IMPLEMENTING CI/CD PIPELINE FOR WEBSITE MONITORING

Mini Project submitted in partial fulfillment of the requirement for the award of the degree of

BACHELOR OF TECHNOLOGY

IN

COMPUTER SCIENCE AND ENGINEERING

Under the esteemed guidance of

Dr R.V.Sudhakar
Associate Professor

By

SRIJA REDDY MITTA (21R11A05E8)
MANDHA RUCHITH (21R11A05D3)
KAKUMANU GNANASWAROOP(21R11A05C5)

Department of Computer Science and Engineering
Accredited by NBA

Geethanjali College of Engineering and Technology
(UGC Autonomous)
(Affiliated to J.N.T.U.H, Approved by AICTE, New Delhi)
Cheeryal (V), Keesara (M), Medchal.Dist.-501 301.

August-2024

Geethanjali College of Engineering & Technology

(UGC Autonomous)

(Affiliated to JNTUH, Approved by AICTE, New Delhi)
Cheeryal (V), Keesara(M), Medchal Dist.-501 301.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Accredited by NBA

CERTIFICATE

This is to certify that the B.Tech Mini Project report entitled **"IMPLEMENTING CI/CD PIPELINE FOR WEBSITE MONITORING"** is a bonafide work done by **Srija Reddy Mitta (21R11A05E8), Mandha Ruchith (21R11A05D3), Kakumanu GnanaSwaroop (21R11A05C5)**, in partial fulfillment of the requirement of the award for the degree of Bachelor of Technology in "**Computer Science and Engineering**" from Jawaharlal Nehru Technological University, Hyderabad during the year 2024-2025.

Internal Guide **HOD - CSE**

Dr R.V. Sudhakar
Associate Professor

Dr A.SreeLakshmi
Professor

External Examiner

Geethanjali College of Engineering & Technology

(UGC Autonomous)

(Affiliated to JNTUH Approved by AICTE, New Delhi)
Cheeryal (V), Keesara(M), Medchal Dist.-501 301.

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

Accredited by NBA

DECLARATION BY THE CANDIDATE

I/We, **Srija Reddy Mitta, Mandha Ruchith, Kakumanu GnanaSwaroop** bearing Roll Nos.**(21R11A05E8), (21R11A05D3), (21R11A05C5),** hereby declare that the project report entitled **IMPLEMENTING CI/CD PIPELINE FOR WEBSITE MONITORING"** is done under the guidance of **Dr R.V. Sudhakar**, **Associate Professor**, Department of Computer Science and Engineering, Geethanjali College of Engineering and Technology, is submitted in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology in Computer Science and Engineering**.

This is a record of bonafide work carried out by me/us in **Geethanjali College of Engineering and Technology** and the results embodied in this project have not been reproduced or copied from any source. The results embodied in this project report have not been submitted to any other University or Institute for the award of any other degree or diploma.

Srija Reddy Mitta (21R11A05E8)

Mandha Ruchith (21R11A05D3)

Kakumanu GnanaSwaroop (21R11A05C5)

Department of CSE,

Geethanjali College of Engineering and Technology,

Cheeryal.

ACKNOWLEDGEMENT

We would like to express our sincere thanks to **Dr. A. Sree Lakshmi, Professor, Head of Department** of Computer Science, Geethanjali College of Engineering and Technology, Cheeryal, whose motivation in the field of software development has made us overcome all hardships during the course of study and successful completion of the project.

We would like to express our profound sense of gratitude to all for having helped us in completing this dissertation. We would like to express our deep-felt gratitude and sincere thanks to our guide **Dr R.V. Sudhakar, Associate Professor**, Department of Computer Science, Geethanjali College of Engineering and Technology, Cheeryal, for his skillful guidance, timely suggestions and encouragement in completing this project.

We would like to express our sincere gratitude to our **Principal Prof. Dr. S. Udaya Kumar** for providing the necessary infrastructure to complete our project. We are also thankful to our Secretary **Mr.G.R. Ravinder Reddy** for providing an interdisciplinary & progressive environment. Finally, we would like to express our heartfelt thanks to our parents who were very supportive both financially and mentally and for their encouragement to achieve our set goals.

Srija Reddy Mitta (21R11A05E8)

Mandha Ruchith (21R11A05D3)

Kakumanu GnanaSwaroop (21R11A05C5)

ABSTRACT

The project is implementing a robust CI/CD pipeline for website monitoring, integrating tools like SonarQube, Nexus, Jenkins, Docker, Kubernetes, Prometheus, and Grafana. The process begins with code commits to a version control system, triggering Jenkins to start the pipeline. Jenkins uses SonarQube for static code analysis to ensure quality and security, then interacts with Nexus to manage dependencies. The application is built and packaged into Docker containers, which are pushed to a Docker registry managed by Nexus. Kubernetes handles the deployment, scaling, and management of these containers. For monitoring, Prometheus collects performance metrics, which are visualized in Grafana for real-time insights. This integrated workflow automates quality checks, dependency management, containerization, deployment, and monitoring, leading to a more efficient and reliable software delivery process.

This CI/CD pipeline implementation integrates a suite of modern tools to automate and streamline the software development lifecycle. Each tool contributes to different stages of the pipeline, from code quality assurance to deployment and monitoring, resulting in a robust, scalable, and reliable system. This integrated approach not only enhances operational efficiency but also supports a continuous delivery model, facilitating faster and more reliable software updates.

