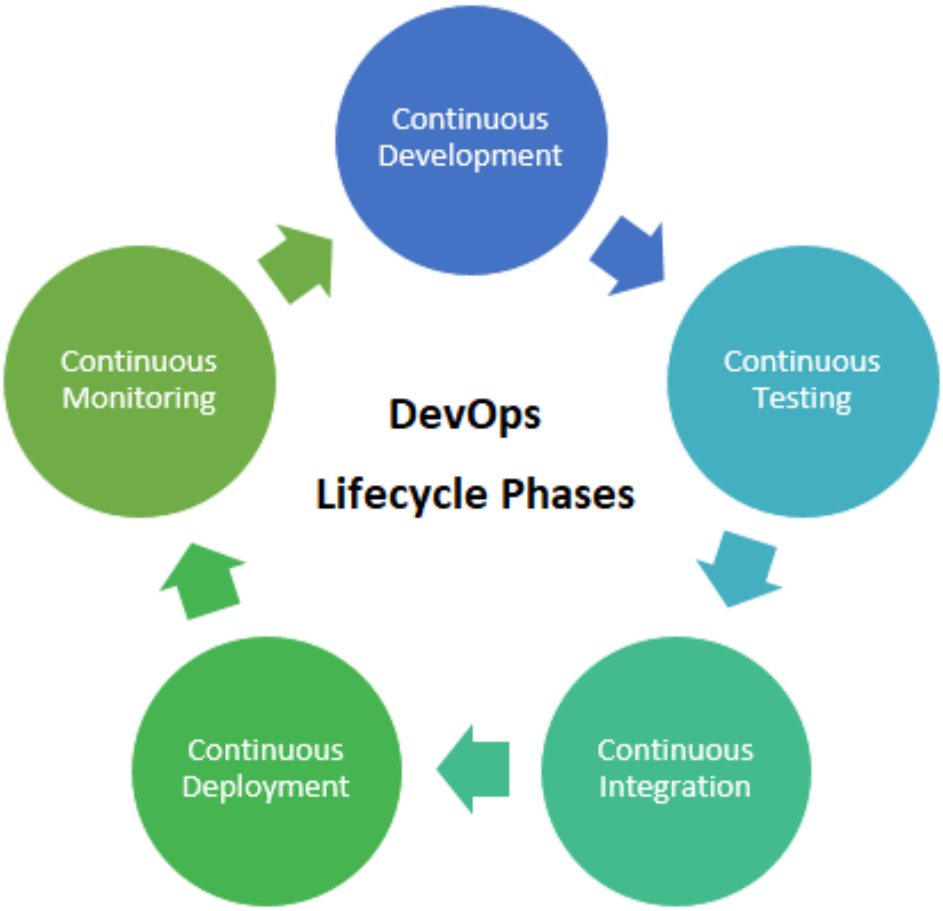
CS 816 / Software Production Engineering

MINI PROJECT

SCIENTIFIC CALCULATOR WITH DEVOPS

-VAIBHAV TANDON "MT2020150"



GitHub Link:

https://github.com/vtandonv/calculatorDevOps (Project Initially Created)

https://github.com/vtandonv/CalculatorUsingDevOps (Project recreated for taking the screenshots)

DockerHub Link:

https://hub.docker.com/repository/docker/vtandonv/calculatorusingdevopslatest

Introduction

DevOps is a set of practices that combines software development and IT operations. It aims to shorten the systems development life cycle and provide continuous delivery with high software quality.

The project is all about building a Java based Scientific Calculator application and integrating it with the DevOps tools (Jenkins, Ansible etc). The calculator will be able to perform exponentiation, natural logarithm, square root, and factorial of numbers given as input.

Talking about the overall build-integrate-deploy process of the application with DevOps it involves the following steps.

1. Git pull
2. Maven Build
3. Docker Build to image
4. Pushing Docker image
5. Ansible pull Docker image

The above 5 steps are performed in stages in a Continuous Integration- Continuous Development pipeline. The tool used for the same is Jenkins.

Setting up the environment

In order to successfully develop and deploy our application with DevOps the following software packages, tools and plugins are required:

1. **IntelliJ IDEA Ultimate Edition v2020.3**: This provides the environment for coding and testing of the application. Integrating it with Git allows us to directly commit and push the application to the Github repository. Download it from here using college student ID

   https://www.jetbrains.com/community/education/#students

2. **Git install and setting up Github:** Git is a version control system which will contain the entire code base in the form of a repository. Any changes to our project can be directly pushed to our Github repository using either Git or Git plugin of IntelliJ. Command for the same is

$ sudo apt install git-all

3. **Setting up Maven on IntelliJ:** Maven is a build tool which resolves various dependencies present in our project like JUnit, log4j etc. It builds and packages the project in the form of a .jar file. Open a new Java application on IntelliJ as a Maven project.
4. **Installing Jenkins and setting up the Jenkins dashboard:** Jenkins is a powerful DevOps tool which streamlines our continuous integration and development in a single pipeline. To install Jenkins use command:

https://www.digitalocean.com/community/tutorials/how-to-install-jenkins-on-ubuntu-18-04

Install various plugins like Git, Github, Docker and Ansible over the Jenkins

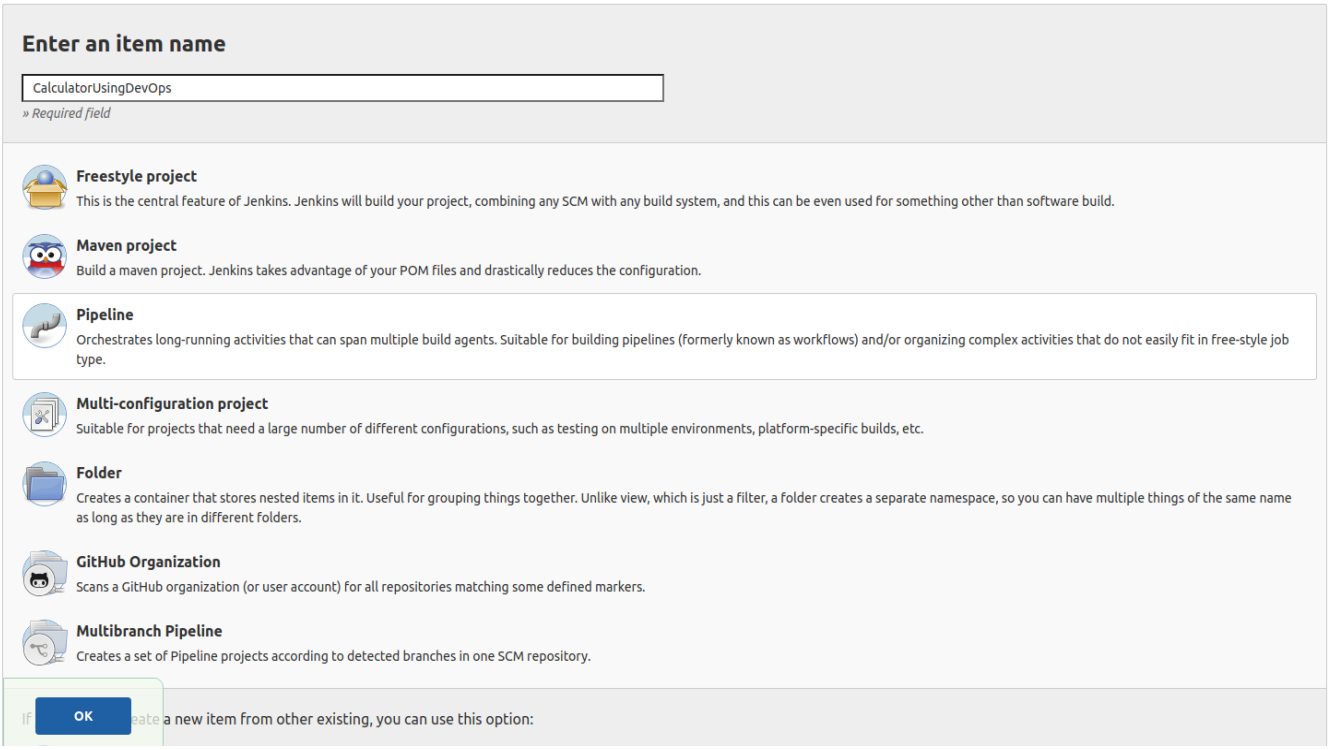


Fig: Logging into Jenkins in the localhost 8080 port and creating a Pipeline

5. **Installing Docker and making an account on Docker Hub**: Docker is a set of platform as a service products that use OS-level virtualization to deliver software in packages called containers. Containers are isolated from one another and bundle their own software, libraries and configuration files; they can communicate with each other through well-defined channels. Install it using the command below:

   https://docs.docker.com/engine/install/ubuntu/

6. **Setting up Ansible:** Ansible is a configuration management tool which is used to deploy the docker image and run the .jar file inside other host specified in Ansible inventory. Refer the below link for installation and setup.

https://docs.ansible.com/ansible/latest/installation_guide/intro_installation.html

Understanding the Jenkins pipeline stages

Jenkins is primarily used for streamlining the entire workflow. It has various plugins for each of the stages (like Git, Maven, Docker, Ansible etc) so that each and every stage can be continuously integrated and developed under one hood. By the word continuous we mean that as soon as the developer makes a single Git push to his repository the Jenkins will automatically perform all the steps which come under CI-CD scheme.

Stage 1: Git Pull

This stage involves creating a public GitHub repository and adding a ReadMe for the project.

Git is a free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency

Basic Git Commands

Initialize the local directory as a Git repository.

$ git init

Add the files in your new local repository. This stages them for the first commit.

$ git add .

Adds the files in the local repository and stages them for commit. To unstage a file, use 'git reset HEAD YOUR-FILE'.

