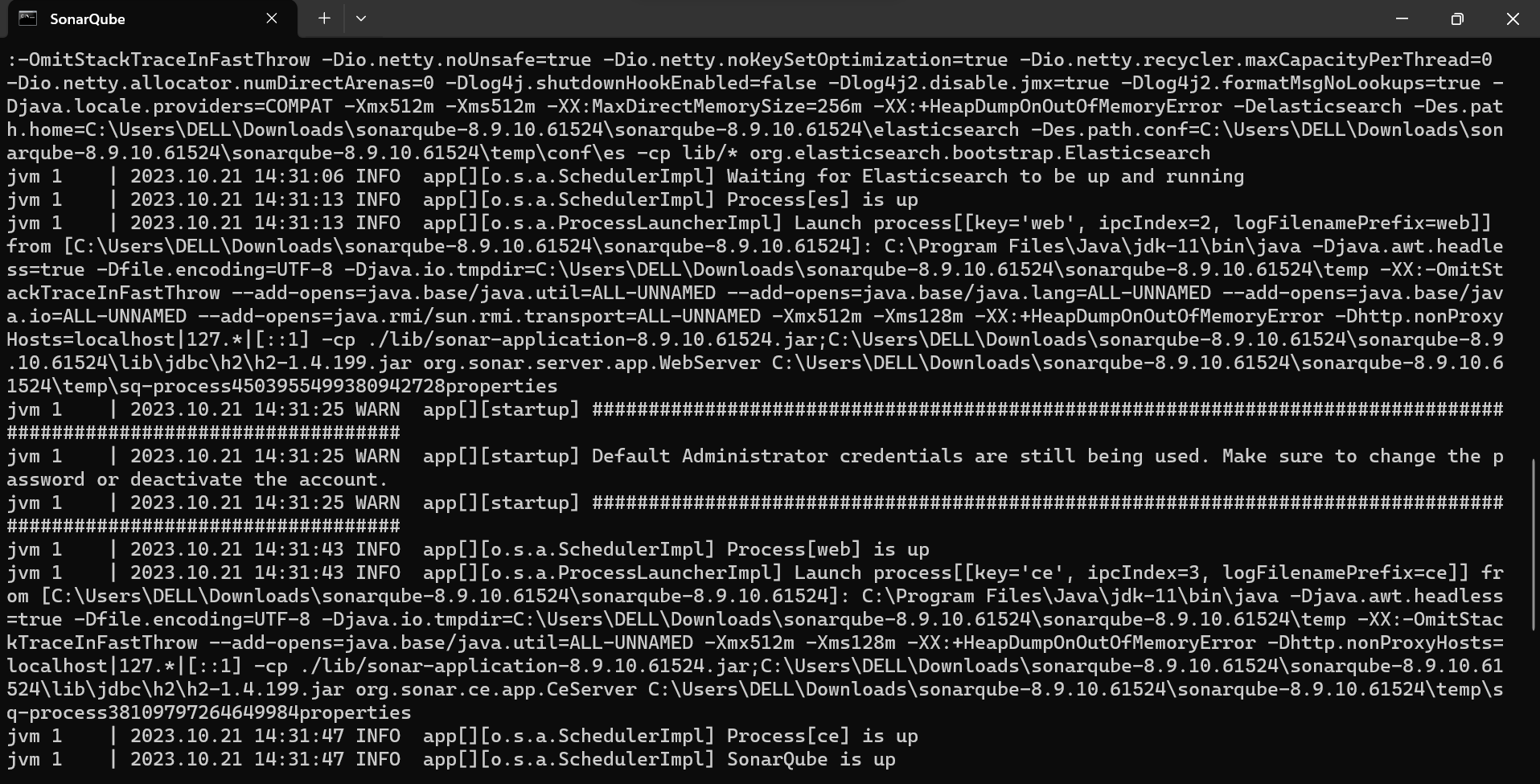


1. Sonarqube-8.9.10.61524 was downloaded and the following changes were made in the config file present in sonarqube folder:

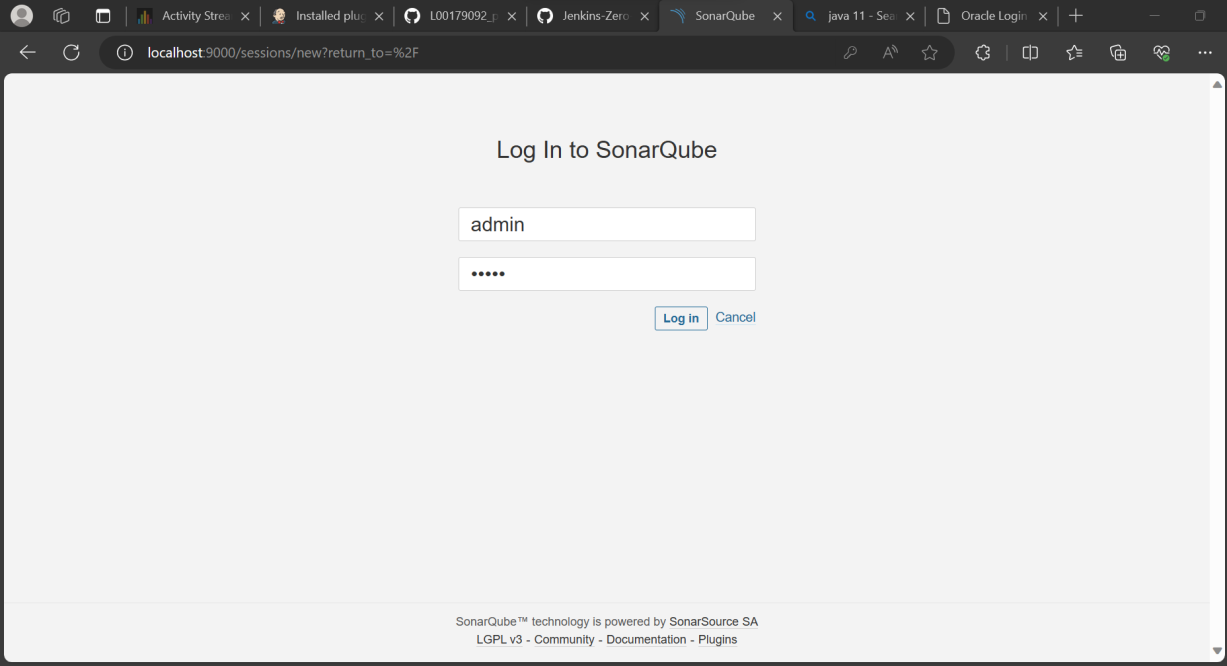
wrapper.java.command=C:/Program Files/Java/jdk-11/bin/java (path of jdk-11 was pasted in the wrapper.conf file. As Sonarqube only supports java-11, jdk 11 was installed in the system)

1. Sonarqube was started using command on CMD:

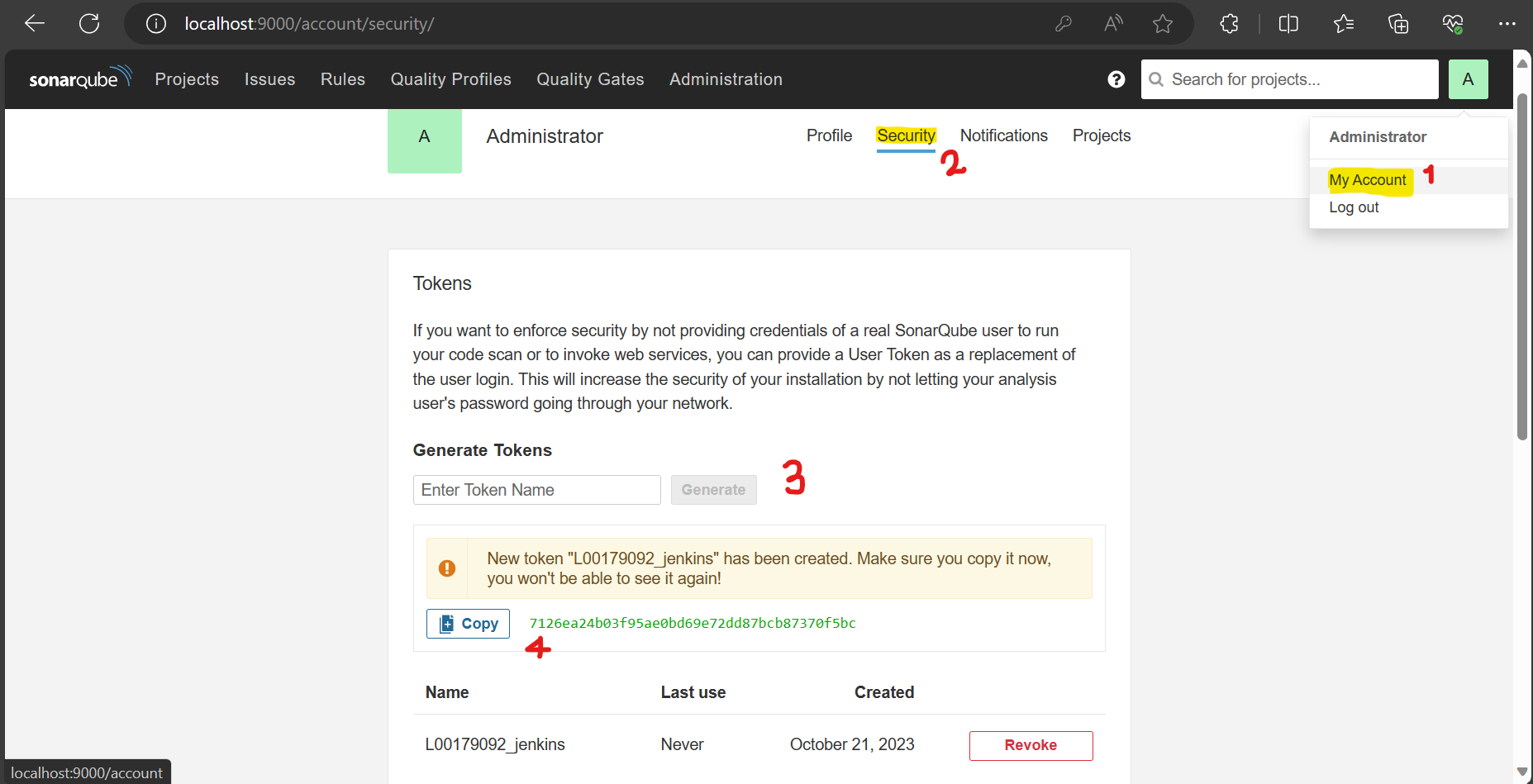
C:\Users\DELL\Downloads\sonarqube-8.9.10.61524\sonarqube-8.9.10.61524\bin\windows-x86-64>StartSonar.bat



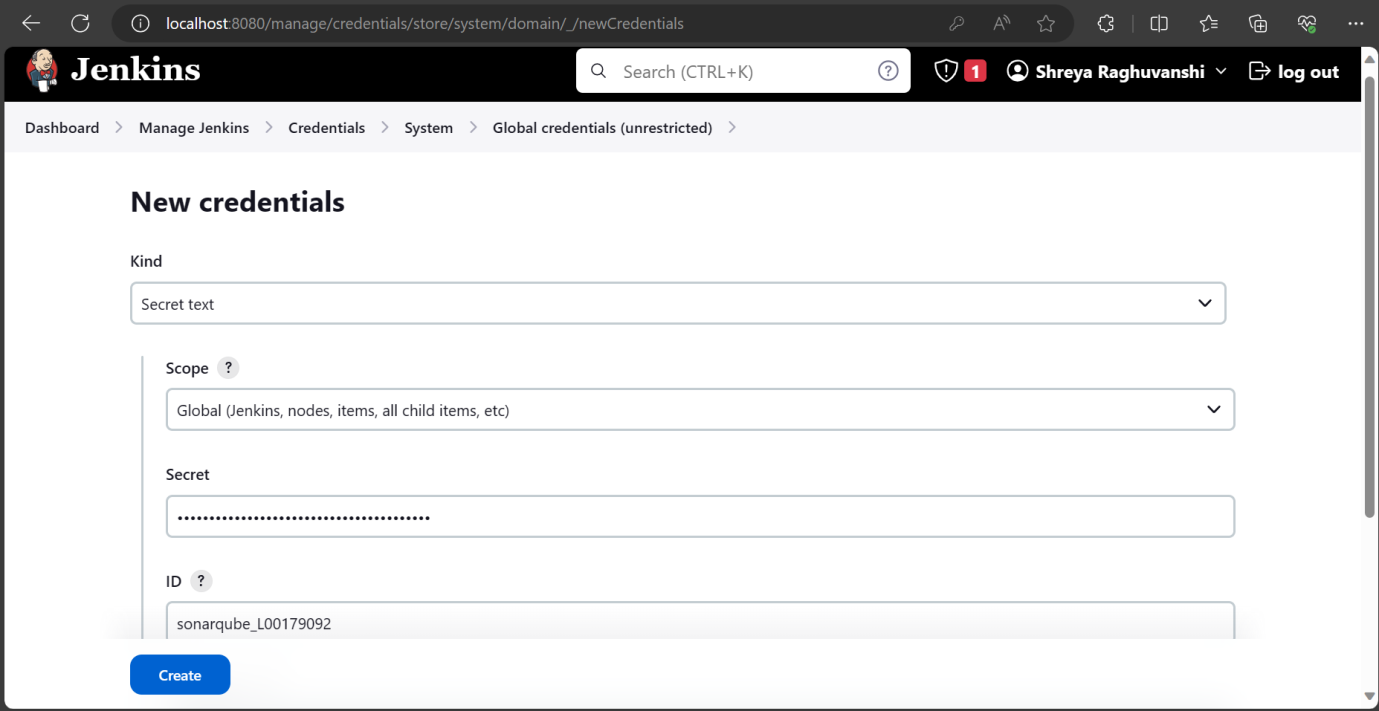
1. Sonarqube was opened on localhost:9000 and then logged in using username and password. By default, username and password was admin.



1. To establish authentication between both Jenkins and Sonarqube, Sonarqube was opened and ‘my account’ was selected from the right. Inside my account, Security option was clicked and a new token of choice was created by clicking on Generate button beside textbox. After which, the token was copied using copy button.