LIST OF FIGURES

LIST OF ABBREVIATIONS

S.no	Acronym	Abbreviation
1	CI	Continuous Integration
2	CD	Continuous Deployment
3	VCS	Version Control System
4	HTTP	HyperText Transfer Protocol
5	VPC	Virtual Private Cloud
6	UML	Unified Modeling Language
7	AWS	Amazon Web Services
8	EC2	Amazon Elastic Compute Cloud

TABLE OF CONTENTS

1. INTRODUCTION

1.1 ABOUT THE PROJECT

In the modern era of software development, implementing a Continuous Integration and Continuous Deployment (CI/CD) pipeline is essential for ensuring efficient, reliable, and rapid delivery of applications. This project details the implementation of a CI/CD pipeline for website monitoring, leveraging an integrated suite of tools: SonarQube, Nexus, Jenkins, Docker, Kubernetes, Prometheus, and Grafana.

Each tool is systematically integrated to create a cohesive workflow that enhances code quality, security, scalability, and monitoring. The workflow begins with code being committed to the version control system, triggering Jenkins to start the CI/CD pipeline. Jenkins initiates the process by integrating with SonarQube to perform static code analysis, ensuring that the code meets predefined quality and security standards. Following this, Jenkins interacts with Nexus to fetch and manage dependencies, ensuring that all required libraries and components are available for the build process.

Once dependencies are resolved, Jenkins builds the application and packages it into Docker containers. These containers encapsulate the application and its dependencies, ensuring consistency across various environments. The Docker images are then pushed to a Docker registry, which can be managed by Nexus. Next, Kubernetes is employed for orchestrating the deployment of these Docker containers. Jenkins communicates with the Kubernetes cluster to deploy the containers, ensuring they are properly distributed and scaled according to the application's needs.

Kubernetes handles the dynamic scaling and management of the application, providing resilience and high availability. For monitoring the deployed application, Prometheus is integrated to collect metrics related to performance and health. These metrics are crucial for understanding the application's behavior and identifying potential issues. Grafana is used to visualize these metrics, providing developers and operations teams with detailed dashboards and alerts that offer real-time insights into the system's performance.

This implementation demonstrates a structured and integrated approach to CI/CD pipeline creation, where each tool plays a specific role in the workflow. By automating code quality checks, dependency management, build processes, containerization, orchestration, and monitoring, the pipeline not only streamlines the development process but also ensures a robust, scalable, and reliable deployment of the website monitoring application. This results in an enhanced software delivery lifecycle, promoting operational efficiency and improved performance.

1.2 OBJECTIVES

The objectives of CI/CD pipelining are to streamline software development, enhance code quality, and accelerate deployment processes. By automating build, test, and deployment tasks, CI/CD pipelines aim to achieve faster iteration cycles, reduced time-to-market, and improved collaboration among development teams. Continuous integration ensures that code changes are quickly integrated and tested, while continuous deployment automates the release process, minimizing manual intervention and reducing the risk of errors. Ultimately, CI/CD pipelines enable organizations to deliver high-quality software more efficiently, enabling them to meet customer demands and stay competitive in today's fast-paced technology landscape.

Key objectives include:

1. **Automate Code Quality Assurance:** Implement automated static code analysis with SonarQube to ensure high code quality and security.
2. **Manage Dependencies Efficiently:** Use Nexus to handle dependencies, ensuring that all required components are available and correctly managed during the build process.
3. **Ensure Consistency Across Environments:** Utilize Docker for containerization, promoting consistency and eliminating environment-related issues during development, testing, and production.

4. **Streamline Deployment:** Leverage Kubernetes for orchestrating the deployment, scaling, and management of Docker containers, enhancing application resilience and operational efficiency.
5. **Enhance Monitoring and Performance Visibility:** Integrate Prometheus for metrics collection and Grafana for visualization, providing real-time insights into application performance and health.
6. **Accelerate Software Delivery:** Automate the CI/CD processes to reduce manual intervention, speed up development cycles, and enable more frequent and reliable software updates.

2. SYSTEM ANALYSIS

2.1 EXISTING SYSTEM

Before adopting a CI/CD pipeline for deployment and monitoring of the website, several significant challenges were prevalent. Deployments were largely manual, which frequently led to human errors such as misconfigurations and missed updates, and often caused inconsistencies between development, staging, and production environments. This manual approach also resulted in long deployment times and cumbersome processes, making it difficult to quickly roll back changes or respond to issues. Furthermore, the lack of automation in testing and deployment meant that code changes could go live without adequate validation, increasing the risk of introducing bugs. Monitoring and alerting systems were typically inadequate, impeding the ability to detect and address problems swiftly. Integration issues arose from manual coordination among development, QA, and operations teams, leading to communication breakdowns and delays. Additionally, scaling deployments to manage increased traffic was challenging without automated resource management. Adopting a CI/CD pipeline was essential to streamline these processes, ensuring consistent and error-free deployments, automated testing, effective monitoring, and efficient rollback capabilities.

Implementing a CI/CD pipeline addresses these issues by automating the deployment process, ensuring consistency across environments, integrating automated testing, and providing better monitoring and rollback capabilities.