Commit the files that you've staged in your local repository.

$ git commit -m "First commit"

Commits the tracked changes and prepares them to be pushed to a remote repository. To remove this commit and modify the file, use 'git reset --soft HEAD~1' and commit and add the file again.

In Terminal, add the URL for the remote repository where your local repository will be pushed.

$ git remote add origin <REMOTE_URL>

Sets the new remote

$ git remote -v

Verifies the new remote URL

Push the changes in your local repository to GitHub.

$ git push origin main

Pushes the changes in your local repository up to the remote repository you specified as the origin

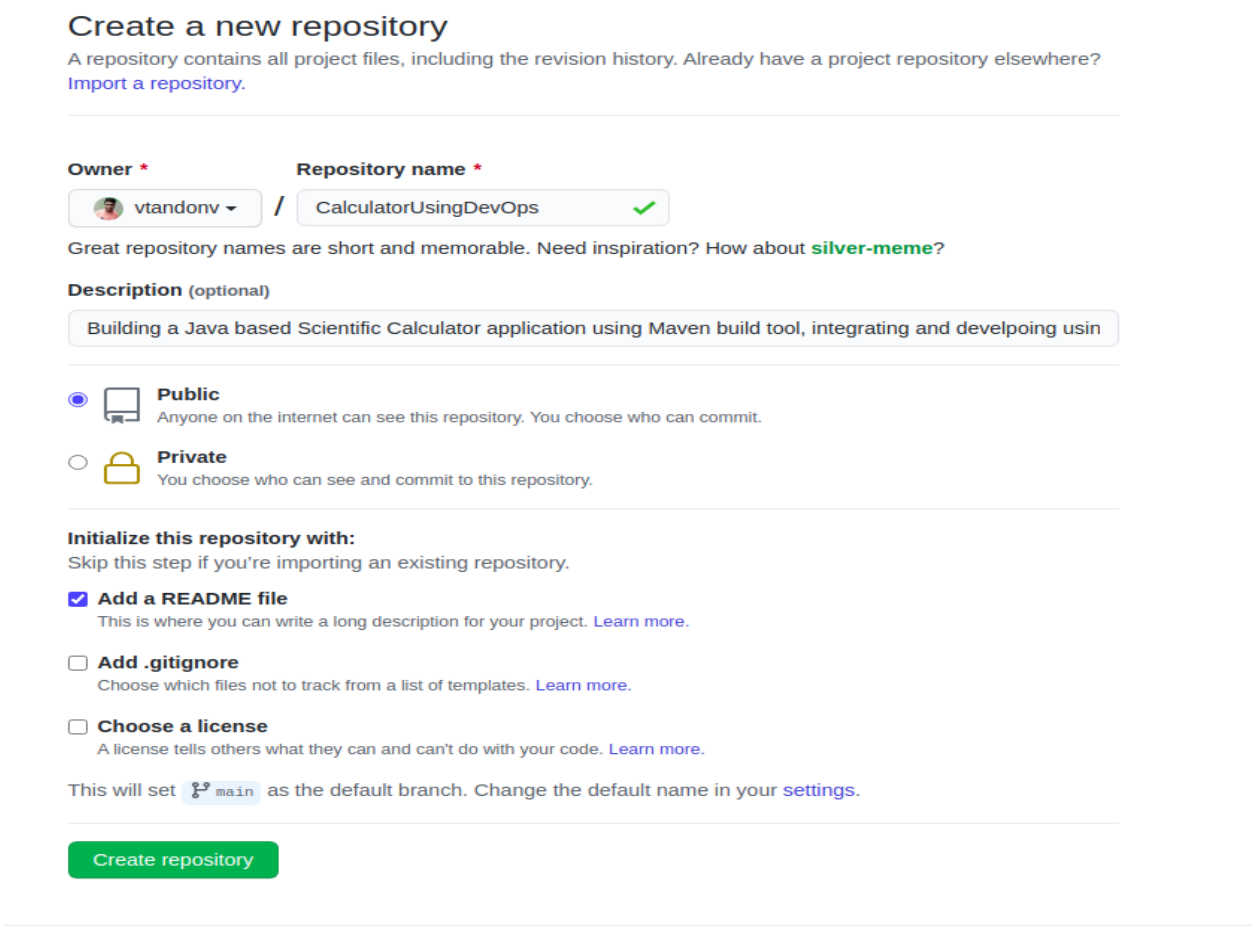


Fig 1.1: Creating a public Repository

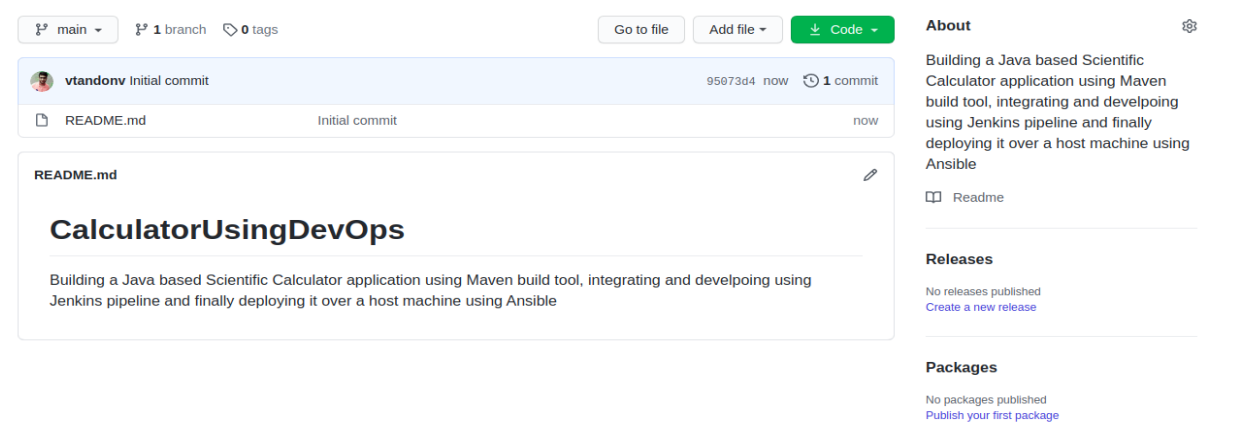


Fig 1.2: Adding ReadMe

Pipeline code for Git Pull:

```
stage('Git pull') {
        steps {
            git branch: 'main', url: 'https://github.com/vtandonv/CalculatorUsingDevOps'
        } }
```

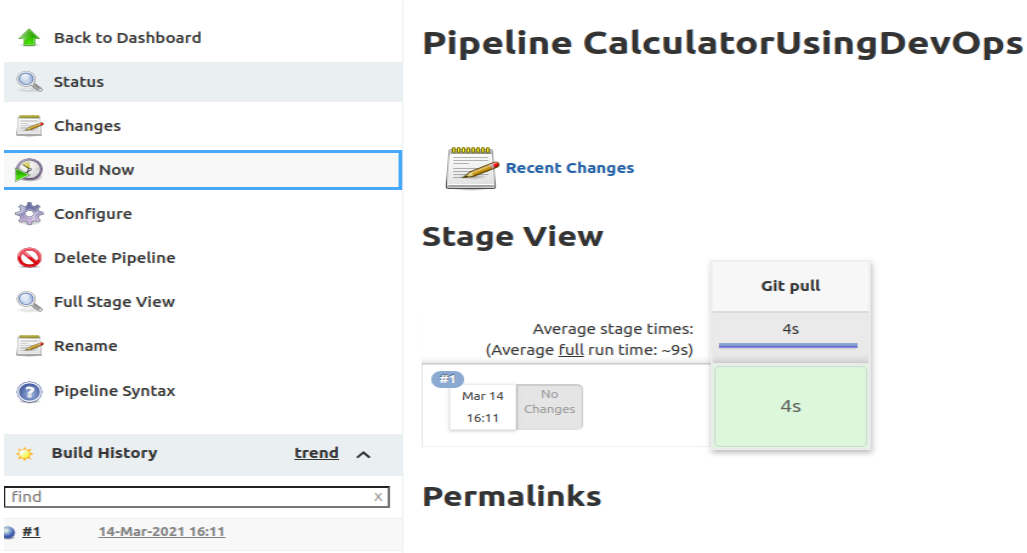


Fig 1.3: Successfully built Stage 1 of Pipeline

Stage 2: Maven Build

In this the entire project is being compiled, validated and tested by Maven and Junit. All this is handled by the Jenkins pipeline which outputs a target folder containing the .jar file.

Maven is a build automation tool used primarily for Java projects. Maven can also be used to build and manage projects written in C#, Ruby, Scala, and other languages. The Maven project is hosted by the Apache Software Foundation.