1. Next Jenkins was opened and new credentials were added in Global credentials section. In the Kind, ‘Secret text’ option was selected and secret key which was copied from SonarQube was pasted here. Then ‘Create’ button was clicked.



Output:



**Following is the final script which was pasted in the ‘L00179092\_jenkins\_pipeline’ file on GitHub after Sonarqube was set up:**

stage('Static Code Analysis') {

environment {

SONAR\_URL = "http://localhost:9000"

}

steps {

withCredentials([string(credentialsId: 'sonarqube\_L00179092', variable: 'SONAR\_AUTH\_TOKEN')]) {

bat 'cd Mvn\_L00179092 && mvn sonar:sonar \

-Dsonar.projectKey=WebApp\_L00179092 \

-Dsonar.host.url=http://localhost:9000 \

-Dsonar.login=838a116cf359d3ec525b15388ea72fd7b3897a0a'

}

}

}

Below is the credentials for sonarqube integrated inside Jenkins:



1. **Maven (Automated Build and Test):**

**What is Maven and how it is used in pipeline:**

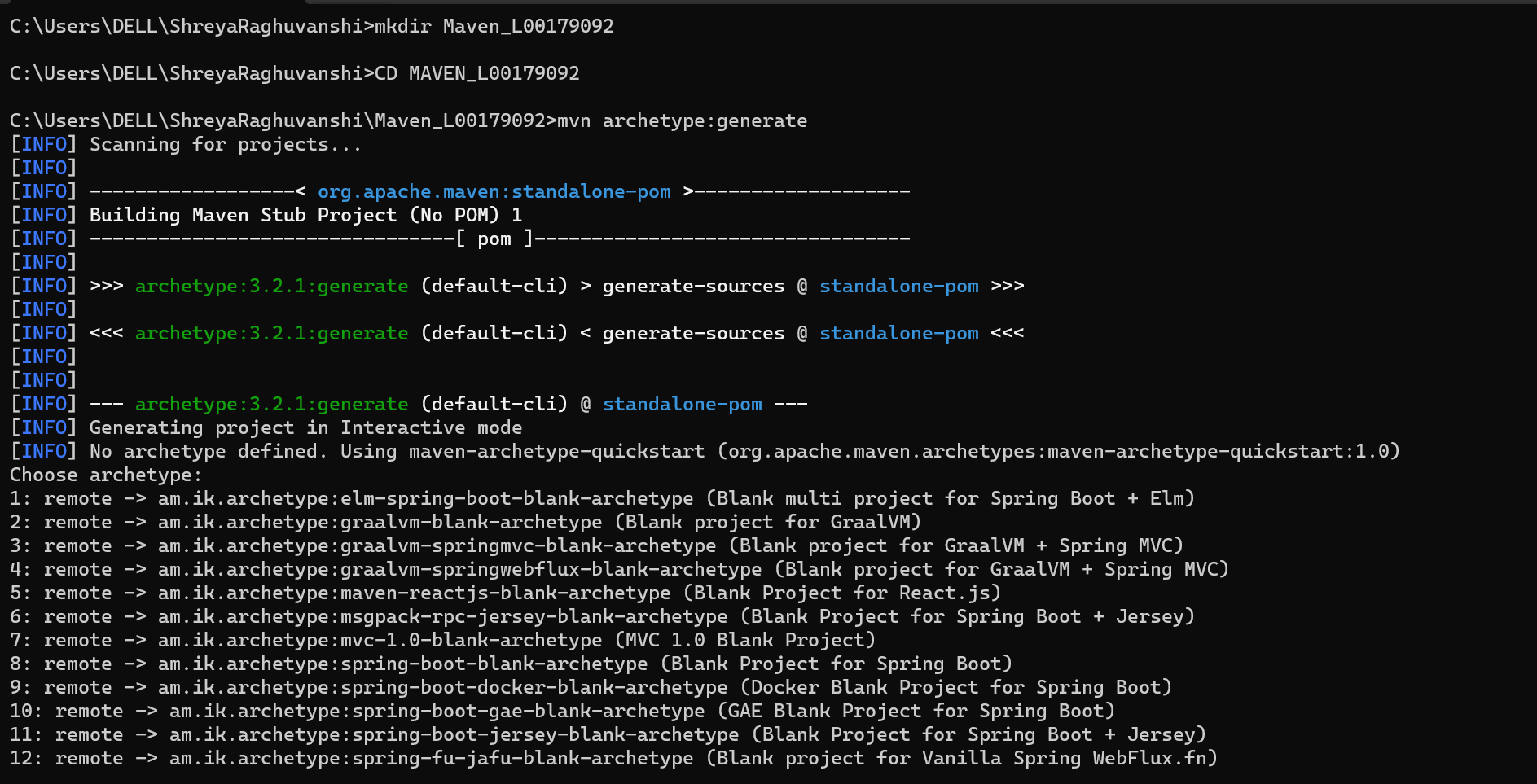
Apache Maven is a tool that simplifies and standardizes the build process. Maven uses a Project Object Model which is represented in an XML file called pom.xml which consists of dependencies and plugins. In a (CI/CD) pipeline, Maven was used as a build automation tool to manage and execute various tasks related to compiling code, running tests, packaging artifacts, and handling dependencies in Java-based projects.

**Reasons for choosing Maven:**

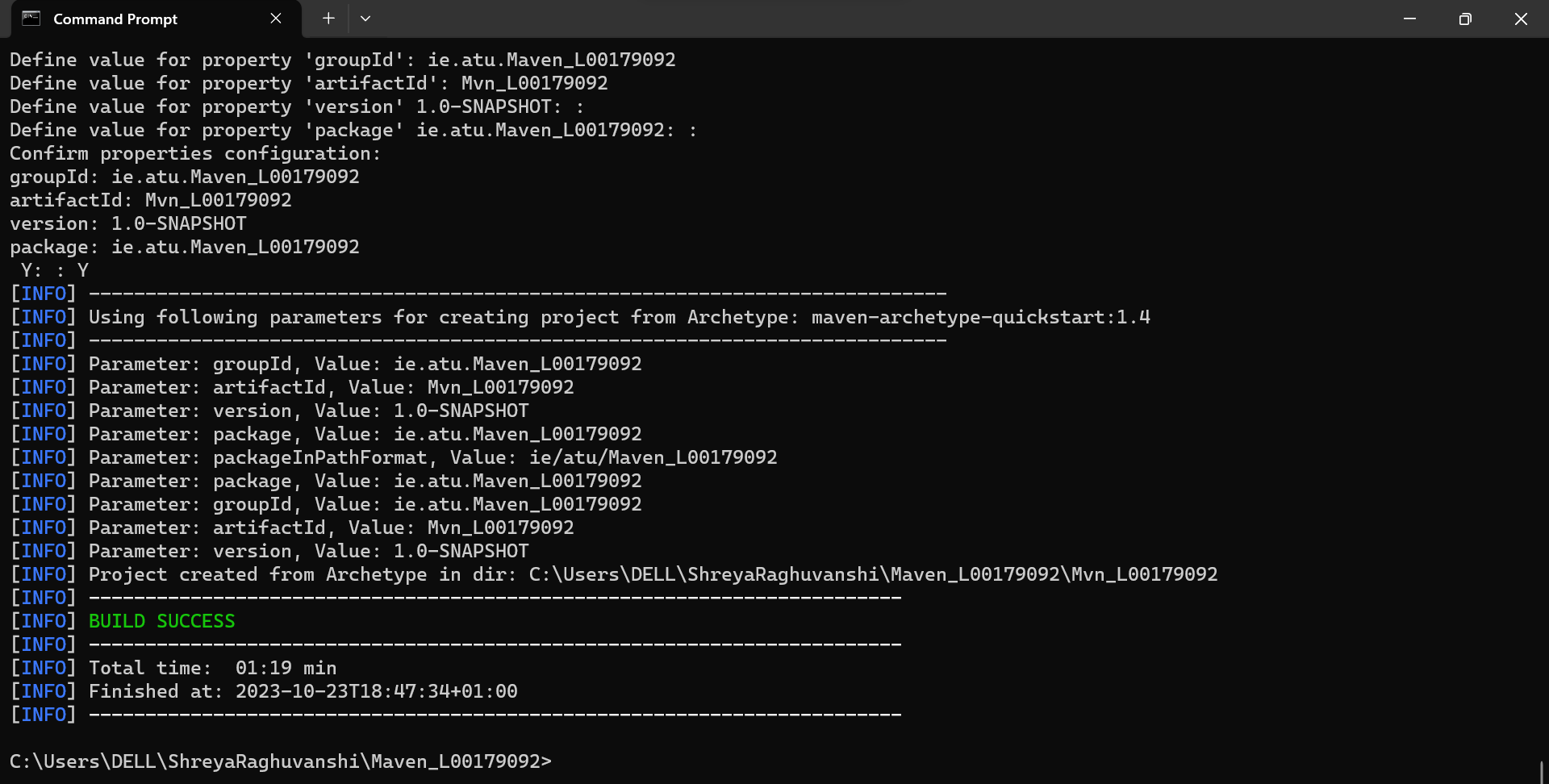
The main reason of using Maven is that it supports projects made through java. It has easy pre-defined phases such as compile, test, install, etc. A facility of custom plugins are also accompanied in Maven. It can automatically download required dependencies which are mentioned in the pom.xml file, from repositories like Maven Central.

**Steps for Maven configuration with Jenkins:**

1. As maven was being used for the building the application in pipeline, new maven project was created using CMD:

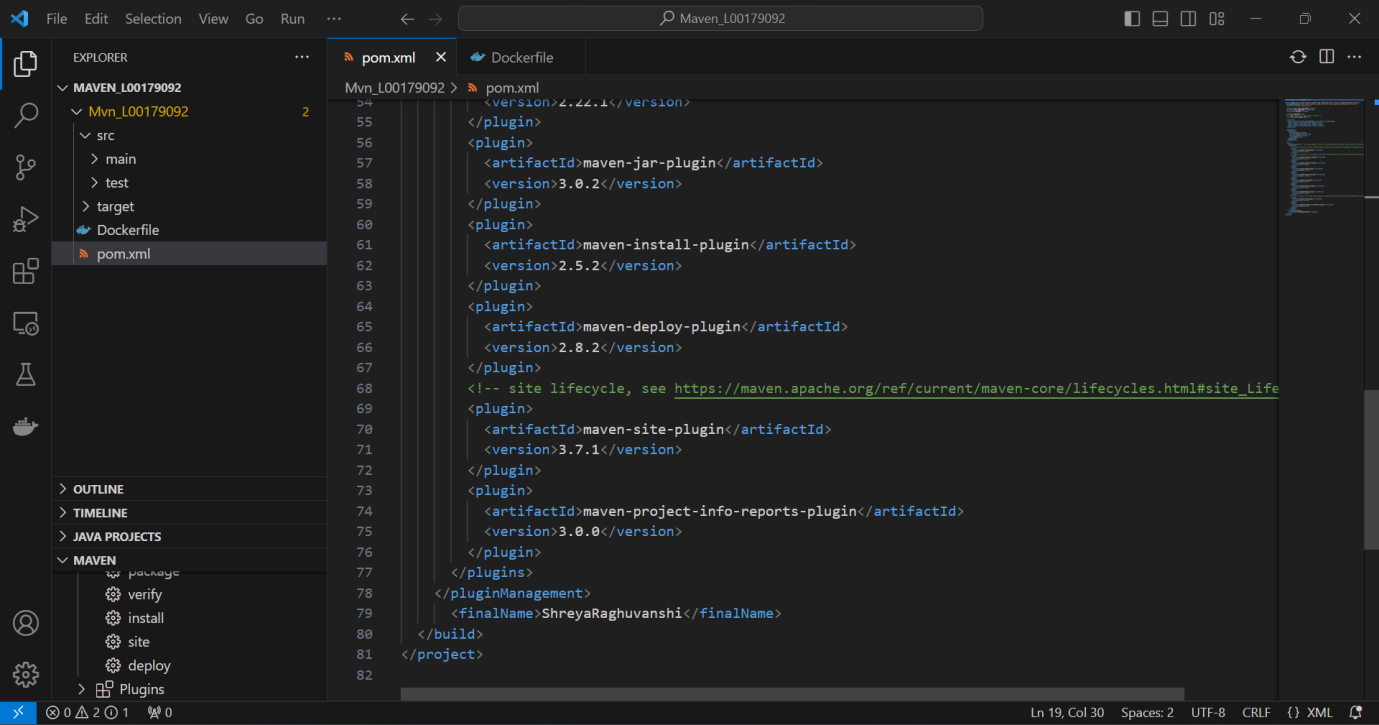


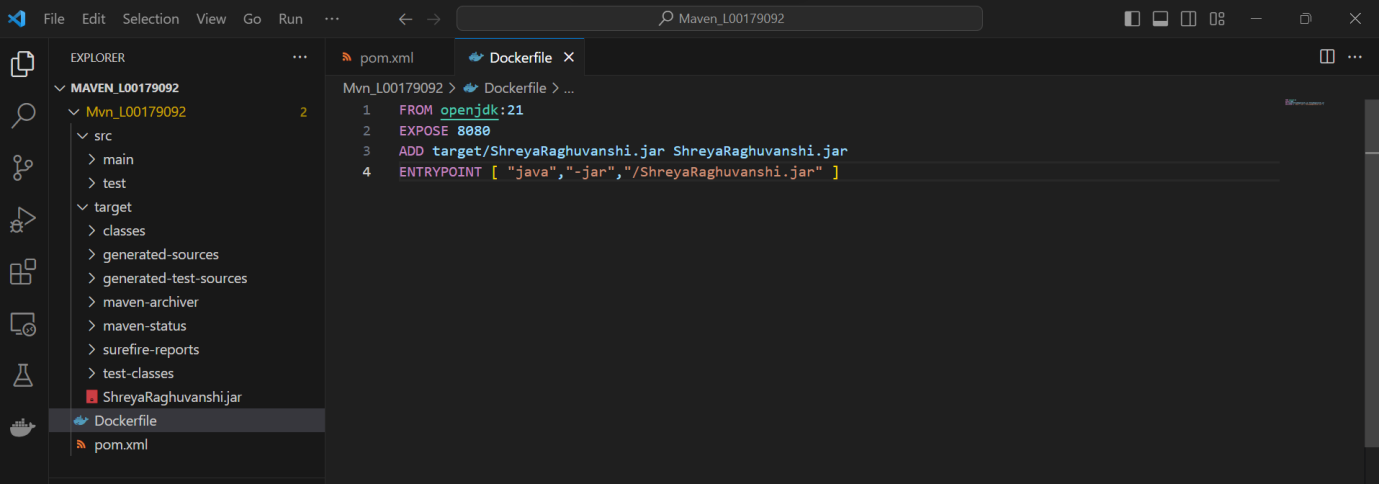
Group ID and artifact ID was specified and the maven project build was successful:



1. Maven project was opened in VS code and Dockerfile is created.

<finalname> ShreyaRaghuvanshi</finalname> was added in the pom.xml file as it specifies the name of the jar file maven was supposed to build.