2.2 PROPOSED SYSTEM

The implementation of a Continuous Integration and Continuous Deployment (CI/CD) pipeline has become an indispensable practice for ensuring the efficient, reliable, and rapid delivery of applications. This project aims to establish a comprehensive CI/CD pipeline specifically designed for website monitoring by integrating a carefully selected suite of tools—SonarQube, Nexus, Jenkins, Docker, Kubernetes, Prometheus, and Grafana—into a unified and cohesive workflow. The pipeline begins with the initial step of code commits to a version control system, such as Git. This action triggers Jenkins, the central automation server, to initiate the CI/CD pipeline. Jenkins plays a pivotal role in orchestrating the various stages of the pipeline, starting with the integration of SonarQube for static code analysis. SonarQube evaluates the quality and security of the code, ensuring it meets predefined standards and is free of critical vulnerabilities and issues before proceeding further.

Following the code analysis, Jenkins interacts with Nexus, a repository manager, to handle and manage dependencies. Nexus ensures that all required libraries and components are available and properly versioned, facilitating a smooth and error-free build process. Once the dependencies are resolved, Jenkins proceeds to build the application and package it into Docker containers. Docker containers encapsulate the application along with its dependencies, providing a consistent and isolated environment for deployment. These Docker images are then pushed to a Docker registry managed by Nexus, where they are stored and versioned for future use.

The deployment phase involves Kubernetes, a powerful container orchestration platform. Jenkins communicates with the Kubernetes cluster to deploy the Docker containers, leveraging Kubernetes' capabilities to manage container orchestration, scaling, and high availability. Kubernetes ensures that the application is distributed and scaled according to the demands, providing resilience and efficient resource utilization.

To monitor the deployed application, Prometheus is integrated into the pipeline to collect and store metrics related to the application's performance and health. Prometheus gathers critical data on various aspects such as response times, error rates, and system resource usage. This data is essential for understanding the application's behavior and identifying potential issues proactively. Grafana complements Prometheus by visualizing the collected metrics through detailed and interactive dashboards. Grafana provides real-time insights and alerts, enabling developers and operations teams to monitor the application's performance and respond swiftly to any emerging issues.

This integrated approach to CI/CD pipeline implementation not only streamlines the development and deployment processes but also ensures that the application adheres to high standards of code quality and security. By automating key tasks, managing dependencies effectively, and leveraging containerization and orchestration technologies, the pipeline enhances operational efficiency and ensures a reliable and scalable deployment of the website monitoring application. The comprehensive monitoring and visualization capabilities provided by Prometheus and Grafana further contribute to maintaining optimal performance and quickly addressing any potential challenges, thereby promoting an efficient and high-performing software delivery lifecycle.

2.3 FEASIBILITY STUDY

2.3.1 Details

1.Technical Feasibility

Infrastructure Resources: Implementing this pipeline requires adequate infrastructure to support the tools. This includes servers or cloud resources for running Jenkins, Kubernetes clusters, and storage for Docker images and logs. The infrastructure needs to be scalable to handle varying loads, especially during peak times.

Scalability: The use of Docker and Kubernetes provides inherent scalability. Docker containers can be easily scaled horizontally, and Kubernetes can manage and orchestrate these containers effectively.

Code Security: SonarQube will help identify and mitigate security vulnerabilities in the code. Additional security practices, such as securing Jenkins and Nexus with proper authentication and access controls, are necessary.

Data Protection: Proper encryption and secure storage practices should be implemented to protect sensitive data in Docker images, logs, and monitoring metrics.

2.Operational Feasibility

Skills Required: Successful implementation requires a team with expertise in CI/CD practices, Docker, Kubernetes, and the specific tools used. This includes knowledge in configuring and maintaining Jenkins pipelines, managing Docker containers, and setting up Kubernetes clusters.

Ongoing Maintenance Management: The CI/CD pipeline will need regular updates and maintenance, including updating tools, managing configurations, and addressing any issues that arise. Automated monitoring and alerting can help in proactively managing these tasks.

3.Financial Feasibility

Licensing Costs : Some tools, like SonarQube and Nexus, have licensing costs for enterprise features. It's important to evaluate these costs against the benefits they provide.

Infrastructure Costs: Costs for cloud services here in this project AWS or on-premises hardware need to be considered. This includes expenses for server instances, storage, and network resources.

4.Implementation Feasibility

Test Phase: Start with a pilot project to test the CI/CD pipeline in a controlled environment. This phase helps identify potential issues and refine configurations before a full-scale rollout.

Quality Improvements: Enhanced code quality and security, enabled by SonarQube, and reduced deployment errors contribute to higher software quality and lower maintenance costs over time.

Monitoring and Optimization: Continuously monitor the pipeline's performance and make necessary adjustments to optimize its efficiency and effectiveness.

2.3.2 Impact on Environment

The software is programmed in such a way that its impact on the environment is very safe and does not impact the environment in the wrong way. Automating and optimizing deployment processes through tools like Kubernetes and Docker can lead to more efficient use of computational resources of the pipeline . Kubernetes enables dynamic scaling, which means resources are allocated only as needed thus potentially reducing energy consumption compared to static resource allocation.

2.3.3 Safety

The safety measures of the CI/CD pipeline for website monitoring project include:

1. Code Security: SonarQube performs static code analysis to identify vulnerabilities, and Nexus manages dependencies to avoid insecure libraries.

2. Operational Stability: Automated testing catches issues early, and Kubernetes ensures stability with automated deployment and scaling.

3. Data Integrity: Docker containers ensure consistent environments, while version control maintains code traceability.

4. System Resilience: Prometheus and Grafana provide real-time monitoring and alerts, enabling proactive issue resolution.

5. Disaster Recovery: Regular backups and Kubernetes’ self-healing capabilities support effective recovery from failures.

2.3.4 Ethics

The application follows the general software ethics. We have followed the general SW ethics not harming anybody physically or virtually, and maintaining confidentiality.