JUnit is a unit testing framework for the Java programming language. JUnit has been important in the development of test-driven development. JUnit is linked as a JAR at compile-time

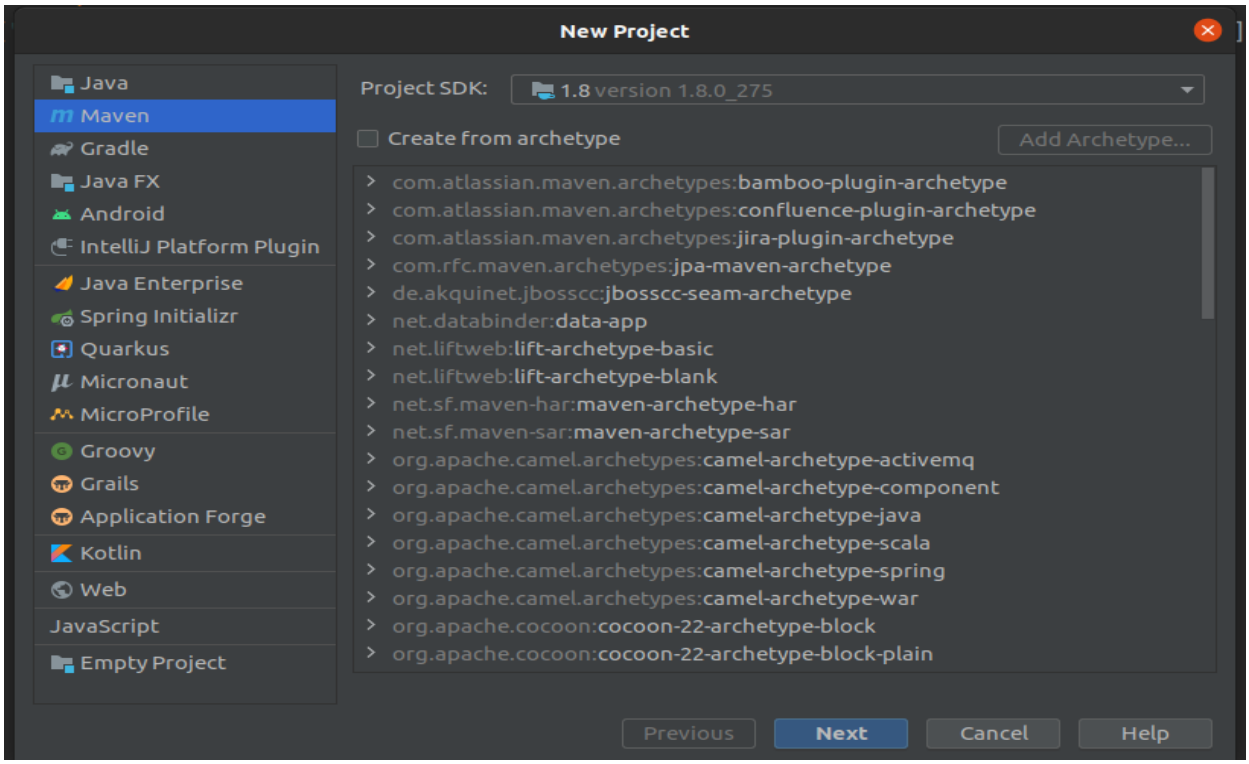


Fig 2.1: Creating a New Maven Project on IntelliJ

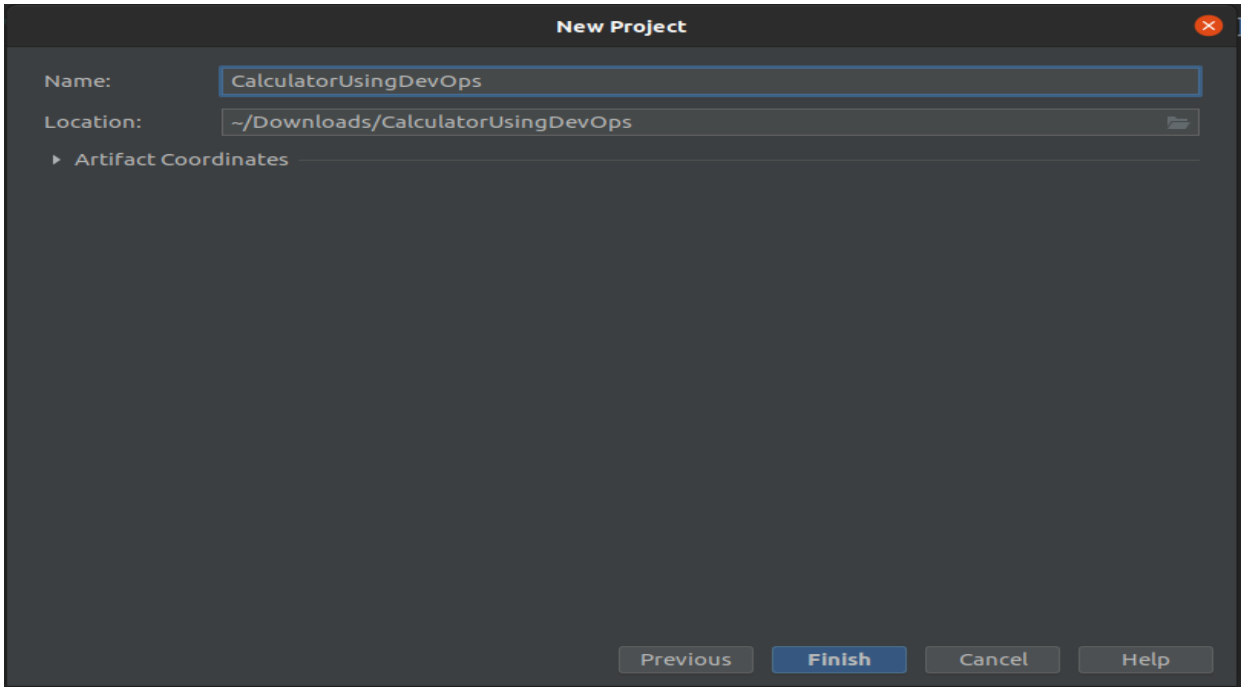


Fig 2.2: Naming the project

```xml
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>org.example</groupId>
    <artifactId>calculatorDevOps</artifactId>
    <version>1.0-SNAPSHOT</version>
    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <executions>
                    <execution>
                        <phase>package</phase>
                        <goals>
                            <goal>single</goal>
                        </goals>
                        <configuration>
                            <archive>
                                <manifest>
                                    <mainClass>calculator.Calculator</mainClass>
                                </manifest>
                            </archive>
                            <descriptorRefs>
                                <descriptorRef>jar-with-dependencies</descriptorRef>
                            </descriptorRefs>
```

Fig 2.3: Adding Dependencies to pom.xml

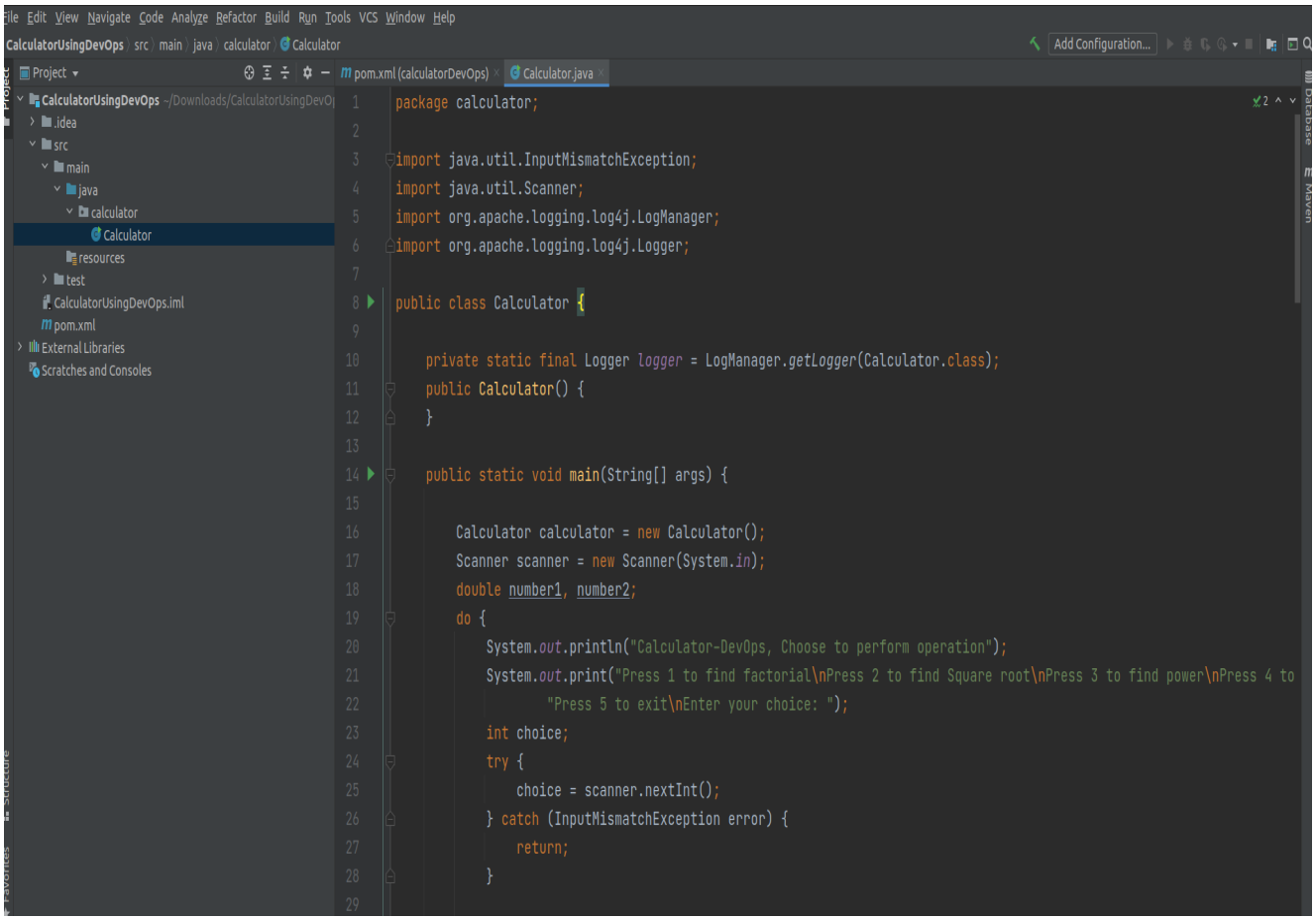


Fig 2.4: Writing the code for Scientific Calculator in calculator/Calculator.java

```java
import calculator.Calculator;
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {
    private static final double DELTA = 1e-15;
    Calculator calculator = new Calculator();

    @Test
    public void factorialTruePositive(){
        assertEquals( message: "Finding factorial of a number for True Positive",  expected: 120, calculator.fact( num: 5), DEL
        assertEquals( message: "Finding factorial of a number for True Positive",  expected: 24, calculator.fact( num: 4), DELT
    }

    @Test
    public void factorialFalsePositive(){
        assertNotEquals( message: "Finding factorial of a number for False Positive",  unexpected: 120, calculator.fact( num: 6
        assertNotEquals( message: "Finding factorial of a number for False Positive",  unexpected: 24, calculator.fact( num: 3)
    }

    @Test
    public void powerTruePositive(){
        assertEquals( message: "Finding power for True Positive",  expected: 8, calculator.power(2, 3), DELTA);
        assertEquals( message: "Finding power for True Positive",  expected: 64, calculator.power(4, 3), DELTA);
    }

    @Test
    public void powerFalsePositive(){
        assertNotEquals( message: "Finding power for False Positive",  unexpected: 6, calculator.power(2, 2), DELTA);
```