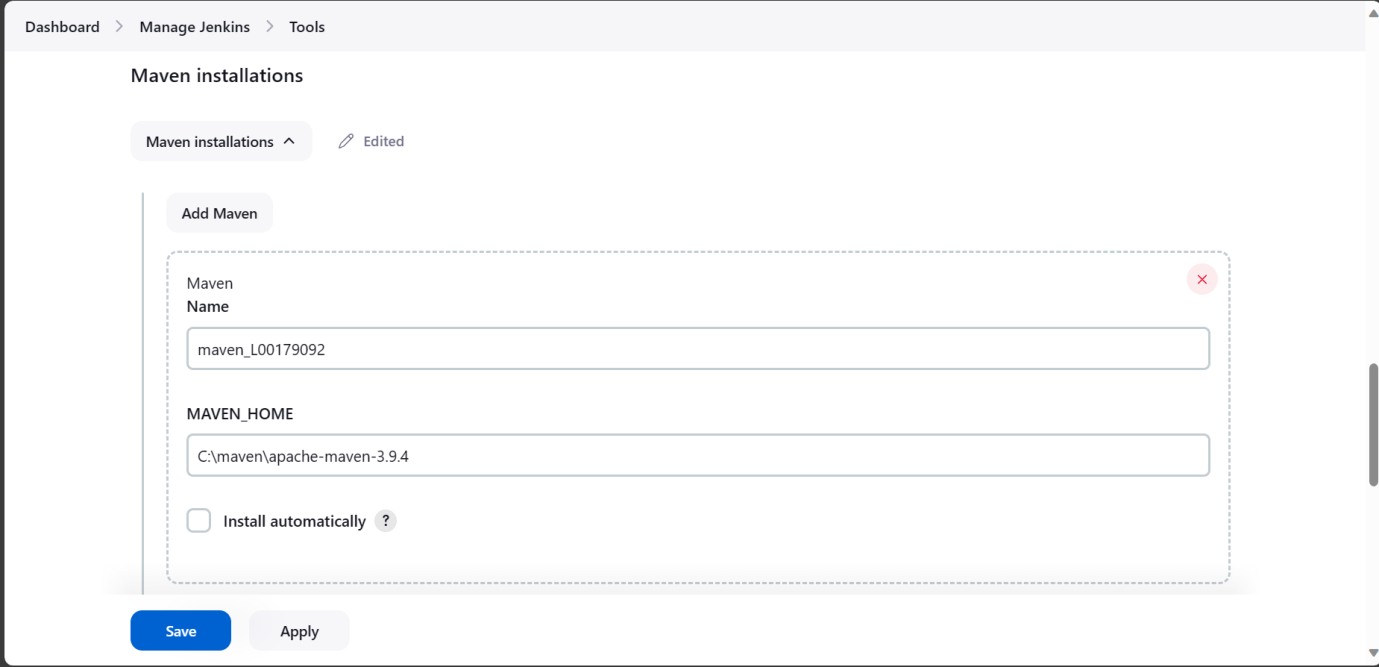




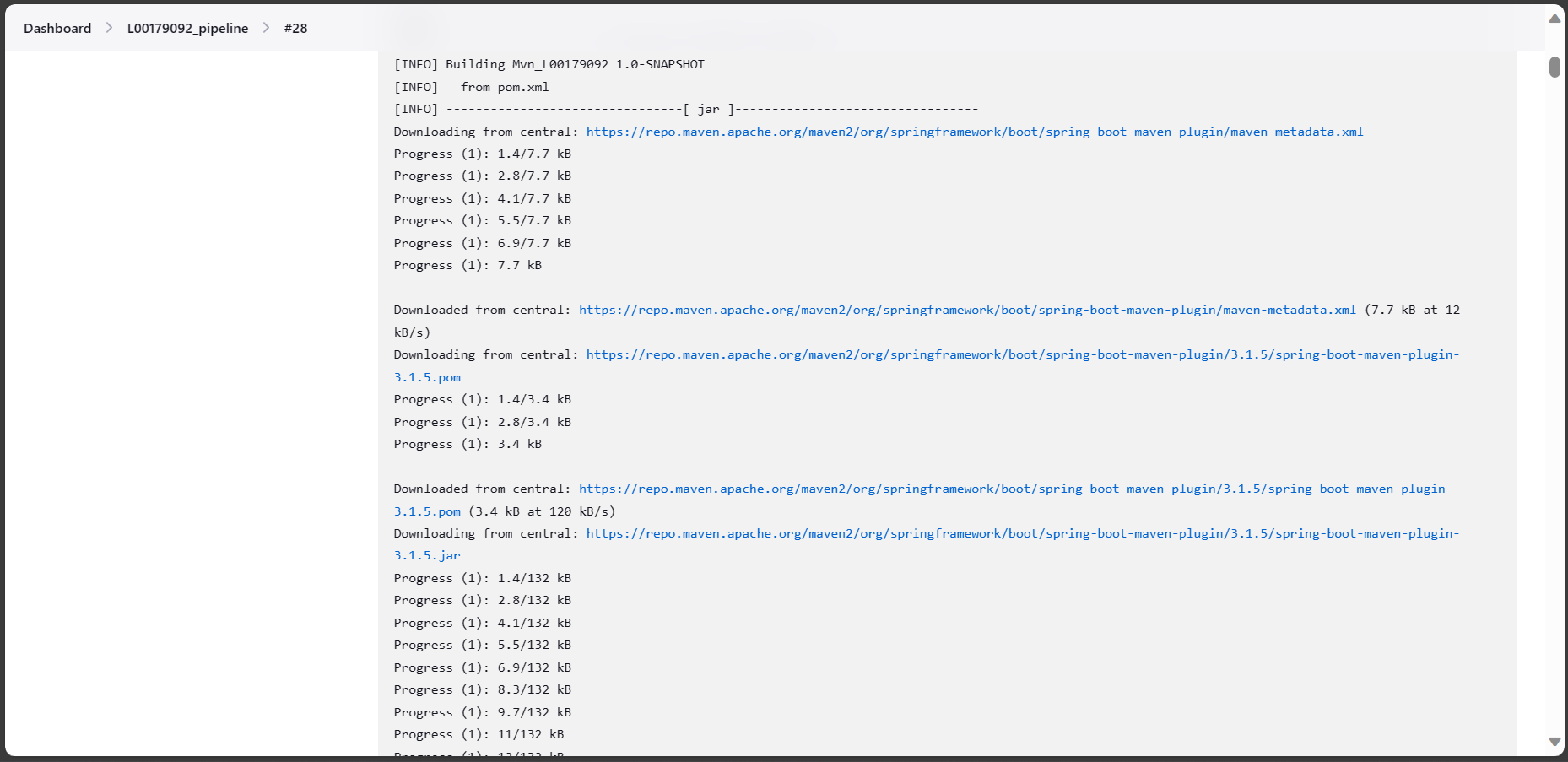
1. This maven project was pushed into the GitHub repository with the following commands:

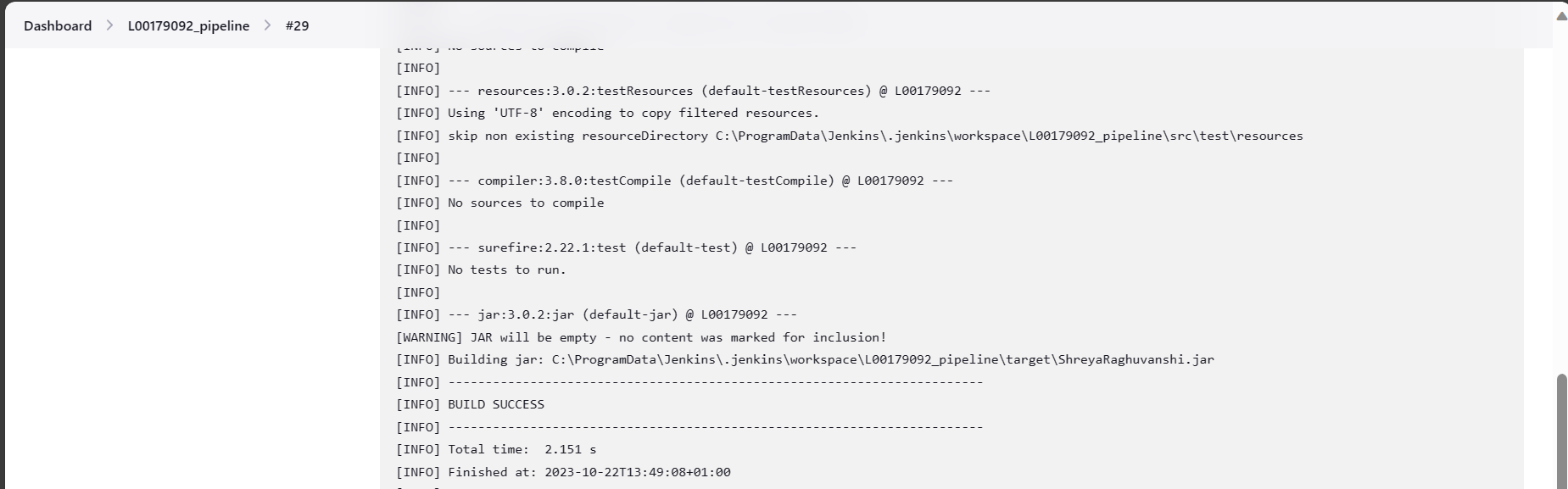
* git init > to initialized git repository
* git add . > to add all the files to staging
* git commit –m “new maven files added” > to commit changes
* git remote add origin [git@github.com:ShreyaRaghuvanshi1/L00179092\_pipeline.git](mailto:git@github.com:ShreyaRaghuvanshi1/L00179092_pipeline.git) > to know which repository to connect to
* git push origin main > to push all the maven files into GitHub

1. For jenkins to know that maven is installed in the local system, path for it was added in the tools section.



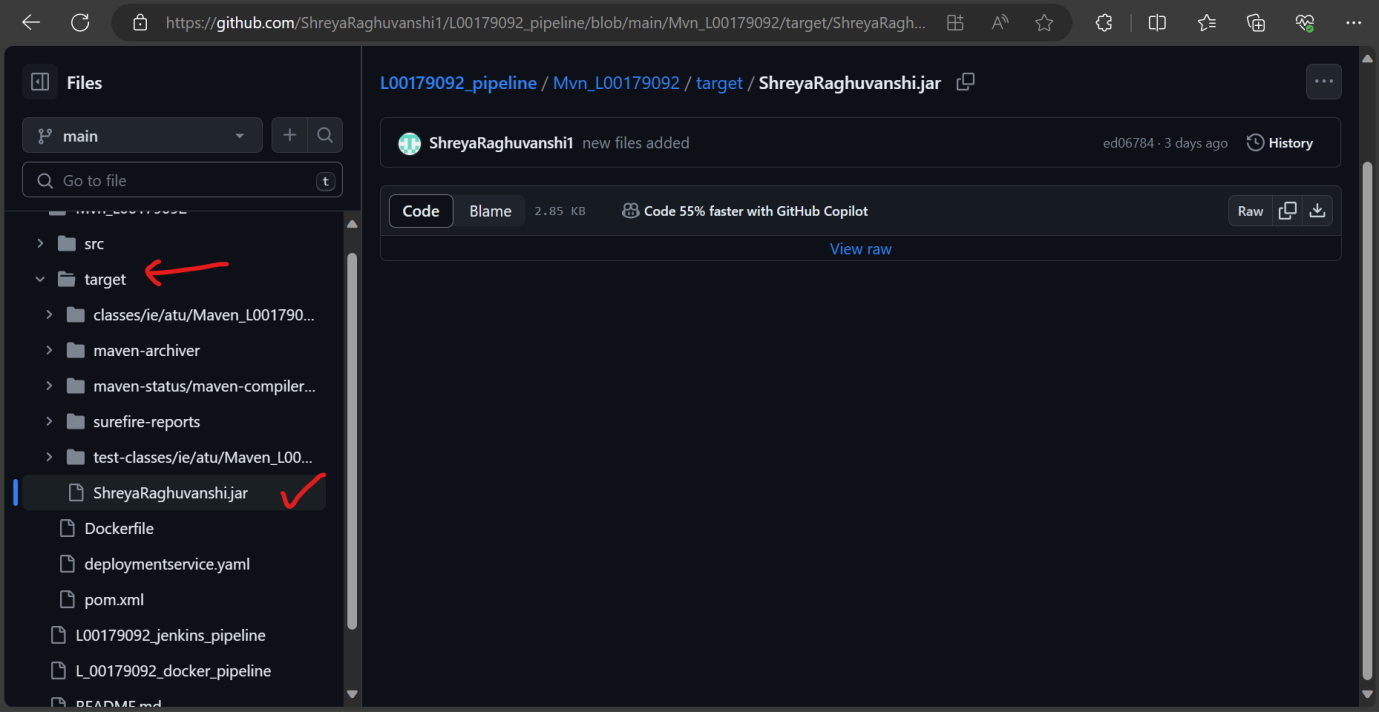
1. After building the pipeline, shreya-raghuvanshi.jar file was successfully created and all the plugins were downloaded