2.3.5 Cost

The cost of the project is very low and can be implemented virtually with anyone that has proper technological access and continuous internet connection.

2.3.6 Type

The project is a software development project that is a web application that is used to deploy , monitor and manage websites all at once, providing a comfortable user-friendly interface.

2.3.7 Standards

Agile model is used that has various advantages such as Iterative and Incremental Development , Adaptive Planning , Flexibility and Responsiveness.

2.4 SCOPE OF THE PROJECT

Incorporating the CI/CD pipelining for application deployment , monitoring and management purposes can be met with a broader scope. The project aims to implement a robust CI/CD pipeline integrated with comprehensive website monitoring to enhance website deployment and performance tracking. It involves setting up automated build and deployment processes, integrating source control, and automating testing to ensure reliable software delivery. Additionally, the project includes configuring performance, uptime, and error monitoring, as well as establishing real and synthetic user interaction tracking. Documentation and training for development and operations teams are essential, alongside thorough testing, validation, and post-implementation review for continuous improvement. The project excludes integration with legacy systems or non-web applications and advanced analytics beyond basic monitoring.

2.5 SYSTEM CONFIGURATION

Software components

- **Continuous Integration (CI) Tool**: Jenkins
- **Continuous Deployment/Delivery (CD) Tool**: Docker and Kubernetes
- **Configuration Management and Containerization :** Docker
- **Website Performance Monitoring:** Grafana and Prometheus
- **Build Automation Tools :** Maven
- **Vulnerability Scanning Tools** : SonarQube
- **User Update and Interaction Tool** : Gmail

2.6 HARDWARE COMPONENTS

Any system with

- 8 GB RAM
- Multi-core CPU processors
- High bandwidth network connections
- Additional storage access for backups

3. LITERATURE OVERVIEW

“The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations” by Gene Kim, Patrick Debois, John Willis, and Jez Humble”

As Christopher Little, a software executive and one of the earliest chroniclers of DevOps, said, “Every company is a technology company, regardless of what business they think they’re in. A bank is just an IT company with a banking license.”† † In 2013, the European bank HSBC employed more software developers than Google. Promo - Not for distribution or sale xxviii • The DevOps Handbook To convince ourselves that this is the case, consider that the vast majority of capital projects have some reliance upon IT. As the saying goes, “It is virtually impossible to make any business decision that doesn’t result in at least one IT change.” In the business and finance context, projects are critical because they serve as the primary mechanism for change inside organizations.

Projects are typically what management needs to approve, budget for, and be held accountable for; therefore, they are the mechanism that achieve the goals and aspirations of the organization, whether it is to grow or even shrink.† Projects are typically funded through capital spending (i.e., factories, equipment, and major projects, and expenditures are capitalized when payback is expected to take years), of which 50% is now technology related. This is even true in “low tech” industry verticals with the lowest historical spending on technology, such as energy, metal, resource extraction, automotive, and construction. In other words, business leaders are far more reliant upon the effective management of IT in order to achieve their goals than they think.

Ideally, small teams of developers independently implement their features, validate their correctness in production-like environments, and have their code deployed into production quickly, safely and securely. Code deployments are routine and predictable. Instead of starting deployments at midnight on Friday and spending all weekend working to complete them, deployments occur throughout the business day when everyone is already in the office and without our customers even noticing—except when they see new features and bug fixes that delight them. And, by deploying code in the middle of the workday, for the first time in decades IT Operations is working during normal business hours like everyone else. By creating fast feedback loops at every step of the process, everyone can immediately see the effects of their actions.

Whenever changes are committed into version control, fast automated tests are run in production-like environments, giving continual assurance that the code and environments operate as designed and are always in a secure and deployable state. Automated testing helps developers discover their mistakes quickly (usually within minutes), which enables faster fixes as well as genuine learning— learning that is impossible when mistakes are discovered six months later during integration testing, when memories and the link between cause and effect have long faded. Instead of accruing technical debt, problems are fixed as they are found, mobilizing the entire organization if needed, because global goals outweigh local goals. Pervasive production telemetry in both our code and production environments ensure that problems are detected and corrected quickly, confirming that everything is working as intended and customers are getting value from the software we create.

Humple, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

"The principles, practices, and techniques we describe are aimed at making releases boring, even in complex "enterprise" environments. Software release can—and should—be a low-risk, frequent, cheap, rapid, and predictable process. These practices have been developed over the last few years, and we have seen them make a huge difference The Problem of Delivering Software Download from www.wowebook.com in many projects. All of the practices in this book have been tested in large enterprise projects with distributed teams as well as in small development groups. We know that they work, and we know that they scale to large projects.

Our goal is to describe the use of deployment pipelines, combined with high levels of automation of both testing and deployment and comprehensive configuration management to deliver push-button software releases. That is, push-button software releases to any deployment target—development, test, or production. Along the way we will describe the pattern itself and the techniques that you will need to adopt to make it work. We will provide advice on different approaches to solving some of the problems that you will face. We have found that the advantages of such an approach vastly outweigh the costs of achieving it. None of this is outside the reach of any project team. It does not require rigid process, significant documentation, or lots of people. By the end of this chapter, we hope that you will understand the principles behind this approach.
As we said, our goal as software professionals is to deliver useful, working software to users as quickly as possible. Speed is essential because there is an opportunity cost associated with not delivering software. You can only start to get a return on your investment once your software is released. So, one of our two overriding goals in this book is to find ways to reduce cycle time, the time it takes from deciding to make a change, whether a bugfix or a feature, to having it available to users.