Fig 2.5: Writing Test Cases in CalculatorTest.java

```xml
<?xml version="1.0" encoding="UTF-8"?>
<Configuration status="INFO">
    <Appenders>
        <Console name="ConsoleAppender" target="SYSTEM_OUT">
            <PatternLayout pattern="%d{HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n" />
        </Console>
        <File name="FileAppender" fileName="calculator.log" immediateFlush="false" append="true">
            <PatternLayout pattern="%d{yyyy-MM-dd HH:mm:ss.SSS} [%t] %-5level %logger{36} - %msg%n"/>
        </File>
    </Appenders>
    <Loggers>
        <Root level="debug">
            <AppenderRef ref="ConsoleAppender" />
            <AppenderRef ref="FileAppender"/>
        </Root>
    </Loggers>
</Configuration>
```

Fig 2.6: Making logger configuration file as log4j2.xml

```
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  6.595 s
[INFO] Finished at: 2021-03-14T15:40:03+05:30
[INFO] ------------------------------------------------------------------------
```

Fig 2.7: Building using command *mvn clean package*

```
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ cd target
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps/target$ ls
archive-tmp                      calculatorDevOps-1.0-SNAPSHOT-jar-with-dependencies.jar  generated-sources       maven-archiver  surefire-reports
calculatorDevOps-1.0-SNAPSHOT.jar  classes                                                 generated-test-sources  maven-status    test-classes
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps/target$ java -jar calculatorDevOps-1.0-SNAPSHOT-jar-with-dependencies.jar
Calculator-DevOps, Choose to perform operation
Press 1 to find factorial
Press 2 to find Square root
Press 3 to find power
Press 4 to find natural logarithm
Press 5 to exit
Enter your choice: 1
Enter a number : 3
15:42:26.115 [main] INFO  calculator.Calculator - [FACTORIAL] - 3.0
15:42:26.125 [main] INFO  calculator.Calculator - [RESULT - FACTORIAL] - 6.0
Factorial of 3.0 is : 6.0
```

Fig 2.8: Running the .jar file (with dependencies)

```
Terminal: Local ×  +
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ git init
Initialised empty Git repository in /home/vaibhav/Downloads/CalculatorUsingDevOps/.git/
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ git add .
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ git commit -m "First commit"
[master (root-commit) f0007bf] First commit
 24 files changed, 468 insertions(+)
 create mode 100644 .idea/.gitignore
 create mode 100644 .idea/compiler.xml
 create mode 100644 .idea/jarRepositories.xml
 create mode 100644 .idea/misc.xml
 create mode 100644 .idea/vcs.xml
 create mode 100644 CalculatorUsingDevOps.iml
 create mode 100644 calculator.log
 create mode 100644 pom.xml
 create mode 100644 src/main/java/calculator/Calculator.java
 create mode 100644 src/main/resources/log4j2.xml
 create mode 100644 src/test/java/CalculatorTest.java
 create mode 100644 target/calculator.log
 create mode 100644 target/calculatorDevOps-1.0-SNAPSHOT-jar-with-dependencies.jar
 create mode 100644 target/calculatorDevOps-1.0-SNAPSHOT.jar
 create mode 100644 target/classes/calculator/Calculator.class
 create mode 100644 target/classes/log4j2.xml
 create mode 100644 target/maven-archiver/pom.properties
 create mode 100644 target/maven-status/maven-compiler-plugin/compile/default-compile/createdFiles.lst
 create mode 100644 target/maven-status/maven-compiler-plugin/compile/default-compile/inputFiles.lst
 create mode 100644 target/maven-status/maven-compiler-plugin/testCompile/default-testCompile/createdFiles.lst
 create mode 100644 target/maven-status/maven-compiler-plugin/testCompile/default-testCompile/inputFiles.lst
 create mode 100644 target/surefire-reports/CalculatorTest.txt
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ git remote add origin  https://github.com/vtandonv/CalculatorUsingDevOps.git
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ git remote -v
origin  https://github.com/vtandonv/CalculatorUsingDevOps.git (fetch)
origin  https://github.com/vtandonv/CalculatorUsingDevOps.git (push)
vaibhav@vaibhav:~/Downloads/CalculatorUsingDevOps$ git push origin --force HEAD:main
Username for 'https://github.com': vtandonv
Password for 'https://vtandonv@github.com':
Enumerating objects: 45, done.
Counting objects: 100% (45/45), done.
Delta compression using up to 4 threads
Compressing objects: 100% (32/32), done.
Writing objects: 100% (45/45), 1.76 MiB | 1.14 MiB/s, done.
Total 45 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/vtandonv/CalculatorUsingDevOps.git
 + 95073d4...f0007bf HEAD -> main (forced update)
```

Fig 2.9: Committing and pushing changes to GitHub using Git commands

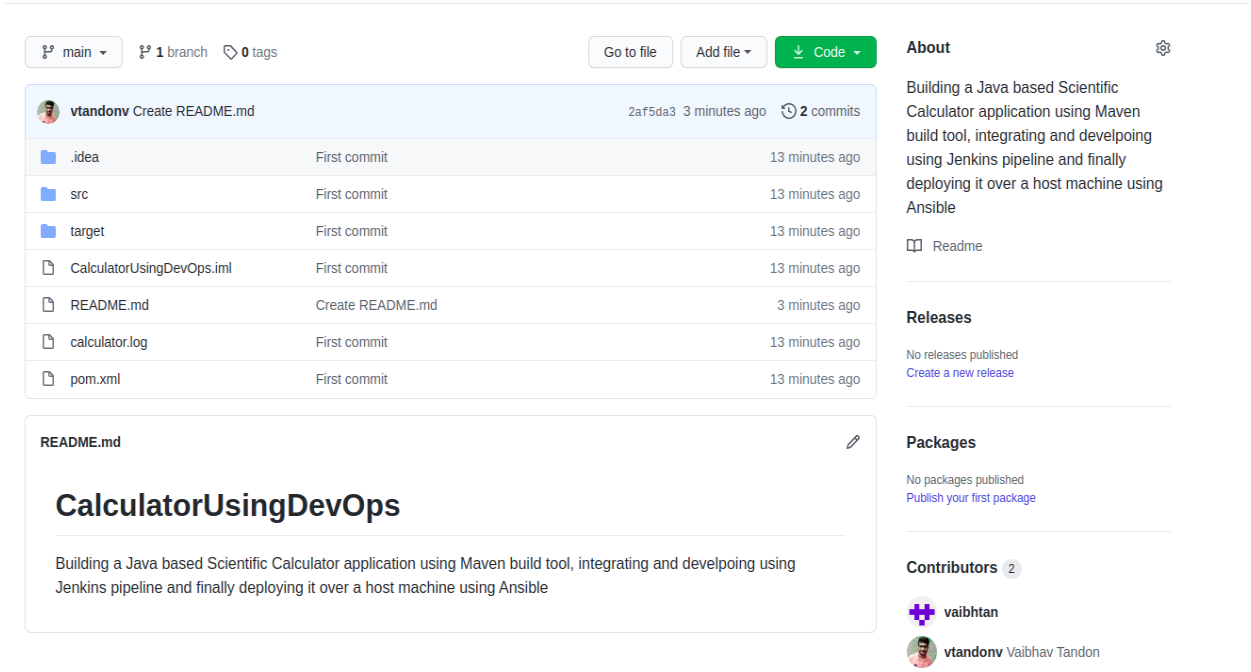


Fig 2.10: Changes visible on GitHub

Pipeline script for Maven Build:

```
stage('Maven Build') {
        steps {
            script {
                sh 'mvn clean install'
            }
        }
    }
```