Output:

As mentioned in the docker file, jar file was supposed to be saved inside the target folder in GitHub:



**Following is the final script which was pasted in the ‘L00179092\_jenkins\_pipeline’ file on GitHub after Maven was set up:**

stage('Build and Test') {

steps {

bat 'cd Mvn\_L00179092 && mvn clean package'

}

}

1. **Docker(Building docker images - Deployment):**

**What is Docker and how it is used in pipeline:**

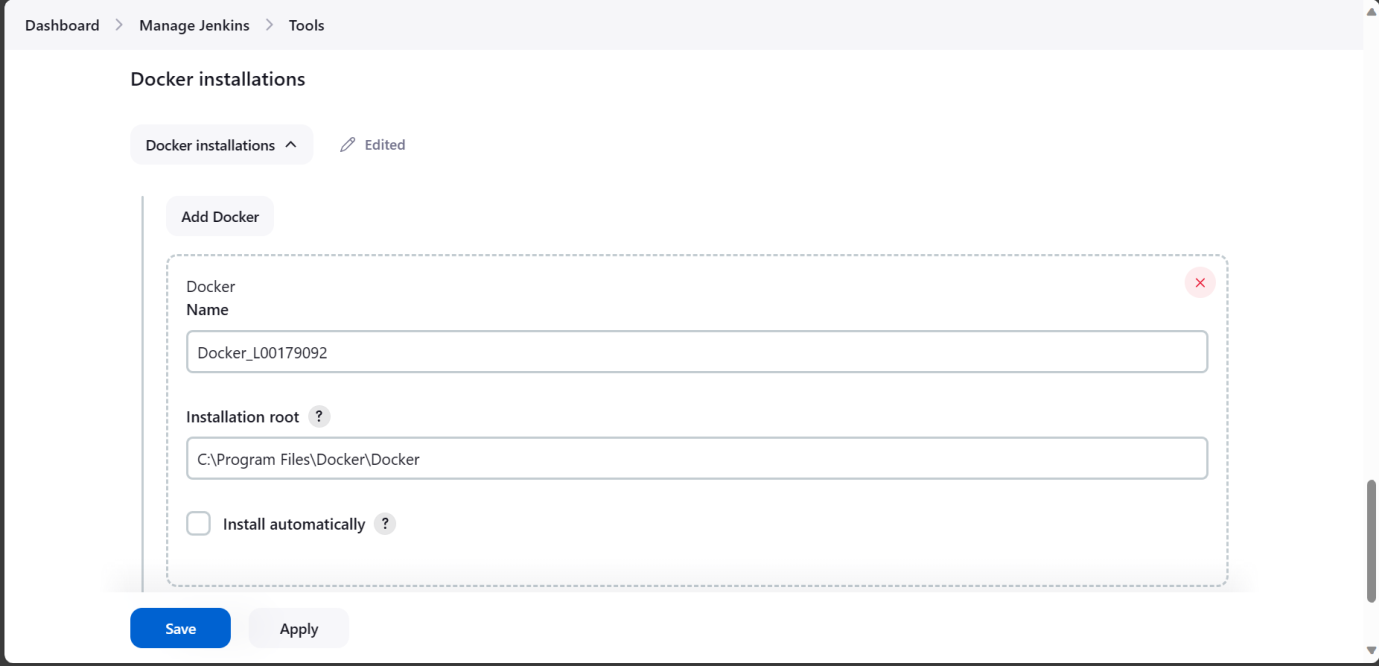
Docker is an open source containerization platform used to develop and deploy applications in virtualized environments called containers. Docker provides a standardized environment encapsulated in containers, ensuring consistency across the various stages of a CI/CD pipeline. In this pipeline, Docker is used to creae docker images and push those images from docker desktop to docker hub.

**Reasons for choosing Docker:**

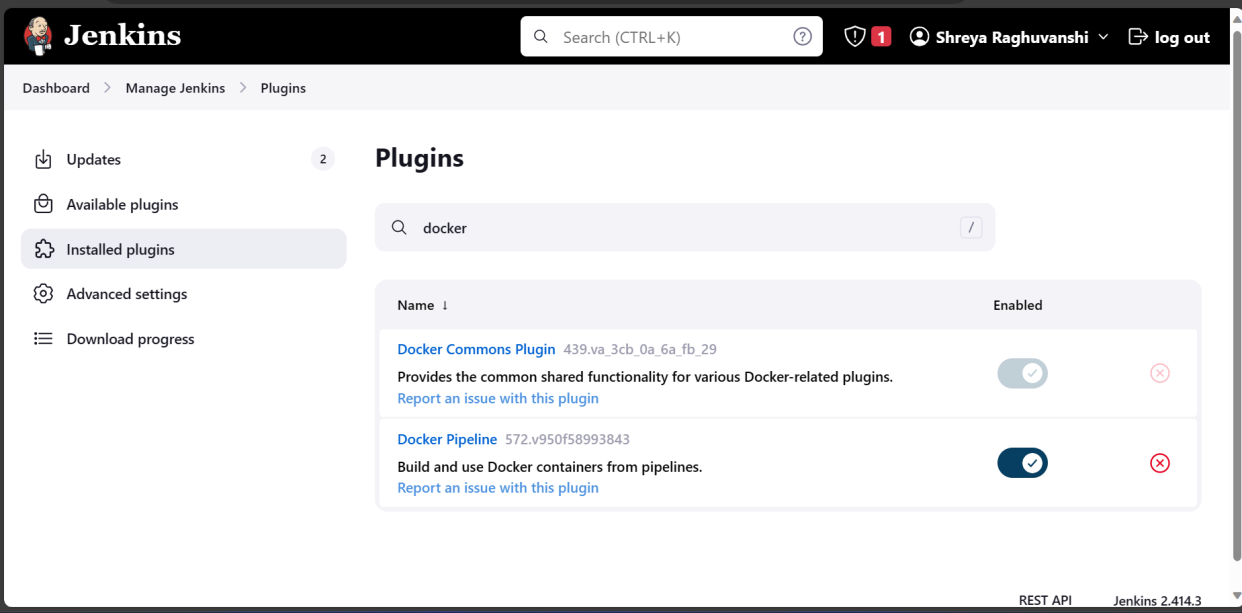
The reason for choosing docker over other tool is that Docker containers start quickly and can be scaled up or down easily and it can be easily integrated with Jenkins. Docker containers encapsulate applications and their dependencies, ensuring consistent behavior across different environments. This portability eliminates the "it works on my machine" problem and streamlines deployments.

**Steps for Docker configuration with Jenkins:**

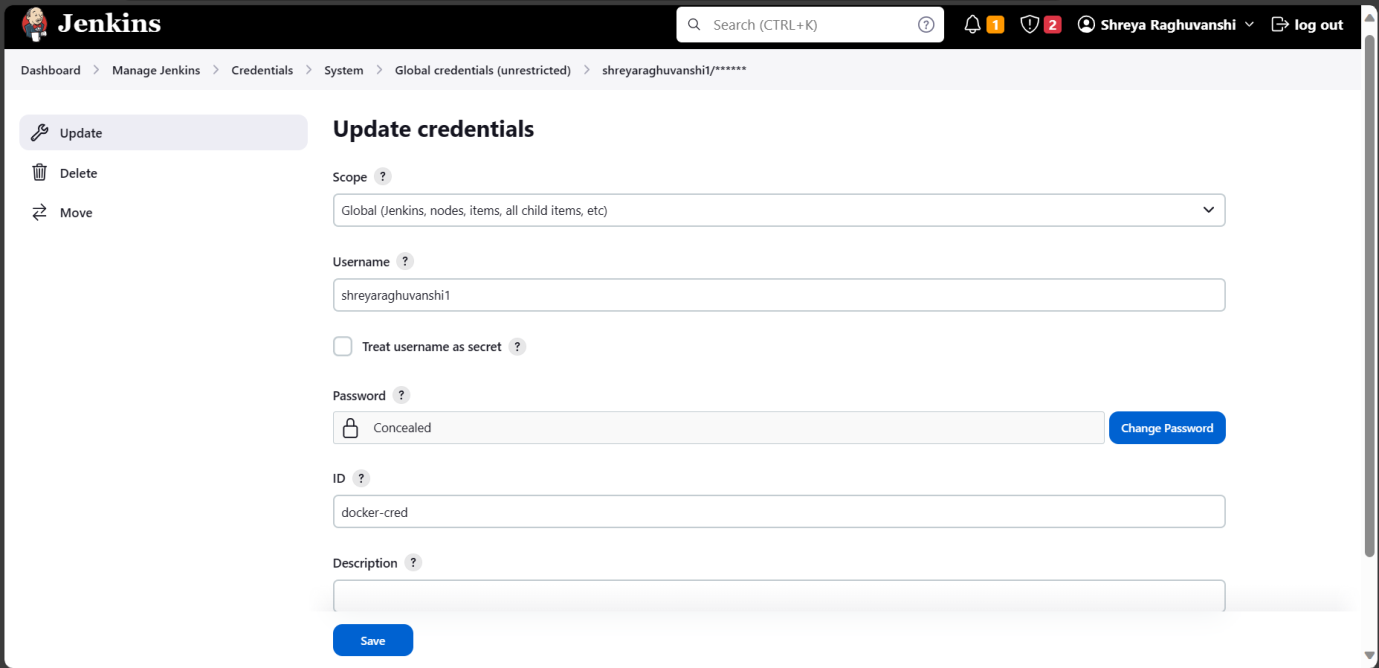
1. Next step was to build image in Docker. For this, Docker was installed in the system and path for it was specified on Jenkins



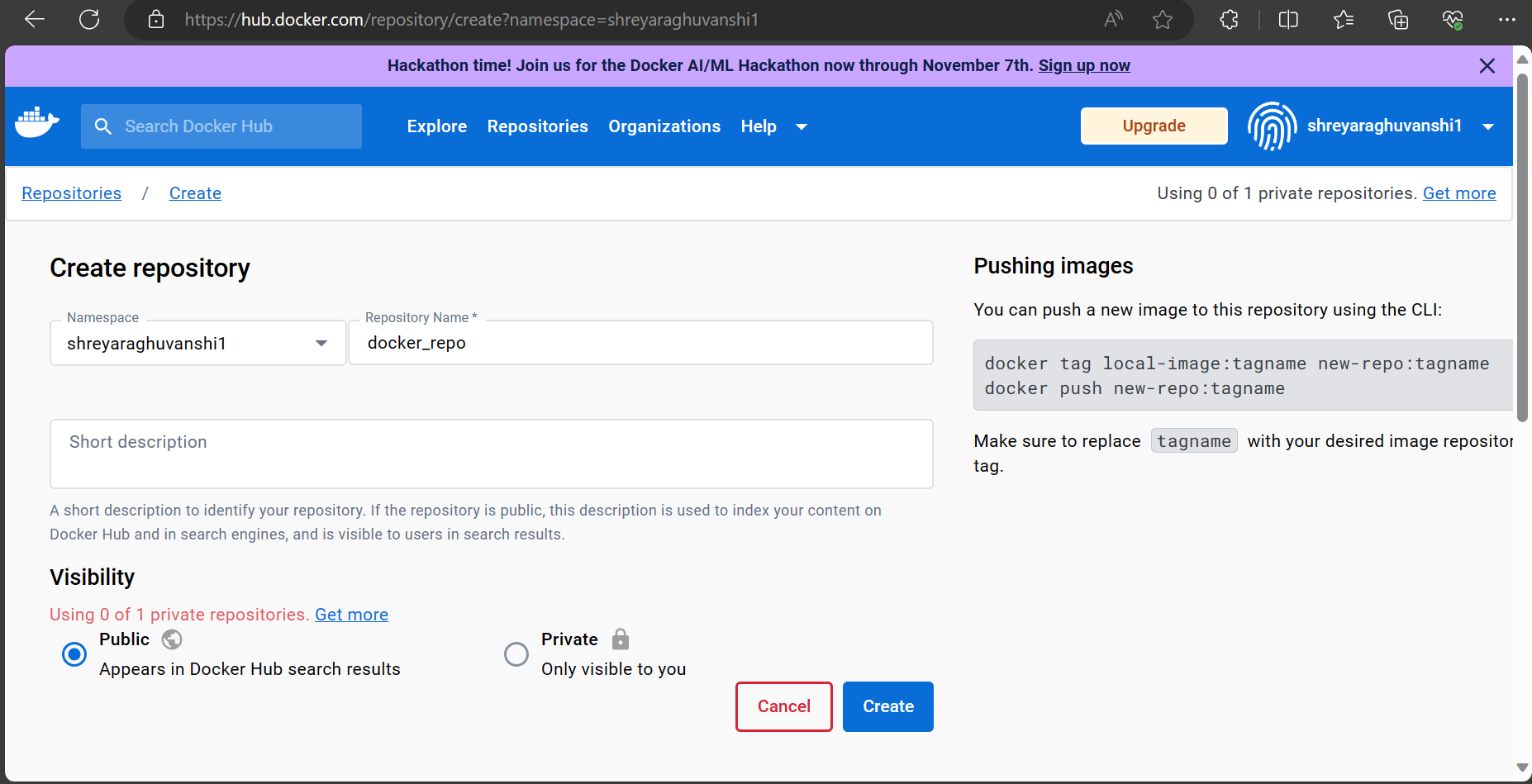
1. Plugin for Docker was installed in Jenkins



1. Credentials for docker was added in the Global credentials of Jenkins:



1. For creating image in dockerHub, a new repository was created first



Output:



**Following is the final script which was pasted in the ‘L00179092\_jenkins\_pipeline’ file on GitHub for building the docker image :**

stage('Build Docker Image') {

environment {

REGISTRY\_CREDENTIALS = credentials('docker-cred')

}

steps {

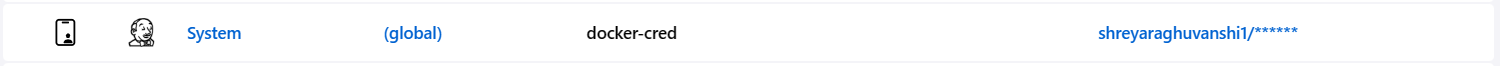
script {

bat 'cd Mvn\_L00179092 && docker build -t docker/shreya-raghuvanshi .'