Delivering fast is also important because it allows you to verify whether your features and bug fixes really are useful. The decision maker behind the creation of an application, who we'll call the customer, makes hypotheses about which features and bugfixes will be useful to users. However, until they are in the hands of users who vote by choosing to use the software, they remain hypotheses. It is therefore vital to minimize cycle time so that an effective feedback loop can be established."

https://about.gitlab.com/topics/ci-cd/

CI/CD falls under DevOps (the joining of development and operations teams) and combines the practices of continuous integration and continuous delivery. CI/CD automates much or all of the manual human intervention traditionally needed to get new code from a commit into production, encompassing the build, test (including integration tests, unit tests, and regression tests), and deploy phases, as well as infrastructure provisioning. With a

CI/CD pipeline, development teams can make changes to code that are then automatically tested and pushed out for delivery and deployment. Get CI/CD right and downtime is minimized and code releases happen faster.
A CI/CD pipeline is an automated process utilized by software development teams to streamline the creation, testing and deployment of applications. "CI" represents continuous integration, where developers frequently merge code changes into a central repository, allowing early detection of issues. "CD" refers to continuous deployment or continuous delivery, which automates the application's release to its intended environment, ensuring that it is readily available to users. This pipeline is vital for teams aiming to improve software quality and speed up delivery through regular, reliable updates.

Integrating a CI/CD pipeline into your workflow significantly reduces the risk of errors in the deployment process. Automating builds and tests ensures that bugs are caught early and fixed promptly, maintaining high-quality software.

Hightower, K., Burns, B., & Beda, J. (2017). ***Kubernetes Up & Running: Dive into the Future of Infrastructure*****. O'Reilly Media.**

“Legend has it that Google deploys over two billion application containers a week. How’s that possible? Google revealed the secret through a project called Kubernetes, an open source cluster orchestrator (based on its internal Borg system) that radically simplifies the task of building, deploying, and maintaining scalable distributed systems in the cloud. This practical guide shows you how Kubernetes and container technology can help you achieve new levels of velocity, agility, reliability, and efficiency. Authors Kelsey Hightower, Brendan Burns, and Joe Bedawhove worked on Kubernetes at Google and other organizatons explain how this system fits into the lifecycle of a distributed application.
You will learn how to use tools and APIs to automate scalable distributed systems, whether it is for online services, machine-learning applications, or a cluster of Raspberry Pi computers. Explore the distributed system challenges that Kubernetes addresses Dive into containerized application development, using containers such as Docker Create and run

containers on Kubernetes, using the docker image format and container runtime Explore specialized objects essential for running applications in production Reliably roll out new software versions without downtime or errors Get examples of how to develop and deploy real-world applications in Kubernetes”

“DevOps for Developers” by Michael Hüttermann

The term DevOps is a blend of development (representing software developers, including programmers, testers, and quality assurance personnel) and operations (representing the experts who put software into production and manage the production infrastructure, including system administrators, database administrators, and network technicians). DevOps describes practices that streamline the software delivery process, emphasizing the learning by streaming feedback from production to development and improving the cycle time (i.e., the time from inception to delivery; see more about this process in Chapter 3). DevOps will not only empower you to deliver software more quickly, but it will also help you to produce higher-quality software that is more aligned with individual requirements and basic conditions. DevOps encompasses numerous activities and aspects, such as the following

Culture: People over processes and tools. Software is made by and for people.

Automation: Automation is essential for DevOps to gain quick feedback.

Measurement: DevOps finds a specific path to measurement. Quality and shared (or at least aligned) incentives are critical.

Sharing: Creates a culture where people share ideas, processes, and tools.

The term DevOps is a slightly overloaded one. To understand the scope of the DevOps concept, it helps to discuss what DevOps is not. DevOps is not a marketing (buzz) term. Although some aspects of DevOps are not new, it is a new and strong movement intended to improve the delivery process. The DevOps approach accepts the daily challenges in software delivery and provides steps to address them. DevOps does not allow developers to work on the production system. “