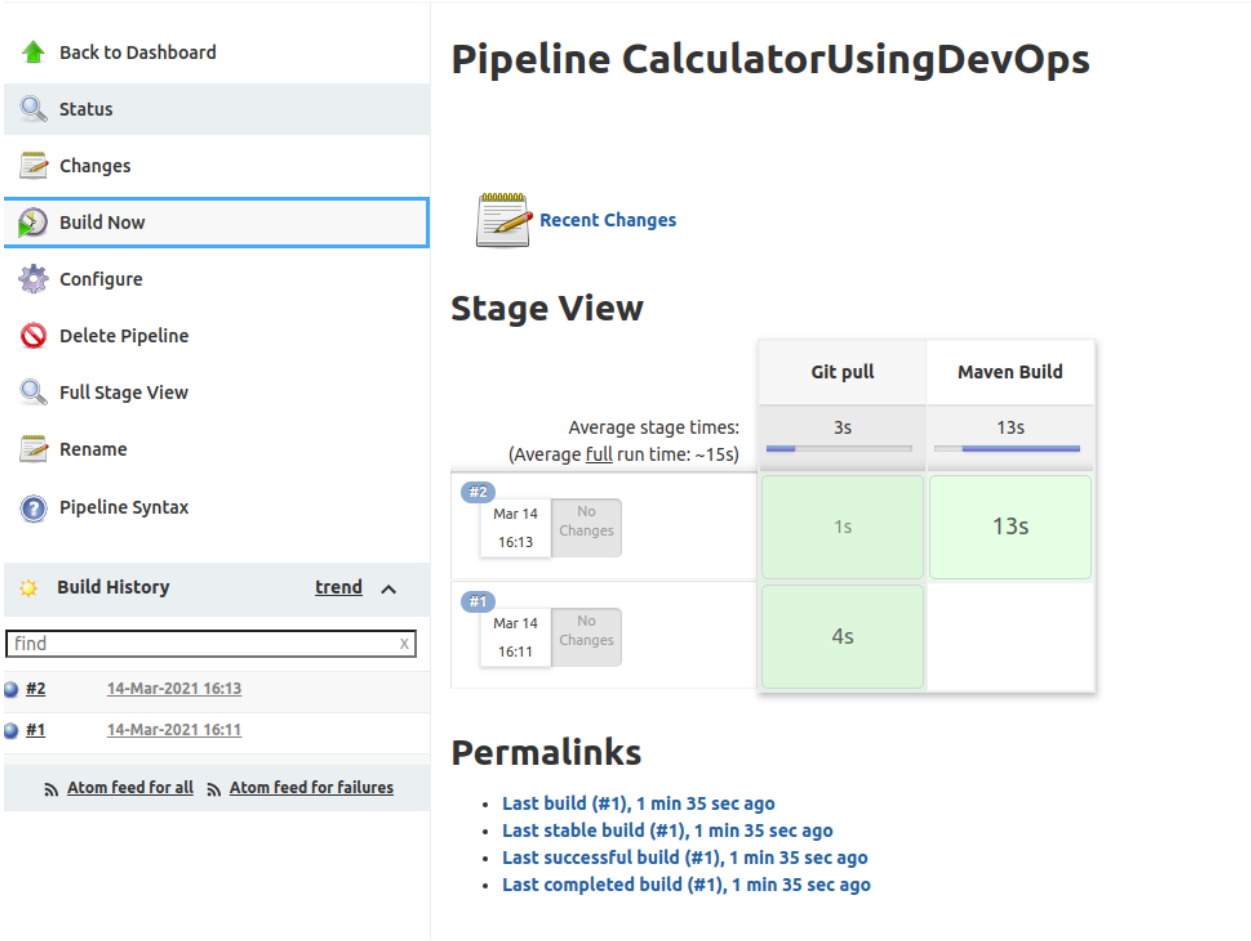


Fig 2.11: Successfully built Stage 2 of Pipeline

Stage 3: Docker Build to image

This step involves containerizing .jar file in a Docker image. For this we first create a Dockerfile in our project and add the name of our .jar file accordingly i.e.

FROM openjdk:8

COPY ./target/calculatorDevOps-1.0-SNAPSHOT-jar-with-dependencies.jar ./

WORKDIR ./

CMD ["java", "-jar", "calculatorDevOps-1.0-SNAPSHOT-jar-with-dependencies.jar"]

We then push this file to our Github account

Pipeline script for this stage is:

```
stage('Docker Build to Image') {
        steps {
            script {
                imageName = docker.build 'vtandonv/calculatorusingdevopslatest'
            }
        }
    }
```

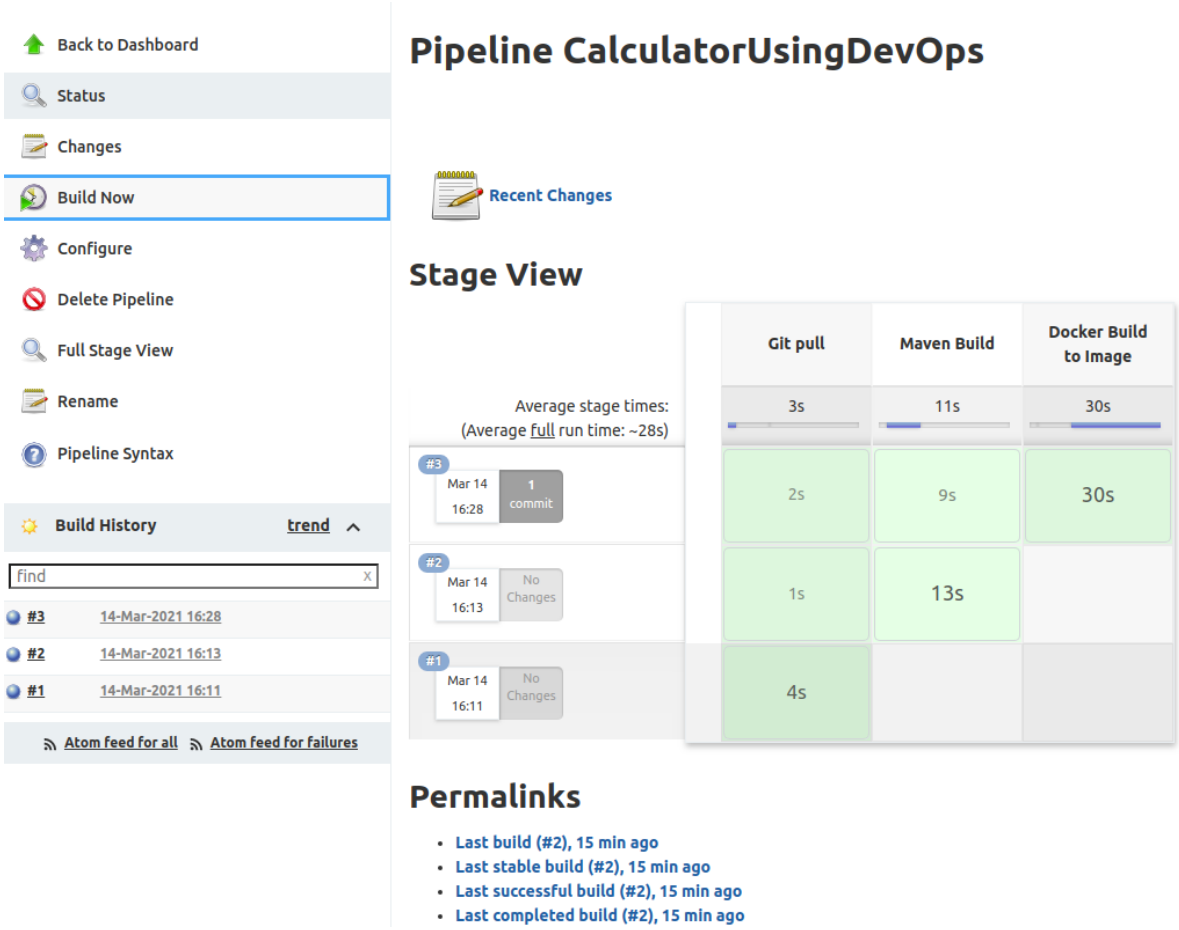


Fig 3.1: Successfully containerized .jar file to Docker Image

Stage 4: Pushing Docker image

In this step we push the image to the Docker hub.

Pipeline Script for this stage is:

```
stage('Push Docker Image') {
        steps {
          script {
            docker.withRegistry('', 'docker-jenkins'){
              imageName.push()
            }
          }
        } }
```