}

}

}

Below is the credentials for Docker which was integrated in jenkins:

1. Next step was to push that docker image to DockerHub:

**Following is the final script which was pasted in the ‘L00179092\_jenkins\_pipeline’ file on GitHub for pushing the docker image on DockerHub:**

stage('Push Docker Image'){

environment {

//DOCKER\_IMAGE = "shreya-raghuvanshi ."

REGISTRY\_CREDENTIALS = credentials('docker-cred')

}

steps{

script {

docker.withRegistry('https://index.docker.io/v1/', "docker-cred"){

bat "docker tag docker/shreya-raghuvanshi:latest shreyaraghuvanshi1/docker\_repo:latest"

bat "docker push shreyaraghuvanshi1/docker\_repo:latest"

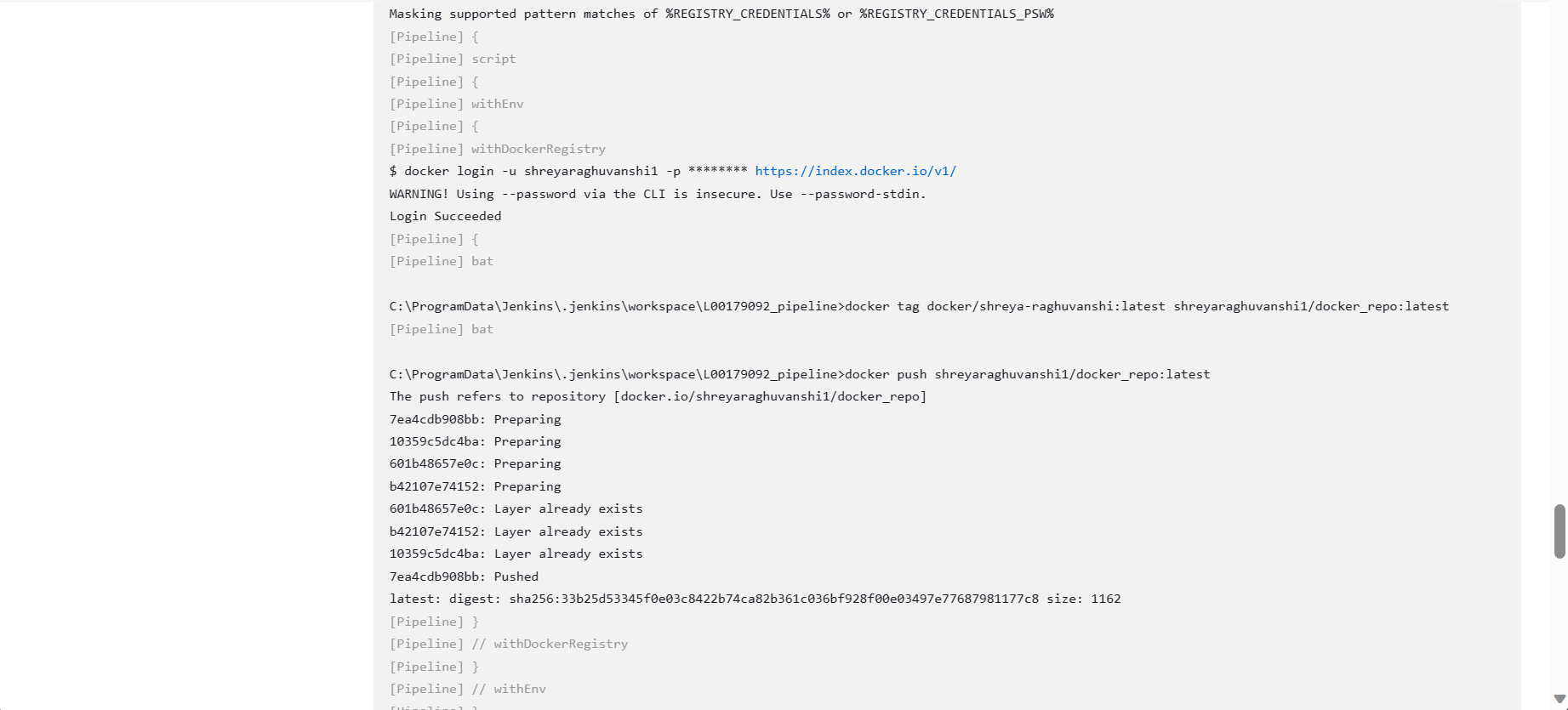
}

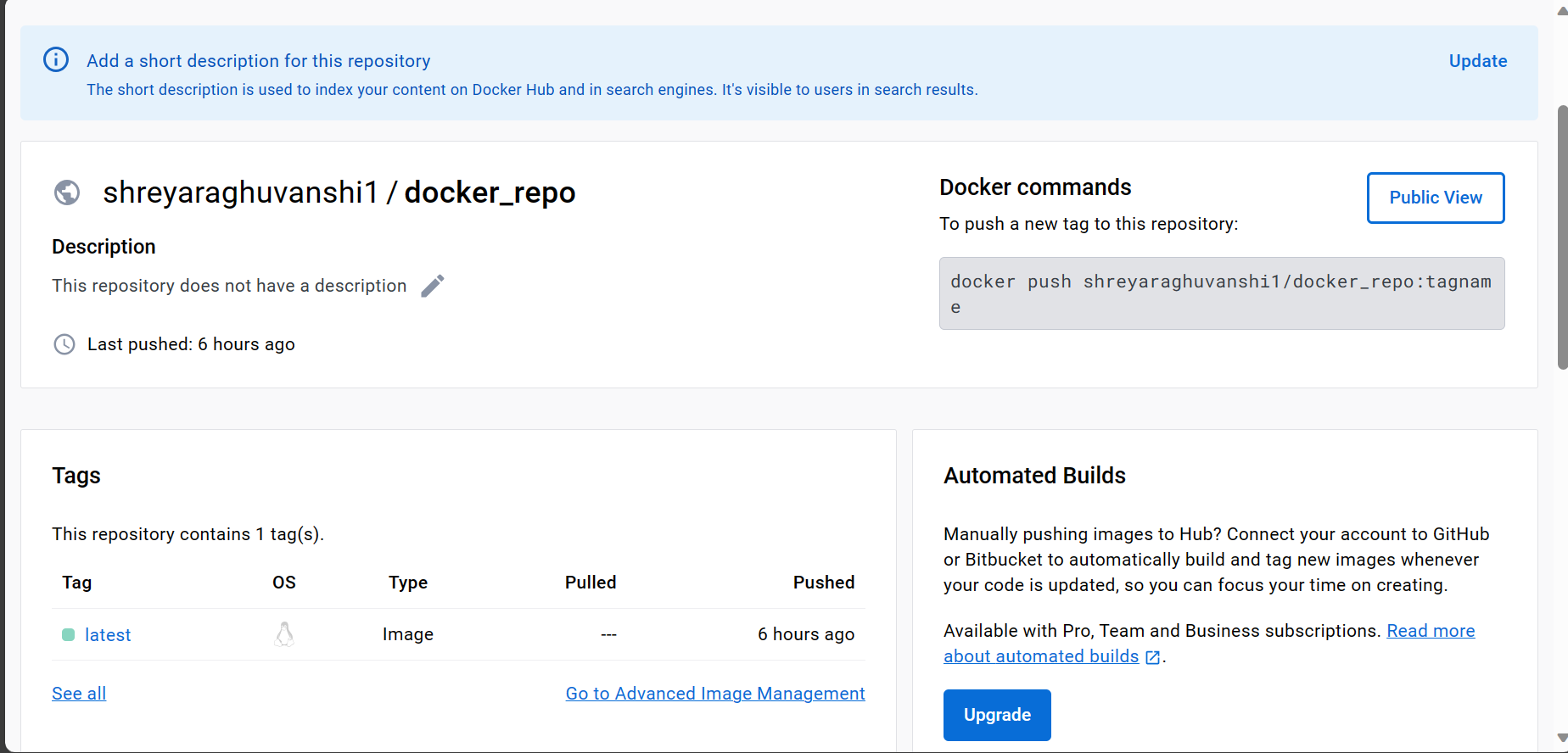
}

}

}

Output:





1. **Kubernetes(Automated Delivery):**

**What is Kubernetes and how it is used in pipeline:**

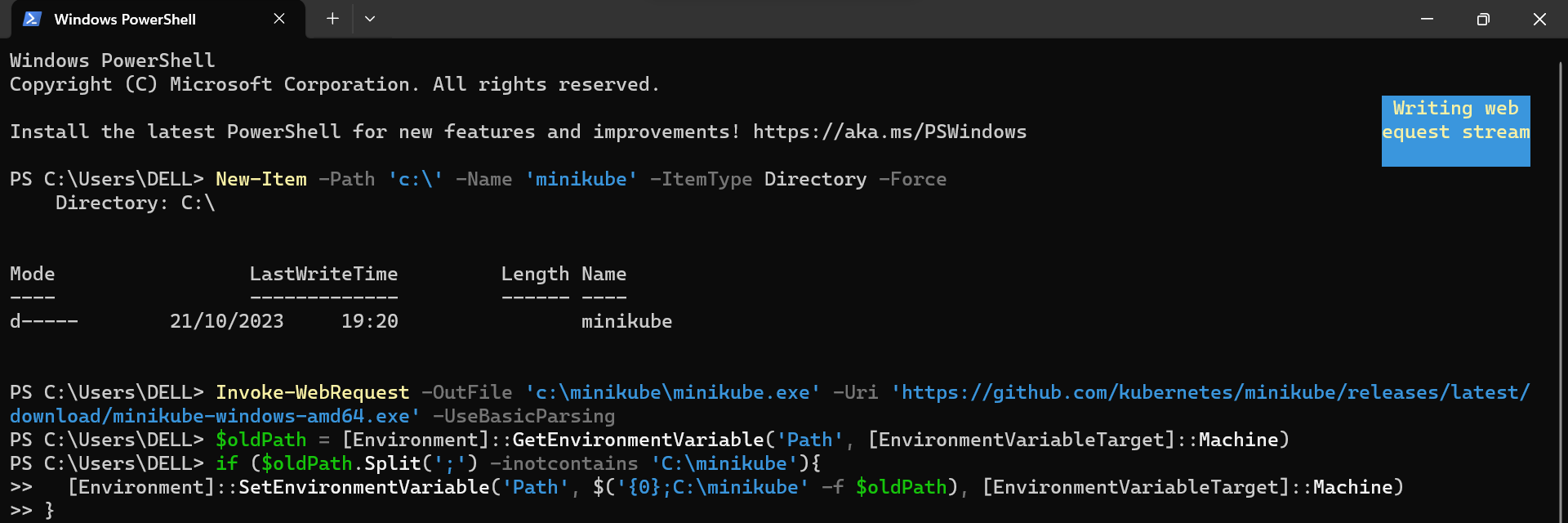
Kubernetes is an extensible and portable platform for managing containerized workloads and services. It provides a strong framework for handling containers across a cluster of machines. Users package their applications and services into Docker containers, which encapsulate the application along with its dependencies. It was used in CI/CD pipeline for receiving images from docker hub and then deploying it.

**Reasons for choosing Kubernetes:**

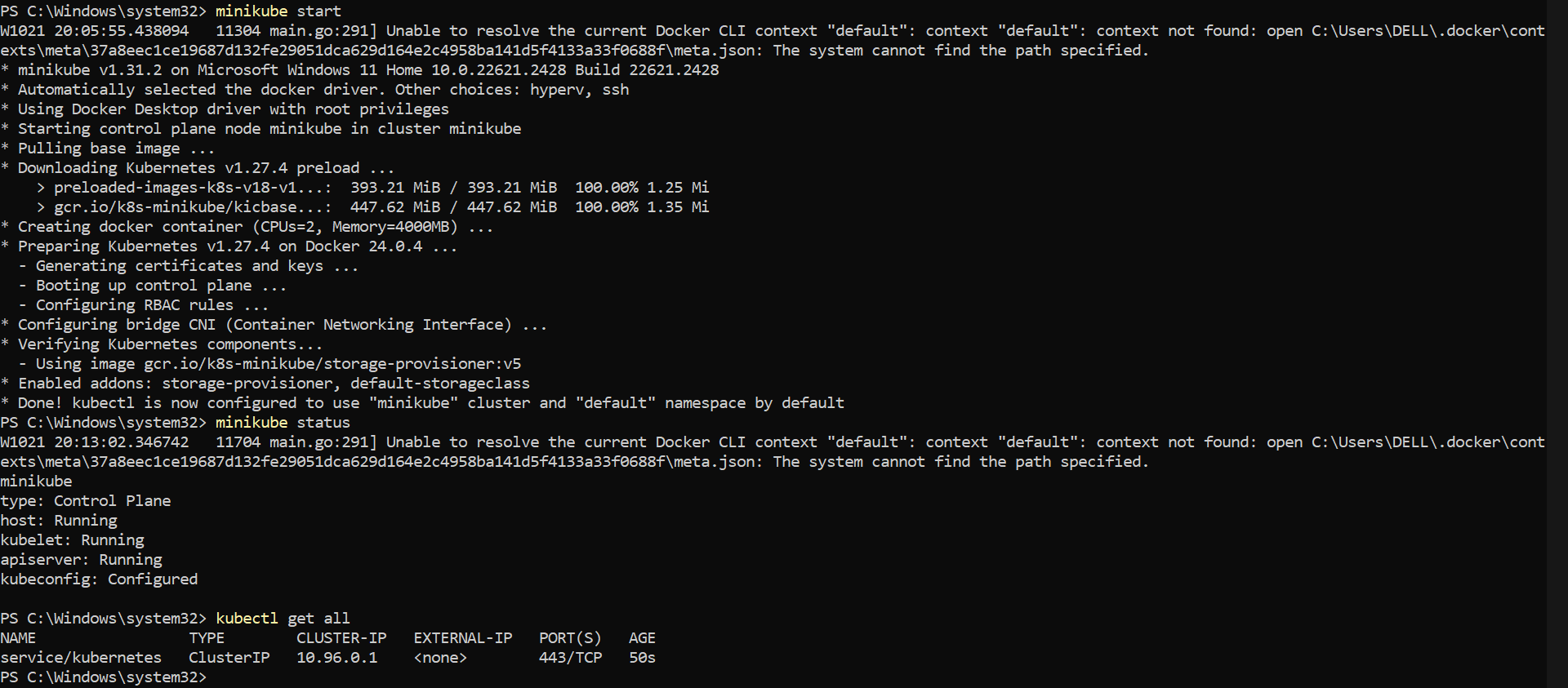
The reason for choosing Kubernetes was that it is open-source. It also has self-healing capabilities which indicates that it restarts, re-schedules and replaces the containers when it needs. It provides built-in capabilities for service discovery and load balancing.