4.SYSTEM DESIGN

4.1 SYSTEM ARCHITECTURE

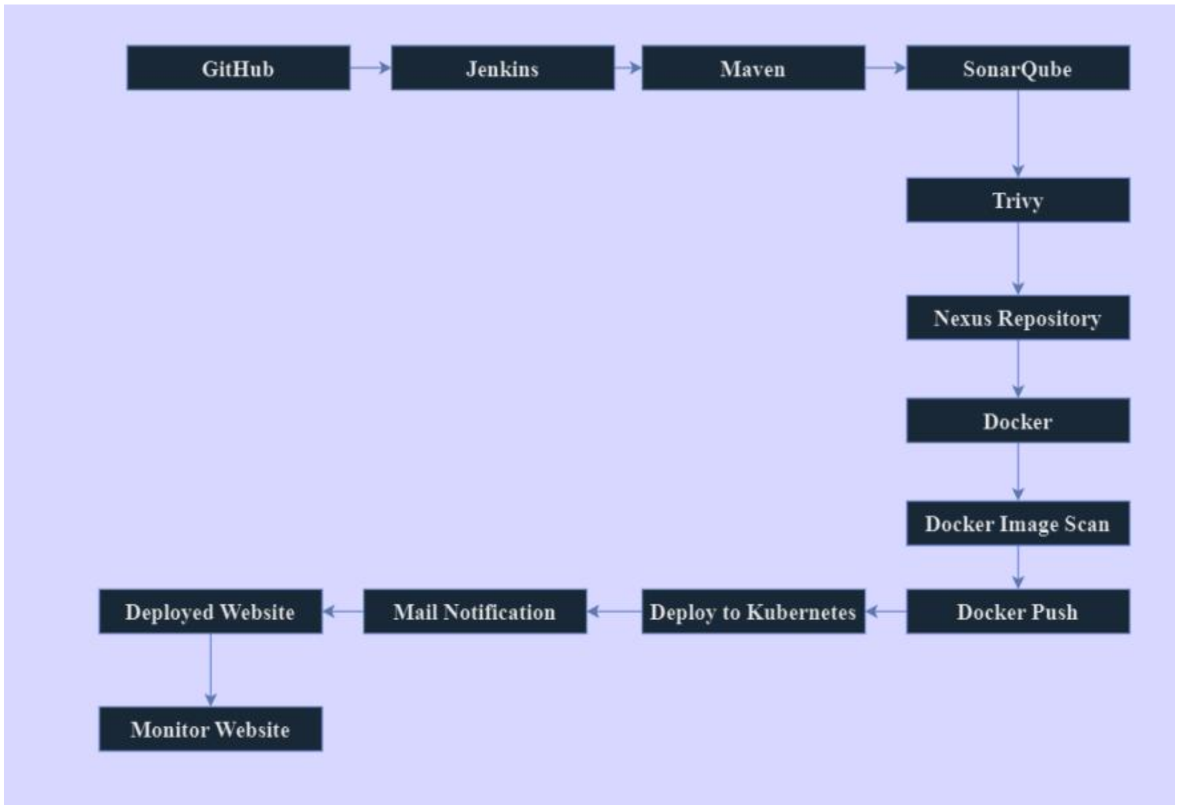


Fig 4.4.1 System Architecture

CI/CD pipeline for website monitoring integrates a suite of tools to ensure a streamlined and effective development process. It begins with code commits to a version control system, which triggers Jenkins to orchestrate the pipeline. Jenkins integrates with SonarQube to perform static code analysis, ensuring code quality and security. It then interacts with Nexus to manage and fetch dependencies, followed by building and packaging the application into Docker containers for consistency across environments. These Docker images are pushed to a Docker registry managed by Nexus. Kubernetes takes over for orchestrating the deployment, scaling, and management of these containers, ensuring resilience and high availability.

For monitoring, Prometheus collects performance and health metrics, while Grafana visualizes these metrics through detailed dashboards and alerts. This comprehensive setup automated code quality checks, dependency management, containerization, deployment, and monitoring, resulting in a robust, scalable, and efficient software delivery lifecycle.

4.1.1 Module Description

Phase 1: Network Environment - all the network connections have to be private,isolated,secure in terms of deployment There is K8 Deployment involved and we also scan for bugs. Creation of virtual machines is also part of this phrase. Vms for Sonarcube , Jenkins and various monitoring applications.

Phase 2: In this phrase push a desired source code into a private Git repository by uploading it to the Git.

Phase 3 : In this phase we perform the core of the project that is working with CI/CD pipeline with practices and security measures. This also includes deployment of the application and configuration of mail notification.

Phase 4: In the last and final phase we set up monitoring tools at two levels :System level for CPU and RAM and Website level for traffic control and monitoring.