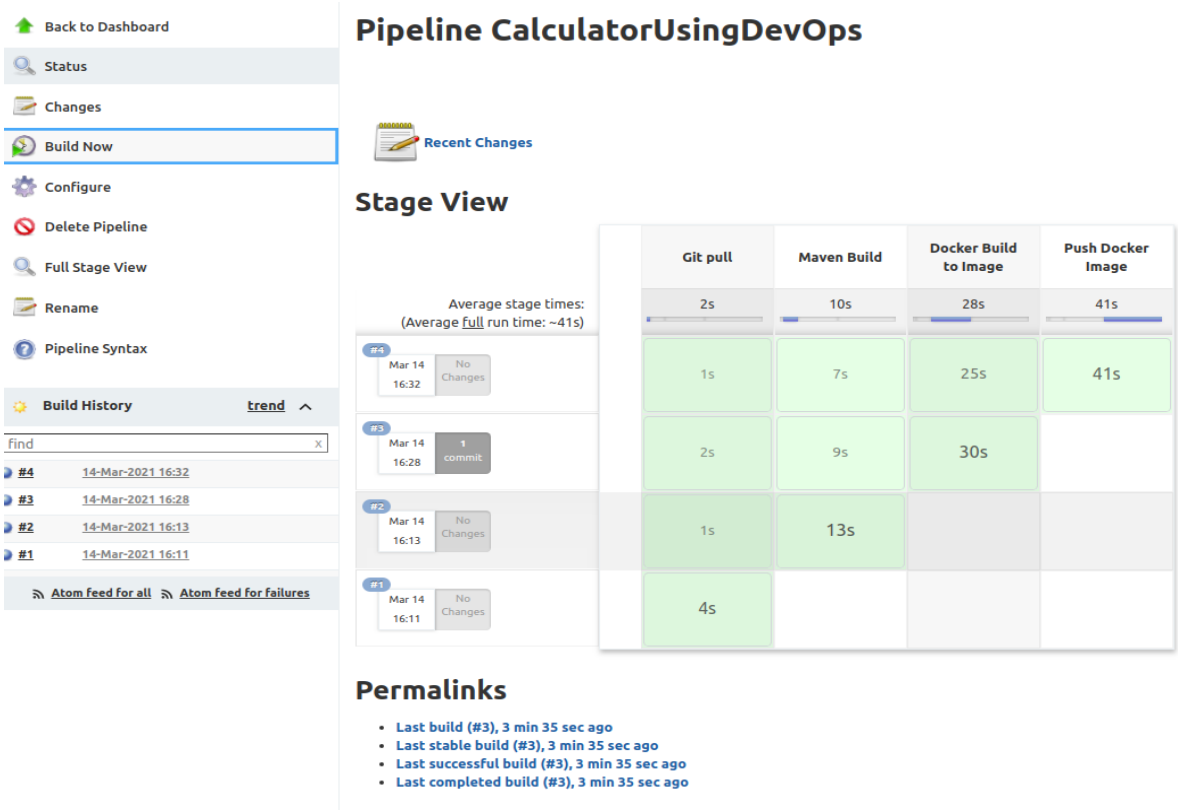


Fig 4.1: Pushed Docker image to Docker Hub

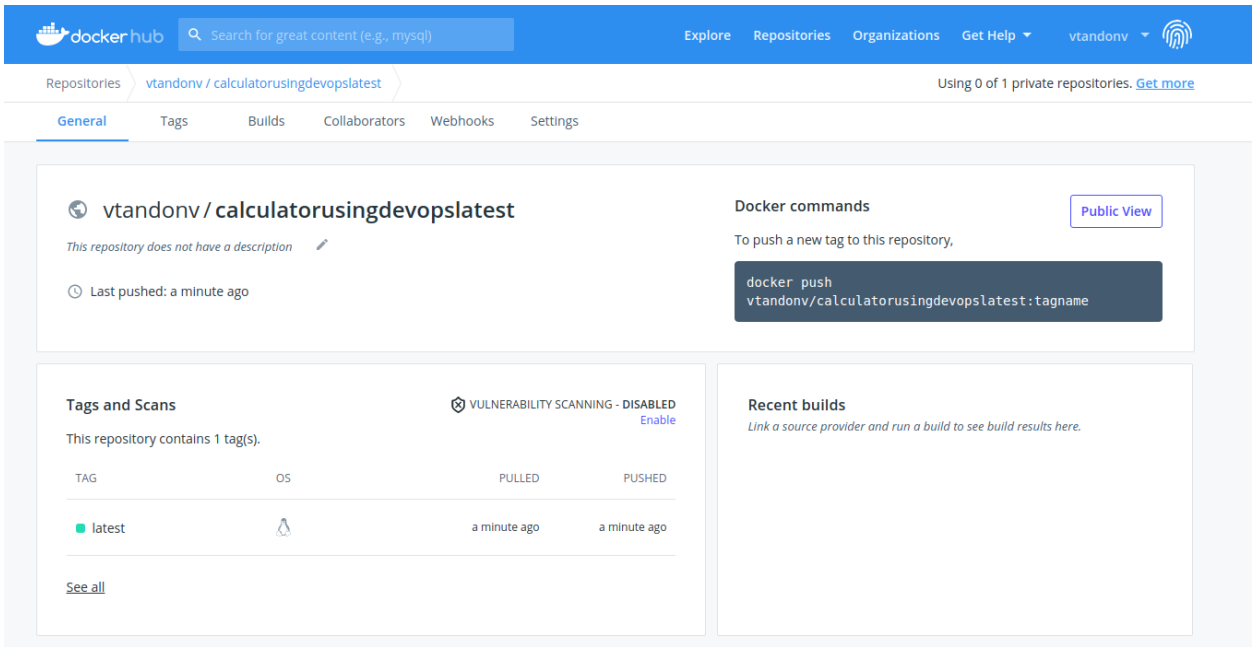


Fig 4.2: Changes reflected to Docker Hub Repository

Stage 5: Ansible pull Docker image

In this step we install Ansible on the controller node and deploy our docker image in all our hosts. In this step we create a directory named deploy-docker in which we first create a file named calc-deploy.yml which contains the contains the instructions required to deploy the docker image to our hosts.

calc-deploy.yml

```
---
- name: Pull docker image of Calculator
  hosts: all
  tasks:
    - name: Pull image
      docker_image:
        name: vtandonv/calculatorusingdevopslatest
        source: pull
```

After that in the same directory we add our inventory file which contains the details of our hosts i.e. their name and IP address.

inventory

[ubuntu]

172.16.129.220 ansible_user=vtandonv

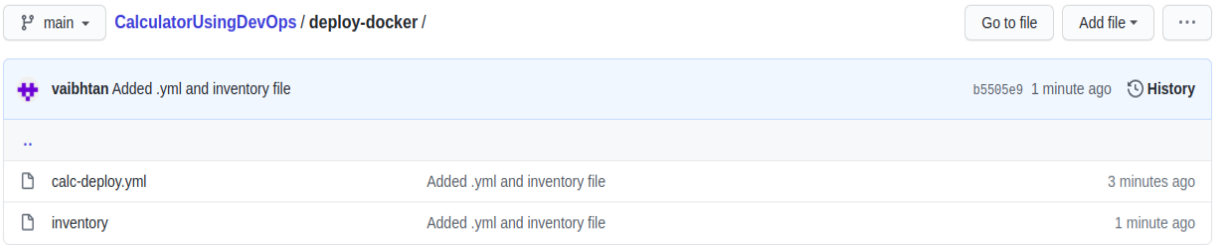


Fig 5.1: Added .yml and inventory file to the Github account

Now we add our Jenkins pipeline syntax for Ansible i.e.

stage('Ansible pull docker image') {

steps { ansiblePlaybook becomeUser: null, colorized: true, disableHostKeyChecking: true, installation: 'Ansible', inventory: 'deploy-docker/inventory', playbook: 'deploy-docker/calc-deploy.yml', sudoUser: null