**Steps for Kubernetes configuration with Jenkins:**

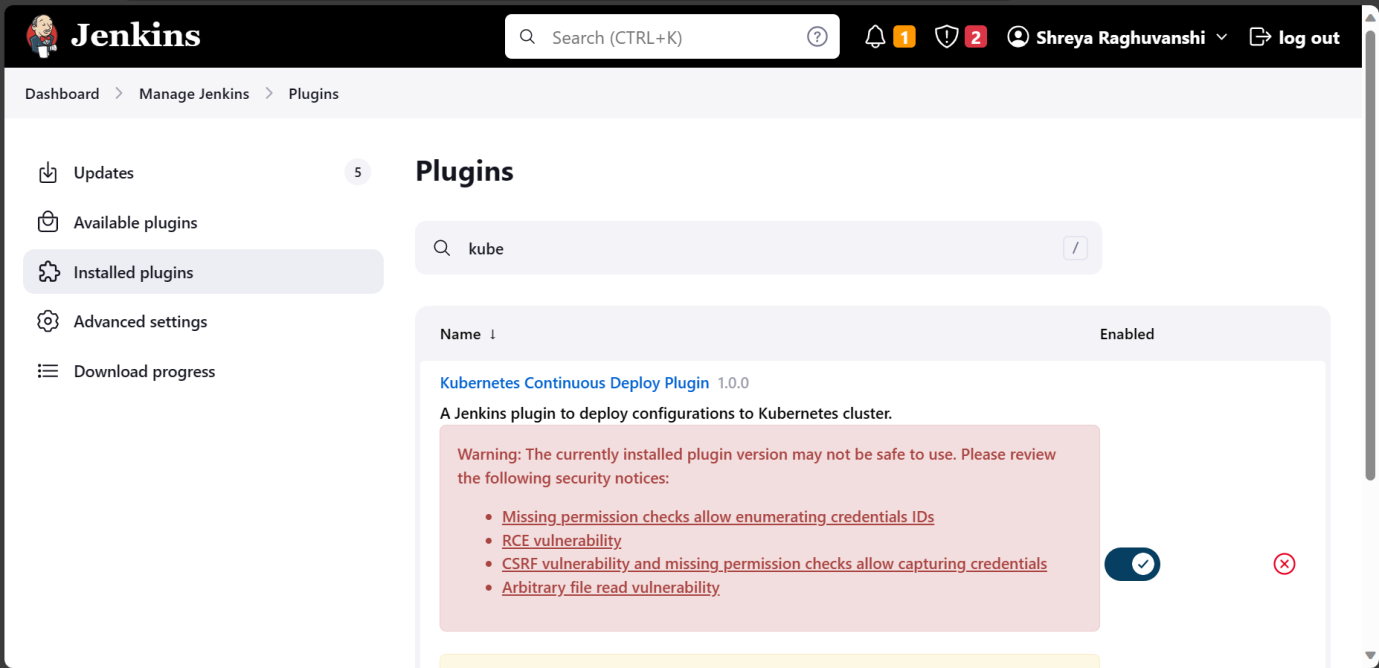
1. The last stage was to push image from DockerHub to Kubernetes for deployment: For kubernetes deployment, minikube was installed. To download minikube the following commands in windows powershell was performed:



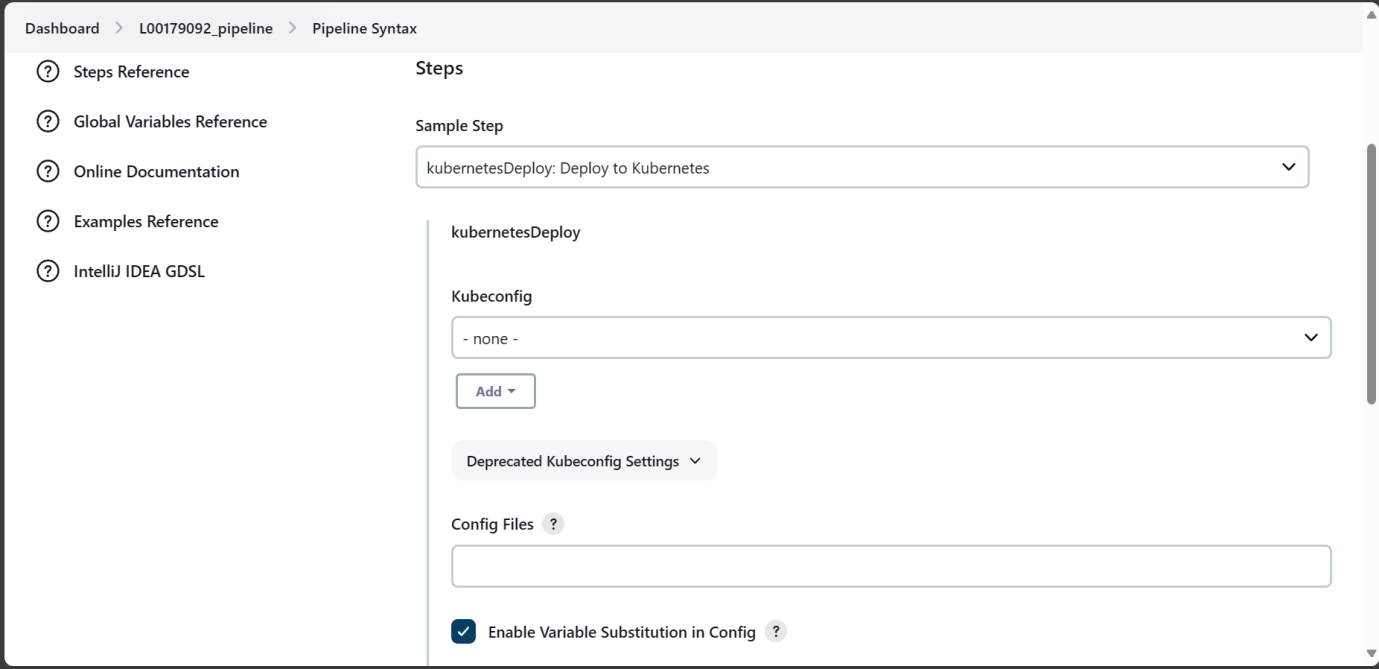
1. ‘Minikube start’ command is used to start the download of kubernetes and to view the status of minikube ‘minikube status’ was used.



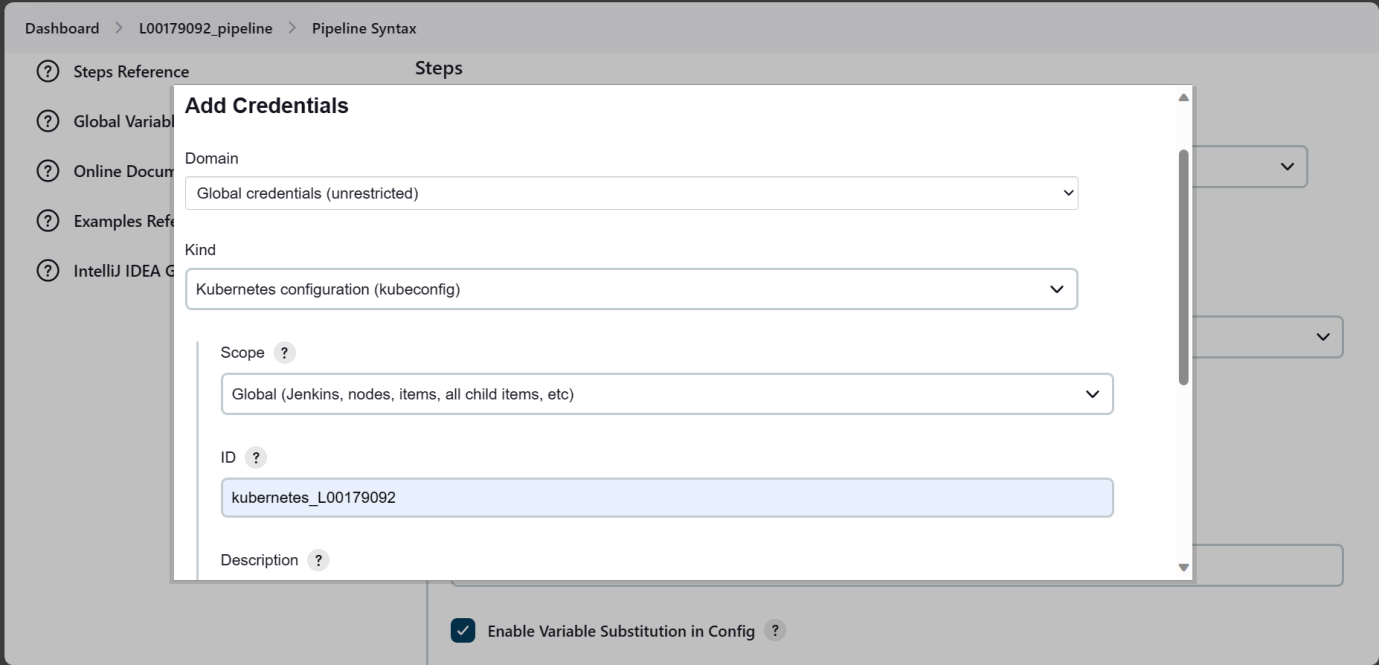
1. To integrate kubernetes into jenkins pipeline, kubernetes plugin was installed into jenkins.



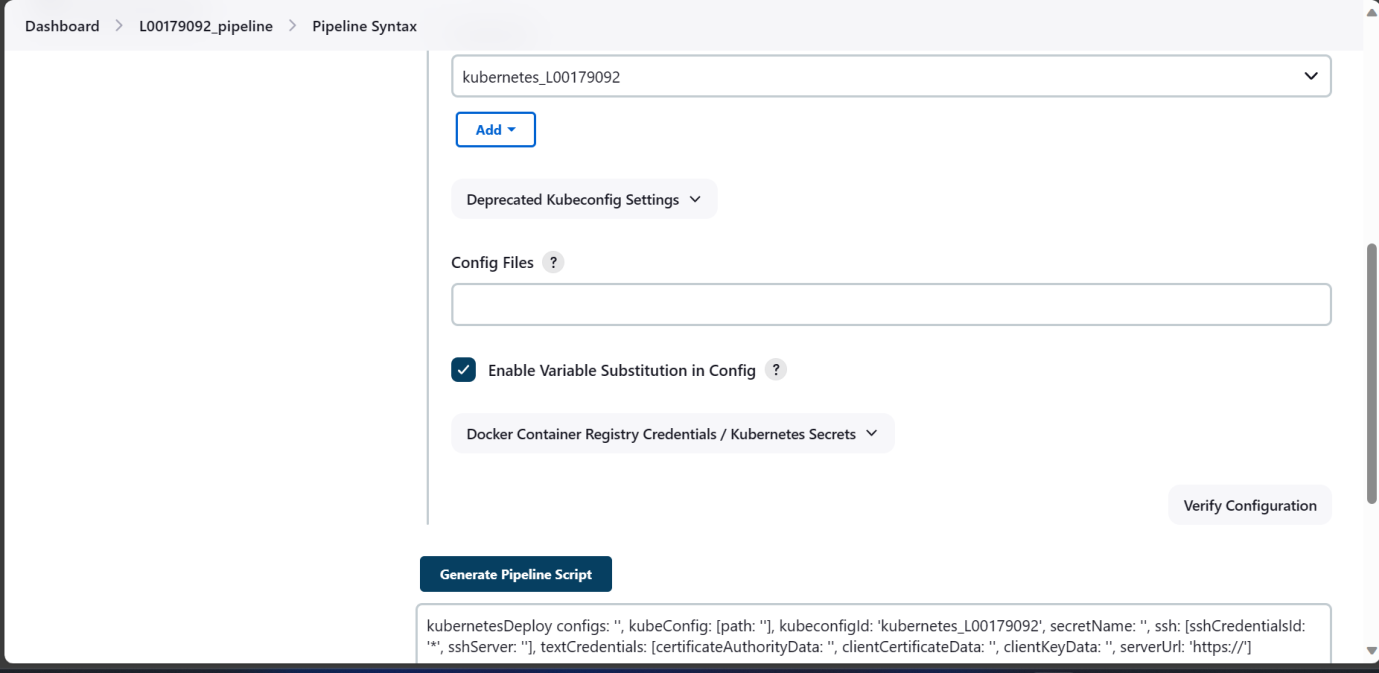
1. To mention deployments and services, a new file ‘deploymentservice.yaml’ was created in VS code.
2. A new stage for kubernetes was added in Jenkins pipeline using pipeline syntax by navigating to the desired pipeline and selecting ‘pipeline syntax’ from the left panel. Following options were chosen:



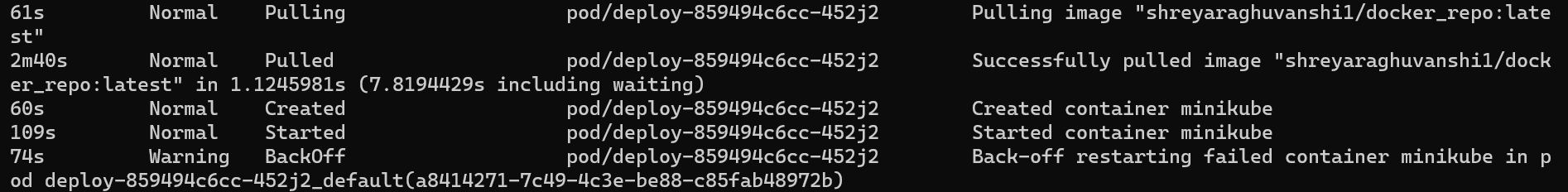
1. In kubeconfig, a new jenkins connector was added and ID of choice was entered:



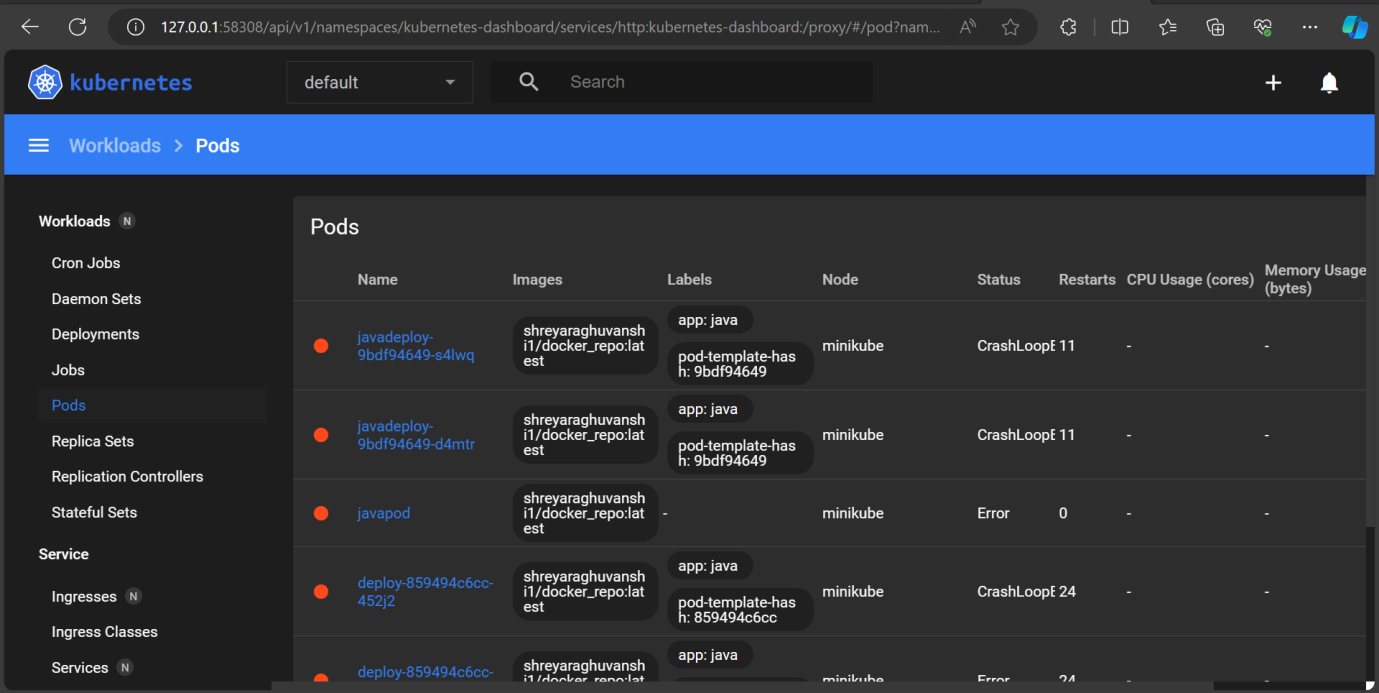
1. Next option was to choose the kubeconfig file. The information under kubeconfig was obtained with the path C:\Users\DELL\.kube\config. Whatever was present in the config file was pasted and then script was generated:



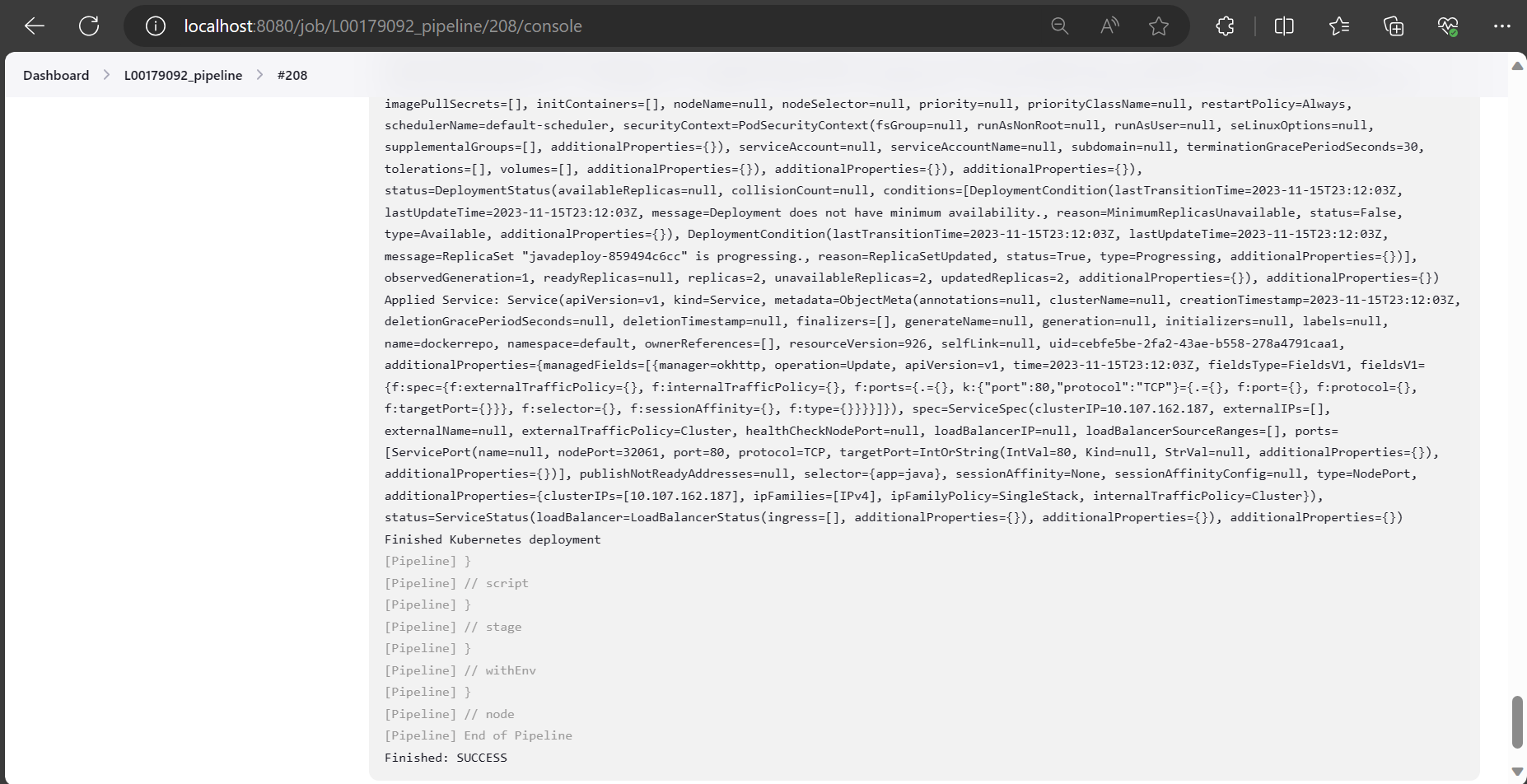
1. To gets the information on images in events, ‘kubectl get events’ command was used in CMD.



1. To check the deployment of docker image on kubernetes dashboard > ‘minikube dashboard’ command was used.



Output:



**Following is the final script which was pasted in the ‘L00179092\_jenkins\_pipeline’ file on GitHub for deploying Docker image to Kubernetes:**

stage('Deploy to Kubernetes'){

steps{

script{

//kubernetesDeploy (configs: 'Mvn\_L00179092/deploymentservice.yaml', kubeconfigId: 'kubernetes\_L00179092',

//dockerCredentials:[[credentialsId: 'docker-cred']])

kubernetesDeploy(

credentialsType: 'KubeConfig',

kubeConfig: [path: 'C:/Users/DELL/.kube/config'],

configs: 'Mvn\_L00179092/deploymentservice.yaml',

dockerCredentials: [

[credentialsId: 'docker-cred'],

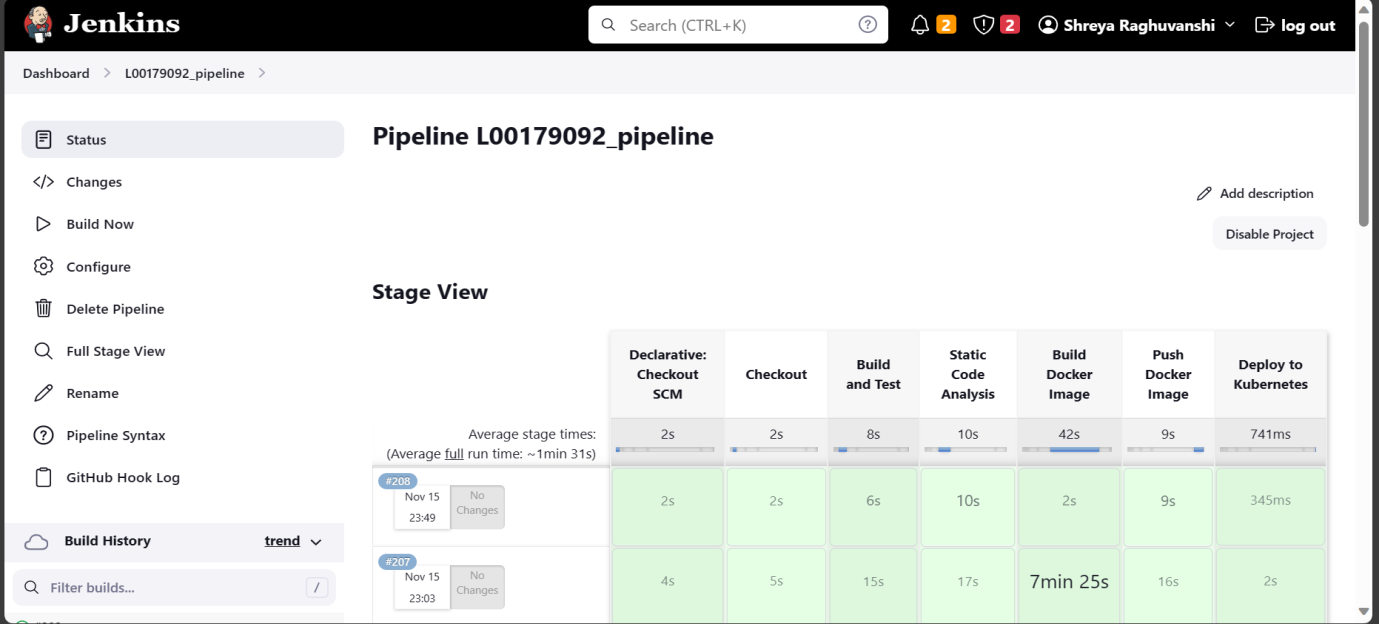
])

} } }

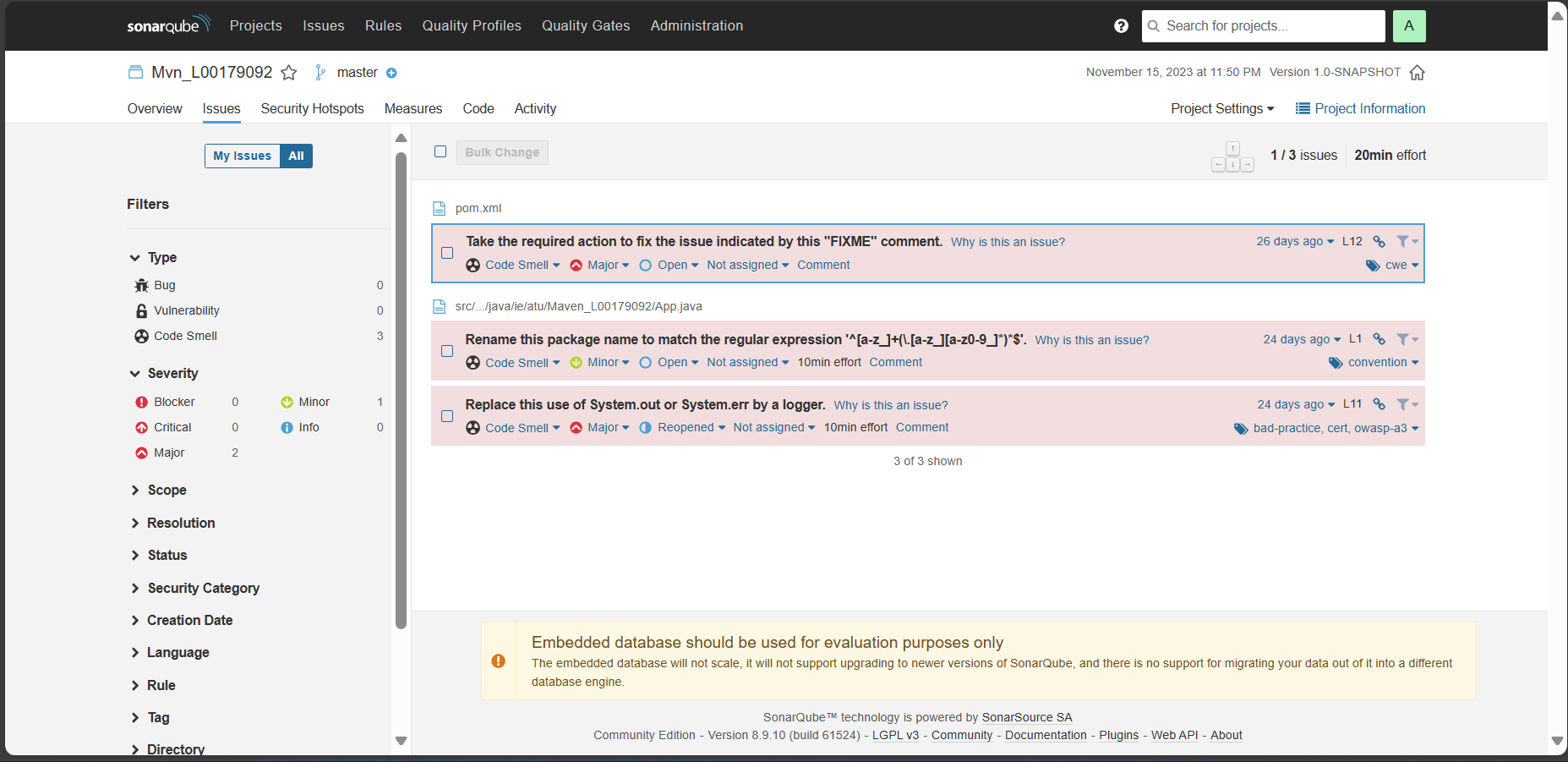
}}

Results:

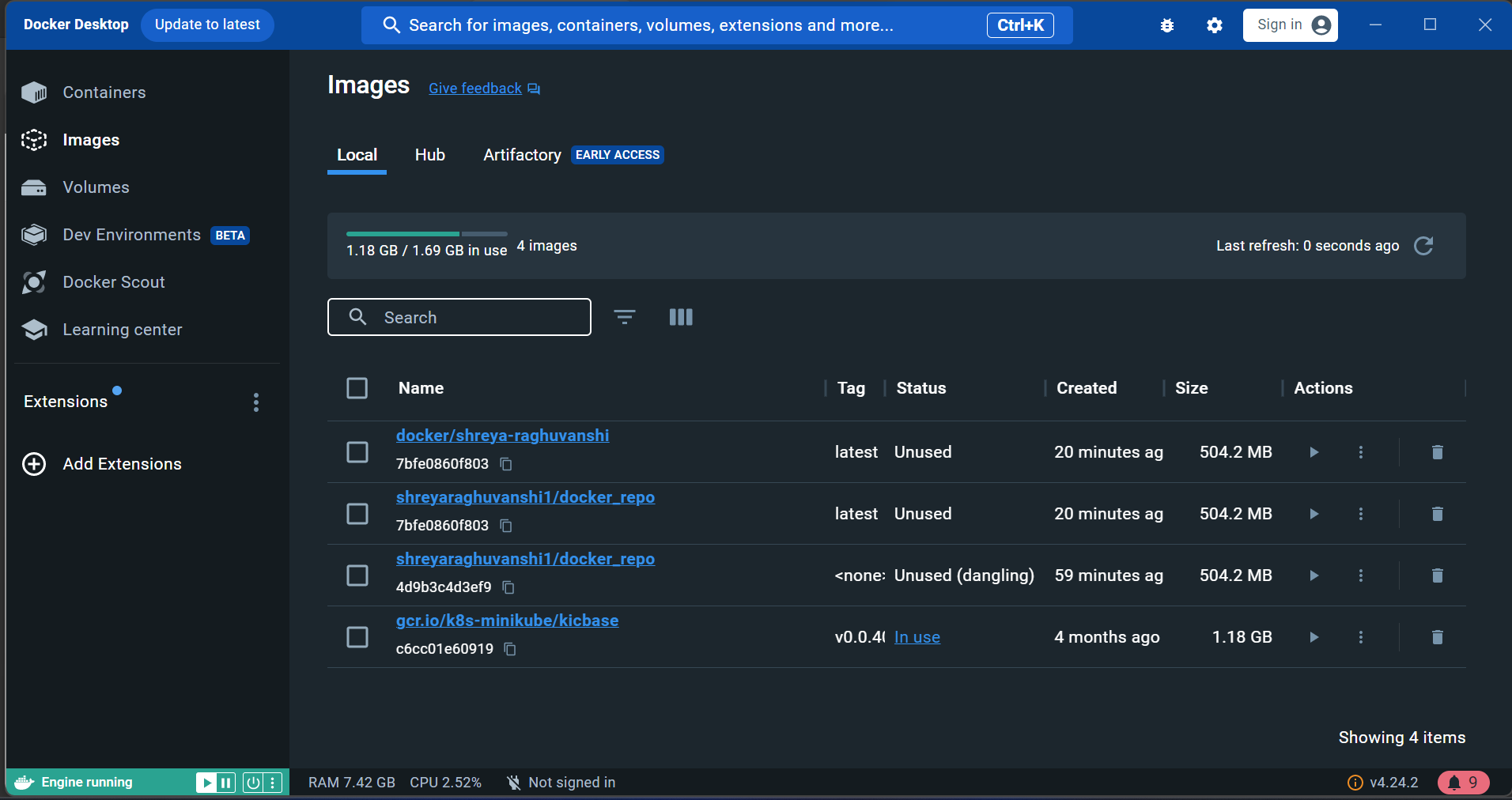
1. Pipeline successful in Jenkins



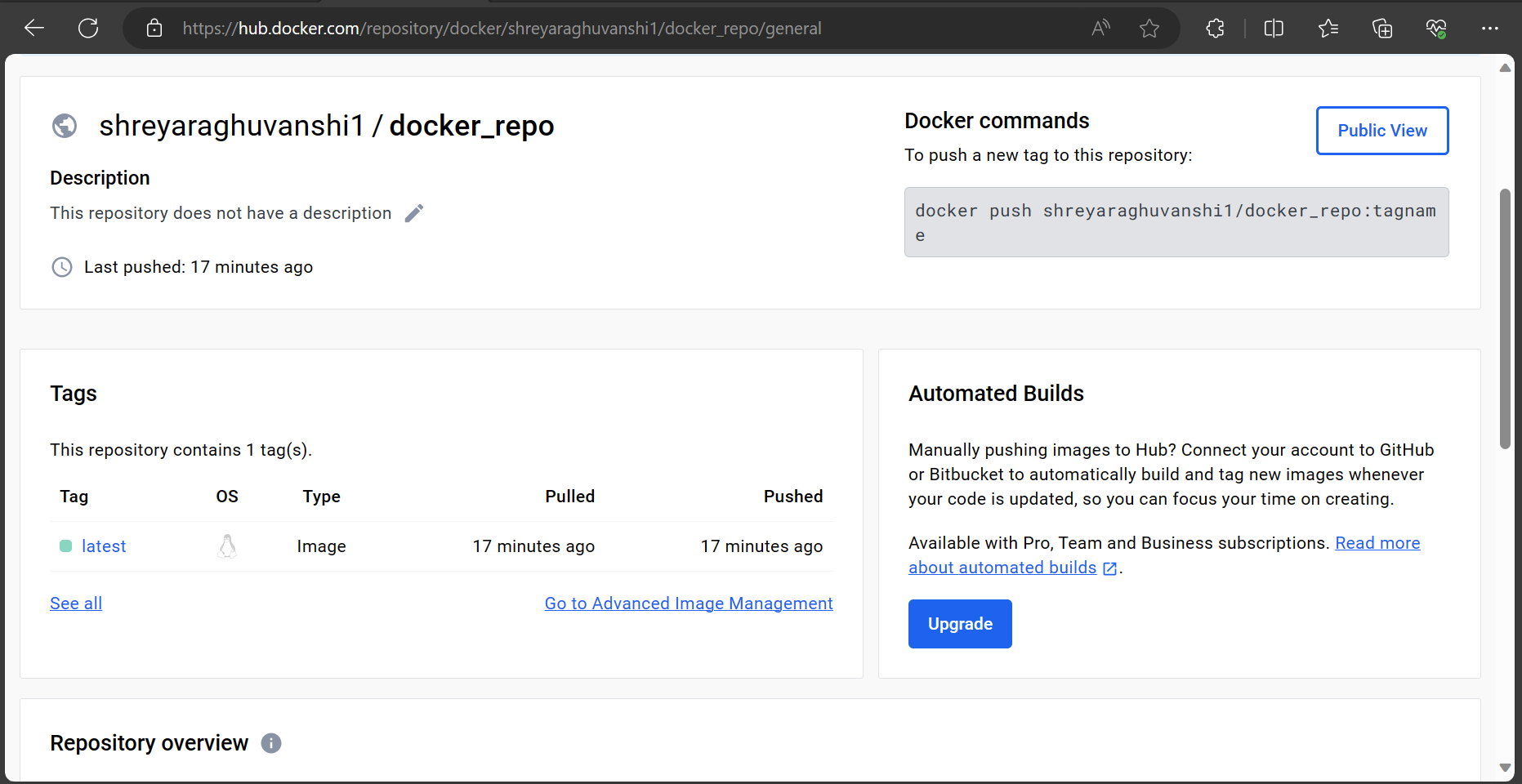
1. Java application pushed on Sonarqube for code analysis:



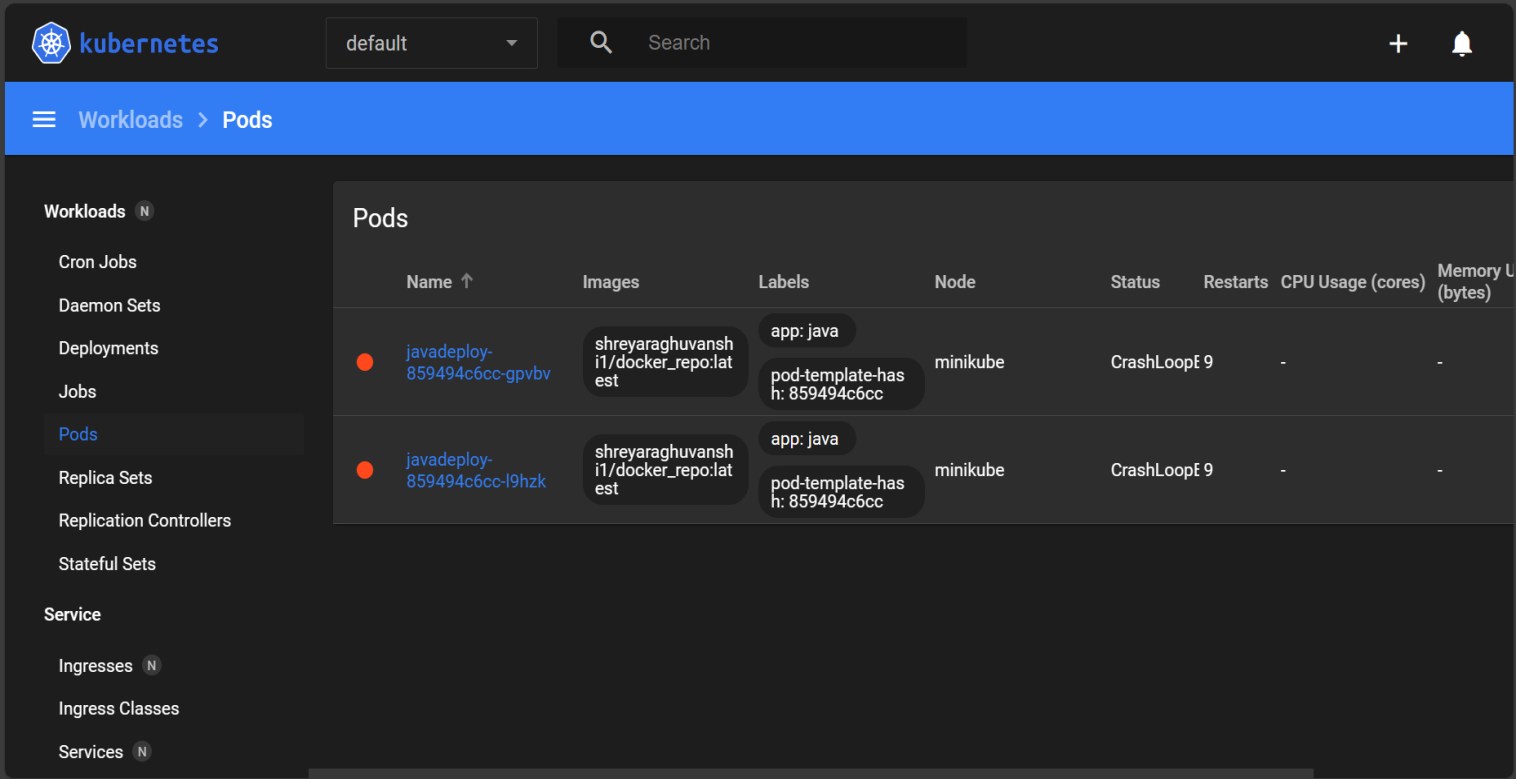
1. Image built in Docker desktop:



1. Image pushed in DockerHub:



1. App deployed on Kubernetes:



**GitHub repository link for reference:**

<https://github.com/ShreyaRaghuvanshi1/L00179092_pipeline.git>

**References:**

Aby, A. (2023). *How to Build a CI/CD Pipeline with Jenkins and Kubernetes*. Hactica. Available at: <https://www.hatica.io/blog/ci-cd-pipeline-with-jenkins-and-kubernetes/> (Accessed 10 November, 2023).

Aleksic, M. (2022). *How to Configure Docker in Jenkins {Step-by-Step Guide}*. Knowledge Base by phoenixNAP. Available at: <https://phoenixnap.com/kb/how-to-configure-docker-in-jenkins> (Accessed 10 November, 2023).

Jagrat (2021). *How to Install Maven plugin and set up Maven project with Jenkins?* TOOLSQA. Available at: <https://www.toolsqa.com/jenkins/jenkins-maven/> (Accessed: 27 October, 2023).

Nandeesh, P. (2023). *Mastering Continuous Integration: Git and Jenkins Integration Best Practices*. Medium. Available at: <https://prajjwalanandeesh.medium.com/mastering-continuous-integration-git-and-jenkins-integration-best-practices-74ba2ead88a4#:~:text=Steps%20to%20Integrate%20Git%20with%20Jenkins%201%20Step> (Accessed: 20 October, 2023).

‌ Shah, V. (2020). *How to Integrate SonarQube with Jenkins*. TatvaSoft Blog. Available at: <https://www.tatvasoft.com/blog/integrate-sonarqube-with-jenkins/> (Accessed 02 November, 2023).

‌