4.2 UML Diagrams

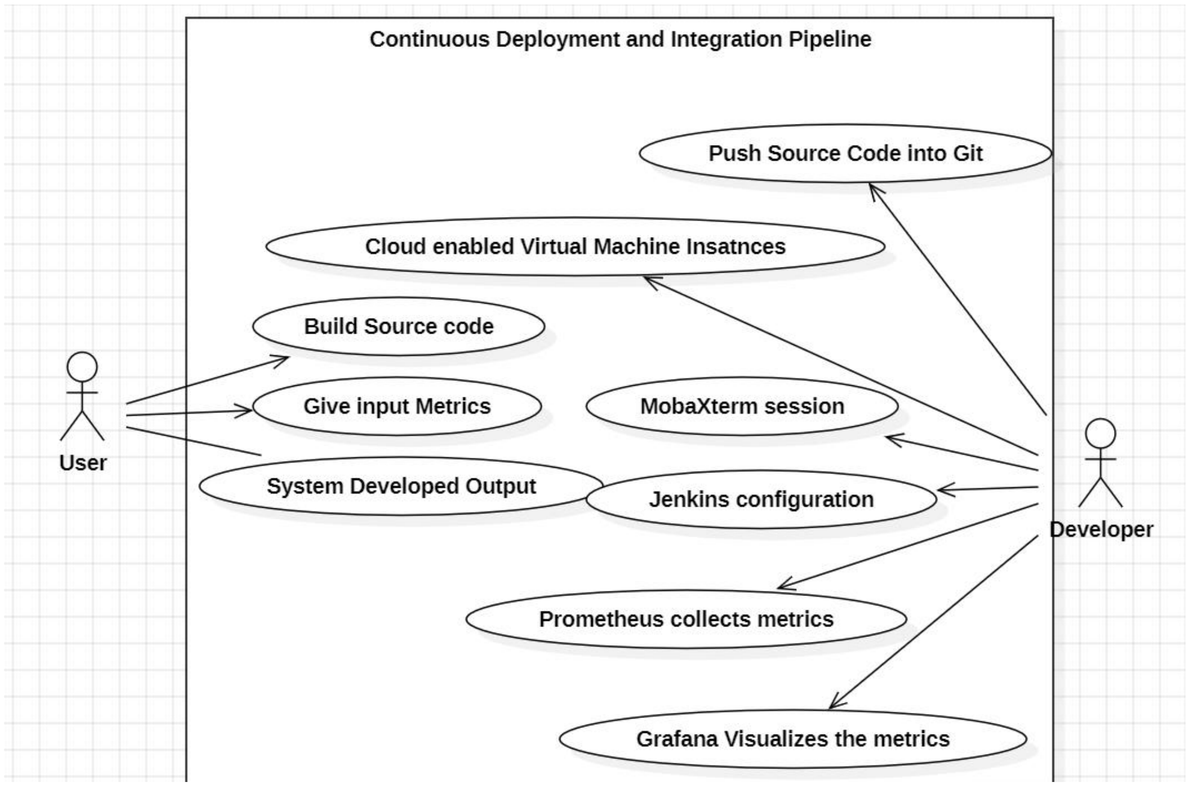

Continuous Deployment and Integration Pipeline
Push Source Code into Git
Cloud enabled Virtual Machine Insatnces
Build Source code
Give input Metrics
MobaXterm session
User
System Developed Output
Jenkins configuration
Developer
Prometheus collects metrics
Grafana Visualizes the metrics

Fig 4.2.1 Use case diagram

User Interaction

User: The process begins with the user providing input. The user can build the source code and give input metrics, which are fed into the system.

Build Source Code: The source code is compiled or built on these cloud-enabled virtual machines.

Give Input Metrics: The user provides metrics that are necessary for monitoring or for other aspects of the deployment process.

System Management

Push Source Code into Git: The developer pushes the source code into a Git repository.

Cloud-enabled Virtual Machine Instances: The virtual machines in the cloud are utilized to perform various tasks. The source code is built on these machines.

MobaXterm Session: A session in MobaXterm is used, possibly for accessing the virtual machines or running commands remotely.

Jenkins Configuration: Jenkins is configured to automate tasks such as building the code, running tests, and deploying the application.

Prometheus Collects Metrics: Prometheus is used to collect various metrics from the system, which could include performance data, resource usage, etc.

Grafana Visualizes the Metrics: Grafana takes the metrics collected by Prometheus and visualizes them, allowing for easy monitoring and analysis

System Developed Output: The output of the system is generated, which could be the deployed application or some form of processed data.

Developer Interaction

Developer: The developer interacts with the system, particularly with the Git repository and Jenkins, to manage the codebase and configuration.