}

}

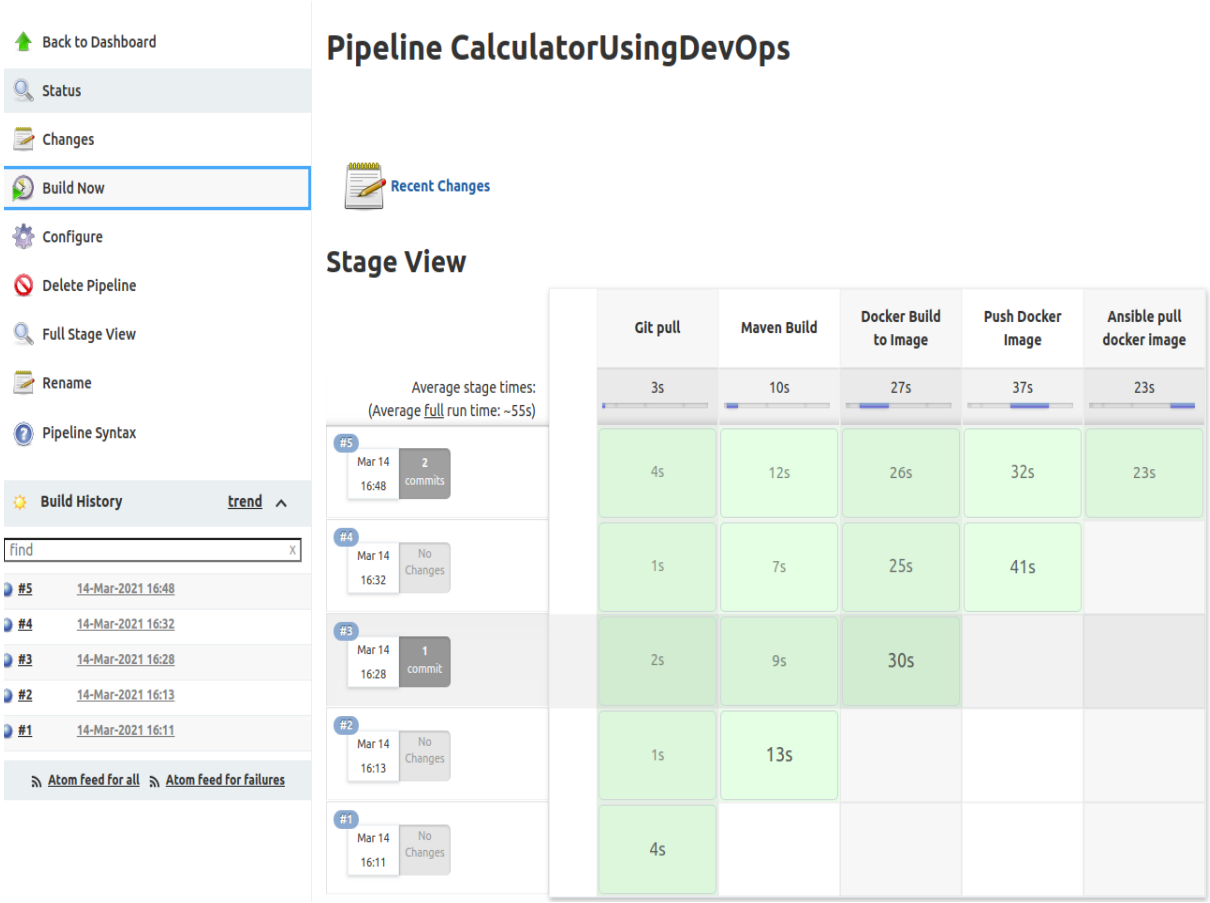


Fig 5.2: Docker image successfully deployed to our host

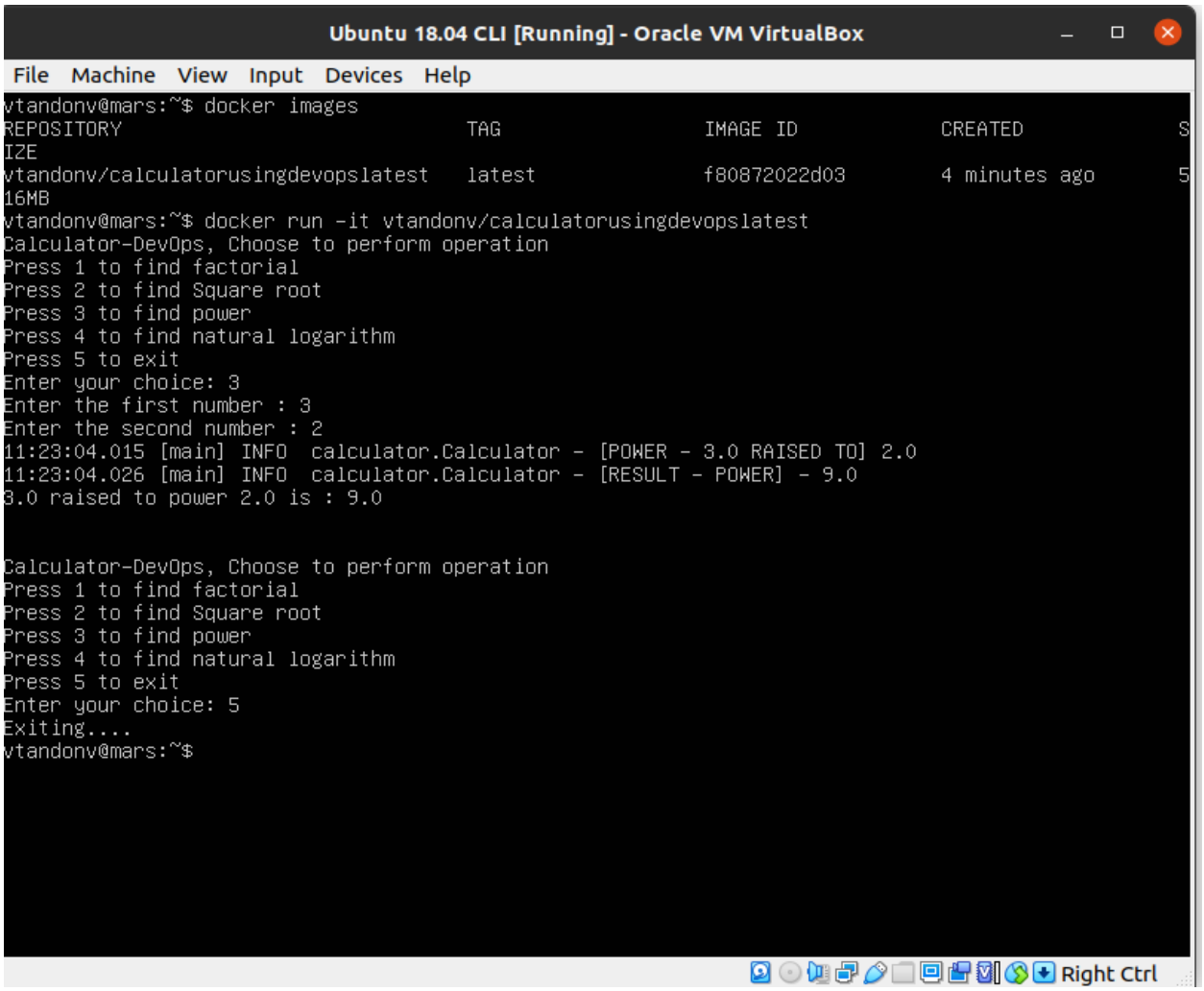


Fig 5.3: .jar file running on Docker image pulled from the Docker hub on our host

Continuous Monitoring using ELK (ElasticSearch - Logstash - Kibana)

"ELK" is the acronym for three open source projects: Elasticsearch, Logstash, and Kibana .

● Elasticsearch is a search and analytics engine.

● Logstash is a server-side data processing pipeline that ingests data from multiple sources simultaneously, transforms it, and then sends it to a "stash" like Elasticsearch.

● Kibana lets users visualize data with charts and graphs in Elasticsearch.

Sign Up for 15 days trial on Elastic Cloud and then create your first deployment

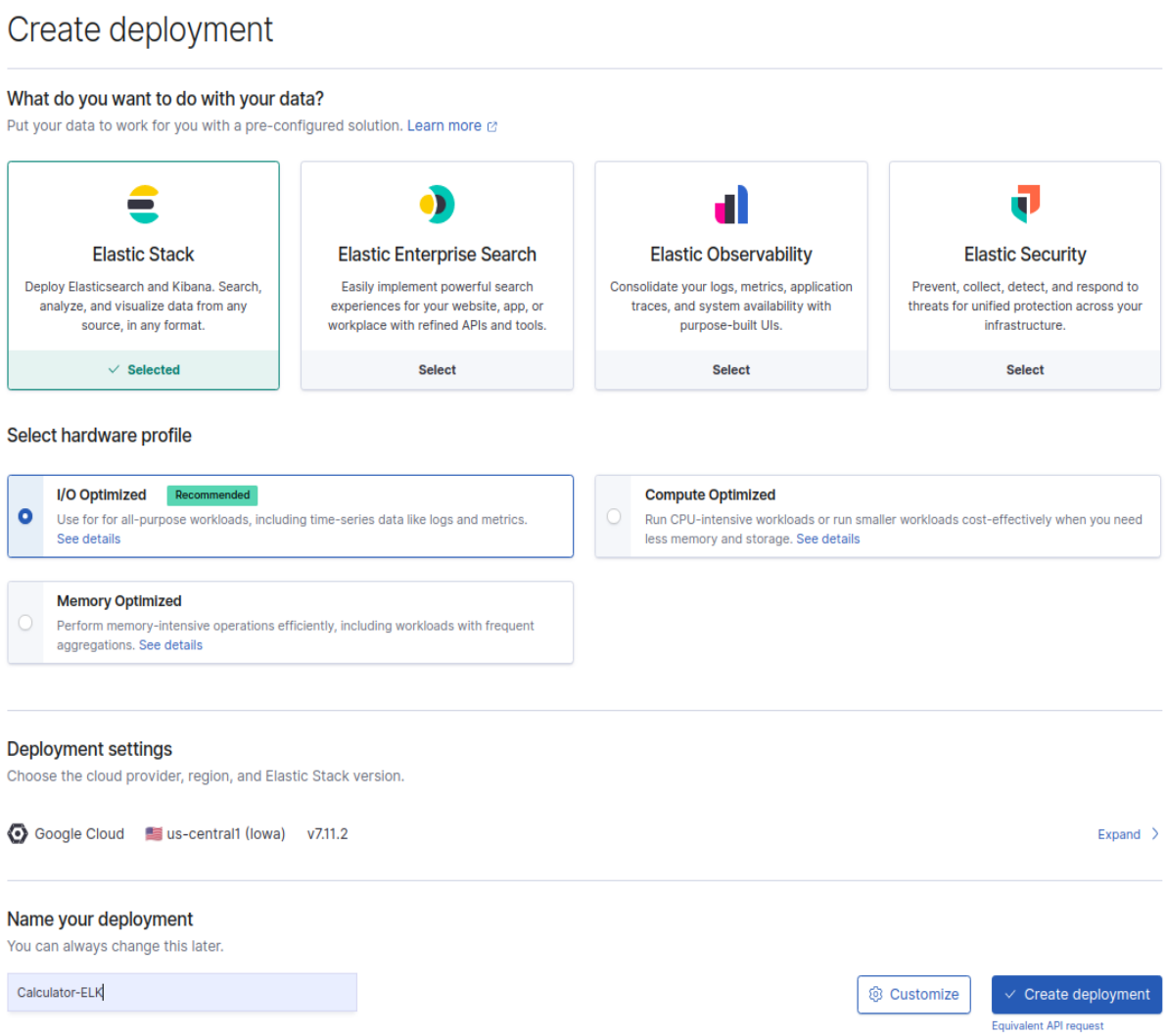


Fig: Creating deployment

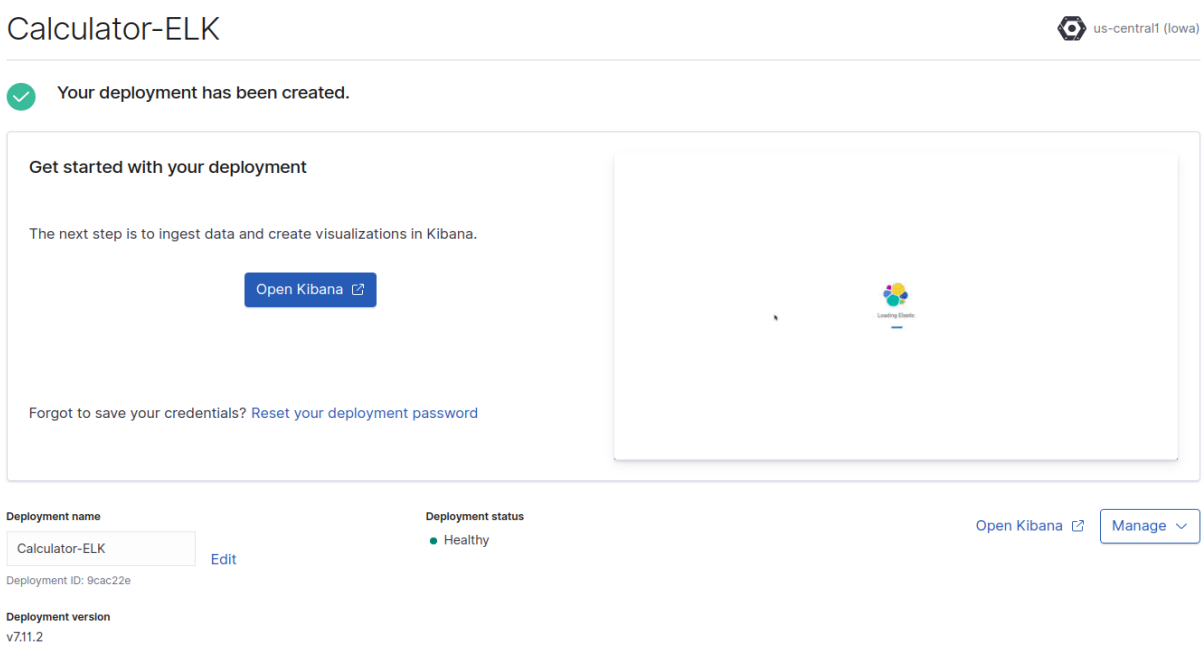


Fig: Deployment Created

Click on Upload a file

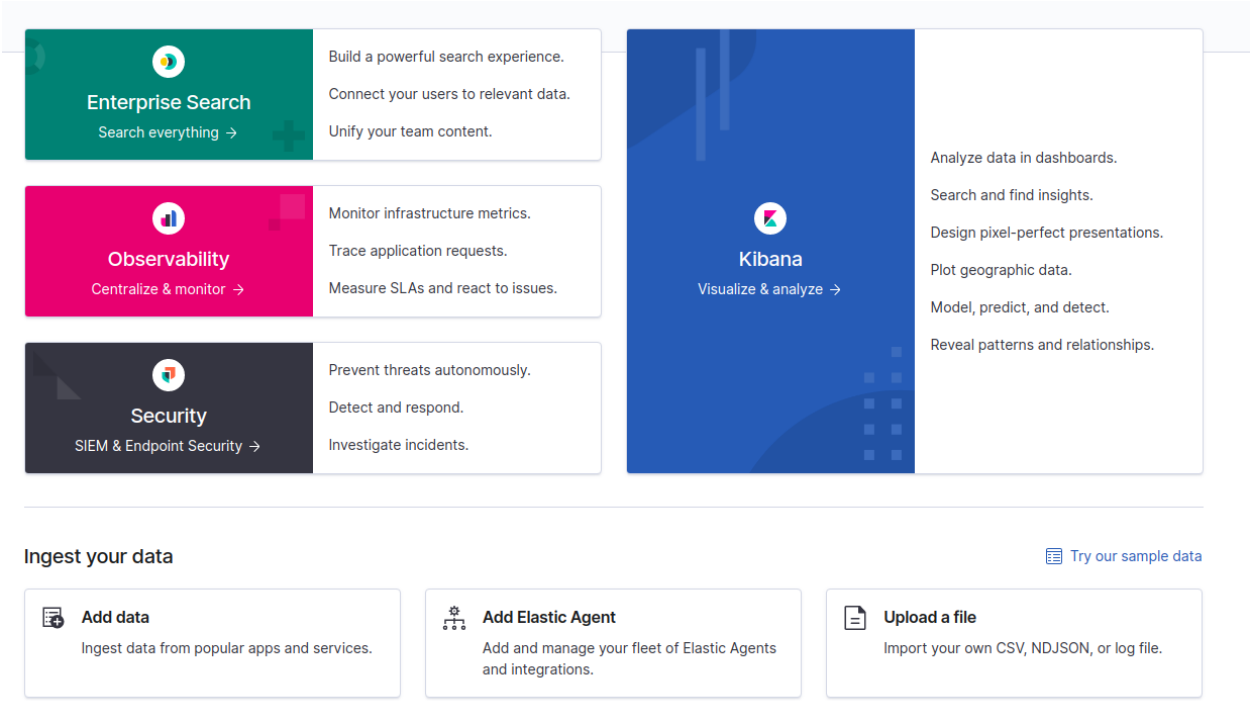


Fig: Upload a the calculator.log file from target folder

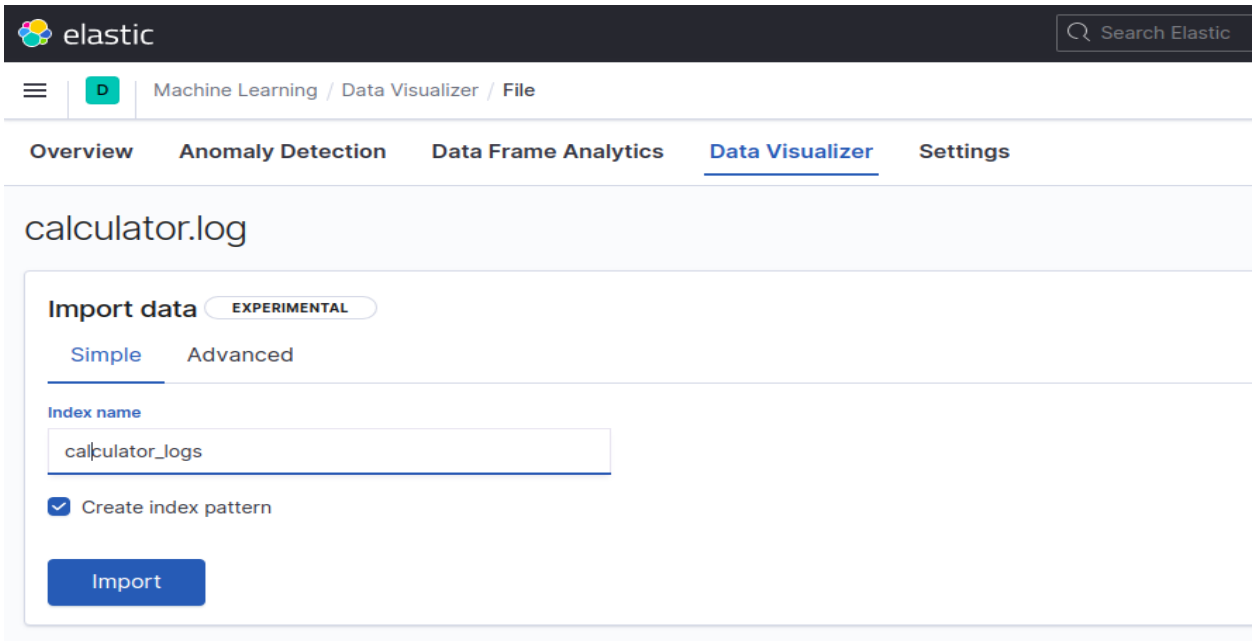


Fig: Assign an index name

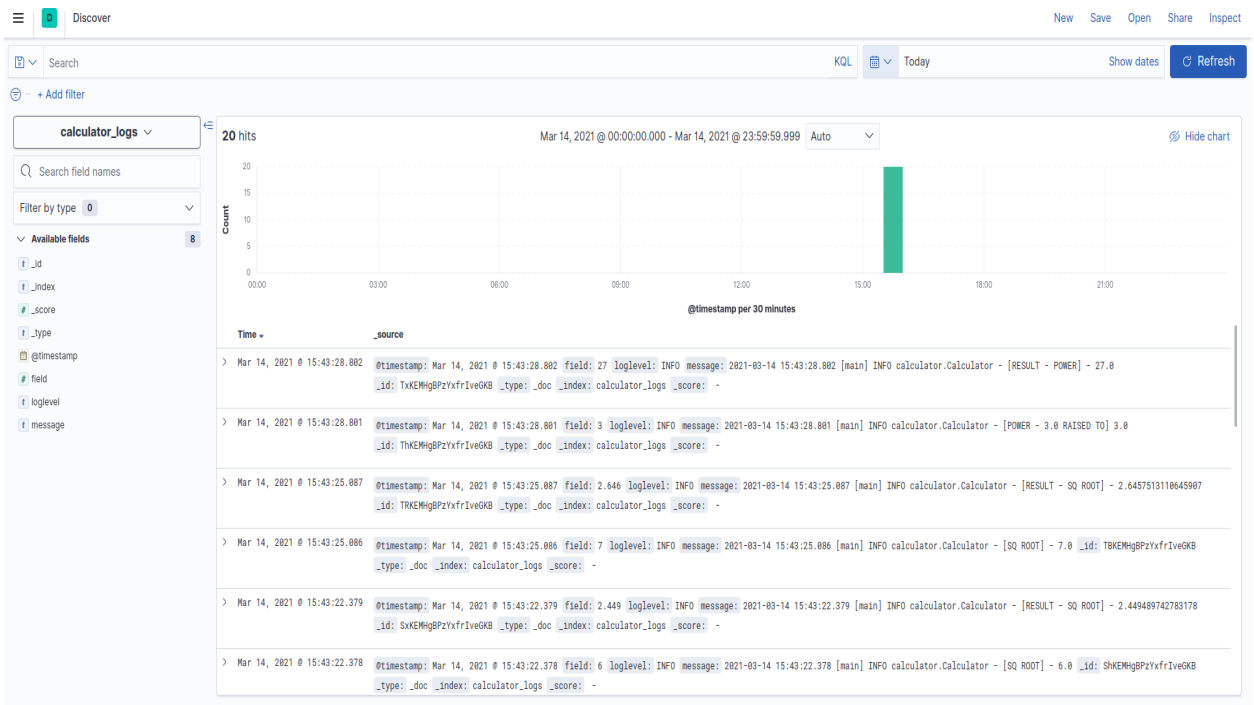


Fig: Visualization using Kibana

Challenges Faced:

1. Log file not getting generated

Solution: Added log4j.api and log4j.core dependencies to pom.xml and built the artifact using *mvn clean package* instead of *mvn clean install* and finally ran the jar file with dependencies.

2. Error in pushing docker image to Ansible remote user

Solution: Installed pip ,pip3-docker and docker on both local and remote server. Added public-private RSA key using ssh-copy-id command so that we can login into the remote server from the host user without any password.