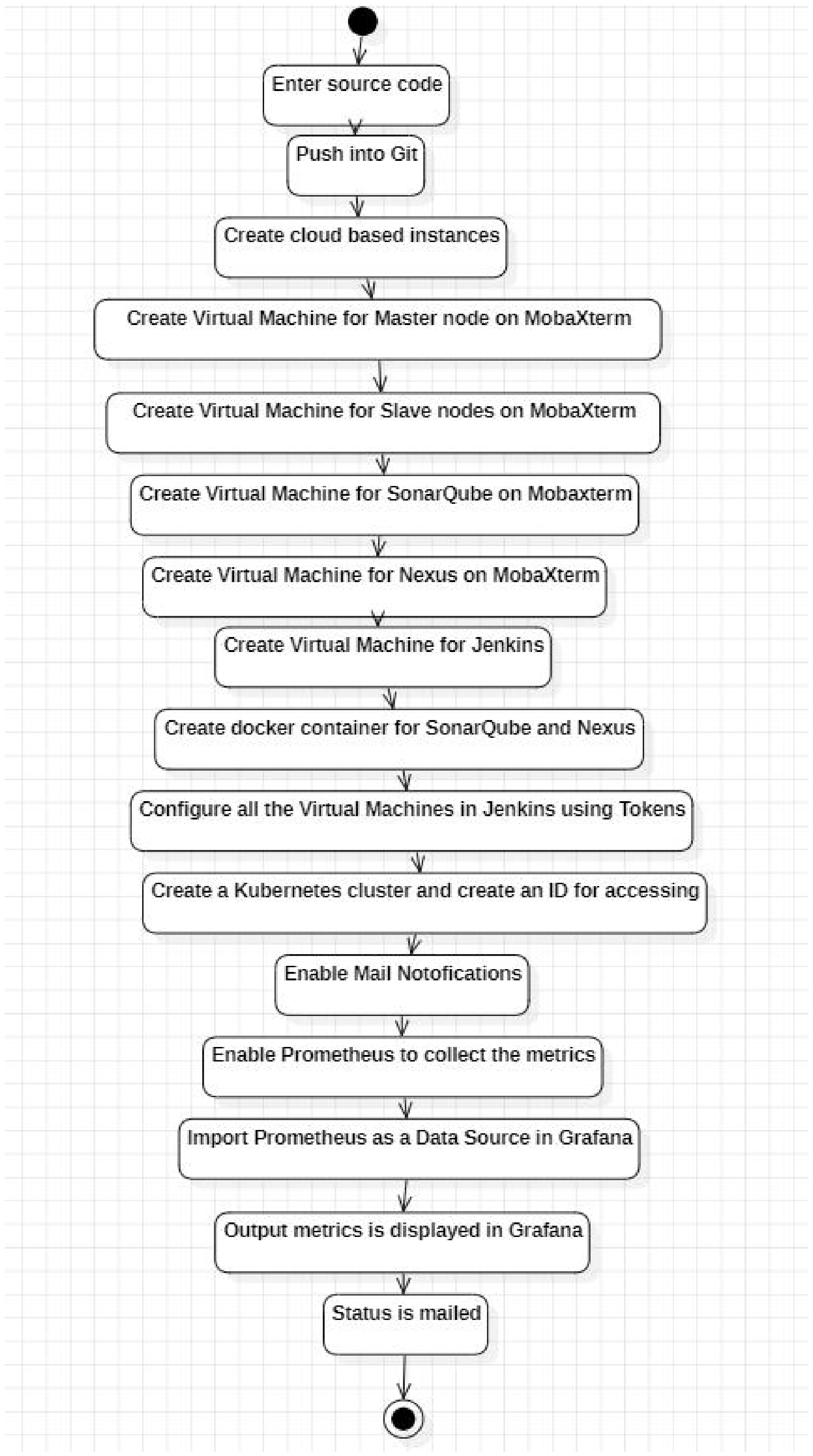

Enter source code
Push into Git
Create cloud based instances
Create Virtual Machine for Master node on MobaXterm
Create Virtual Machine for Slave nodes on MobaXterm
Create Virtual Machine for SonarQube on Mobaxterm
Create Virtual Machine for Nexus on MobaXterm
Create Virtual Machine for Jenkins
Create docker container for SonarQube and Nexus
Configure all the Virtual Machines in Jenkins using Tokens
Create a Kubernetes cluster and create an ID for accessing
Enable Mail Notofications
Enable Prometheus to collect the metrics
Import Prometheus as a Data Source in Grafana
Output metrics is displayed in Grafana
Status is mailed

Fig 4.2.2 Activity Diagram

4.3 SYSTEM DESIGN

4.3.1 Modular Design

1. Code Integration and Build Module

The code integration process in a CI/CD pipeline is essential for ensuring that changes from multiple developers are consistently integrated into a shared codebase. This process typically starts with developers pushing their code to a version control system (VCS) such as Git. The pipeline is configured to automatically trigger when changes are detected, usually through webhooks or Git hooks. A well-defined branching strategy, like GitFlow or trunk-based development, is crucial here. Developers work on feature branches, and changes are merged into the main branch after passing code reviews and automated tests. This approach helps detect integration issues early, reducing the risk of conflicts and ensuring that the main branch remains stable.

2. Deployment Module

The deployment module in a CI/CD pipeline is responsible for automating the process of delivering build artifacts to various environments, such as staging, production, or other testing environments. This module ensures that deployments are consistent, reliable, and repeatable, reducing the risk of errors that can occur with manual deployments.

In a CI/CD pipeline that utilizes Docker and Kubernetes, the deployment module automates the process of transitioning an application from development to production using containerization and orchestration.

The process begins with containerization using Docker. During the build stage, Docker packages the application and its dependencies into a Docker image, ensuring consistency across various environments. A Dockerfile defines how the Docker image is built, specifying the installation of dependencies, the application code, and the entry point for the container. Once the Docker image is created, it is pushed to a container registry, such as Docker Hub, Google Container Registry, or a private registry. This image serves as a portable and consistent artifact that can be deployed across different environments.

The deployment to staging involves deploying the Docker image to a staging environment managed by Kubernetes. Kubernetes provides a robust orchestration platform that handles the deployment, scaling, and management of containerized applications. Following successful validation in staging, the pipeline proceeds with the production deployment. Kubernetes facilitates this process through its orchestration capabilities. The Docker image is deployed to the production cluster, where Kubernetes manages the rollout. Deployment strategies such as blue-green deployment or canary deployment can be employed to minimize risk. In a blue-green deployment, two environments (blue and green) are maintained. The application is first deployed to the green environment, and once it is confirmed to be stable, traffic is switched from the blue environment to the green one. Canary deployments gradually roll out the new version to a subset of users before a full-scale deployment, allowing for early detection of issues.

Throughout the deployment process, Kubernetes manages the scaling, health checks, and rolling updates of the application, ensuring minimal downtime and high availability. By leveraging Docker and Kubernetes, the deployment module ensures that the application is consistently and reliably deployed, from development through to production, with robust handling of scaling and management tasks.

3. Website Monitoring Module

The website monitoring module in a CI/CD pipeline is crucial for ensuring that the website remains reliable, performant, and secure throughout its lifecycle. This module encompasses several key components, each focused on different aspects of website health.

Prometheus serves as the core monitoring and alerting toolkit. It collects and stores time-series data from various sources, including application endpoints, infrastructure components, and other services. Prometheus uses a pull-based model to scrape metrics from configured endpoints at specified intervals. This approach allows it to gather detailed performance data, such as response times, error rates, and resource utilization. Prometheus also supports powerful querying capabilities through its PromQL query language, enabling users to aggregate and analyze metrics to gain deep insights into application health and performance.

Grafana complements Prometheus by providing a robust visualization layer. It connects to Prometheus as a data source and allows users to create interactive and customizable dashboards. Grafana’s rich set of visualization options, including graphs, heatmaps, and gauges, enables users to present metrics in a meaningful and visually appealing way. These dashboards can be tailored to display critical performance indicators, such as server load, request latency, and error rates, helping teams to monitor the application's behavior and detect issues quickly.

Uptime monitoring can be enhanced with Prometheus by instrumenting the application and infrastructure to expose metrics related to availability. For instance, Prometheus can scrape metrics from a web server to monitor HTTP status codes and response times. Grafana dashboards can then visualize these metrics, providing real-time insights into the website’s uptime and availability.

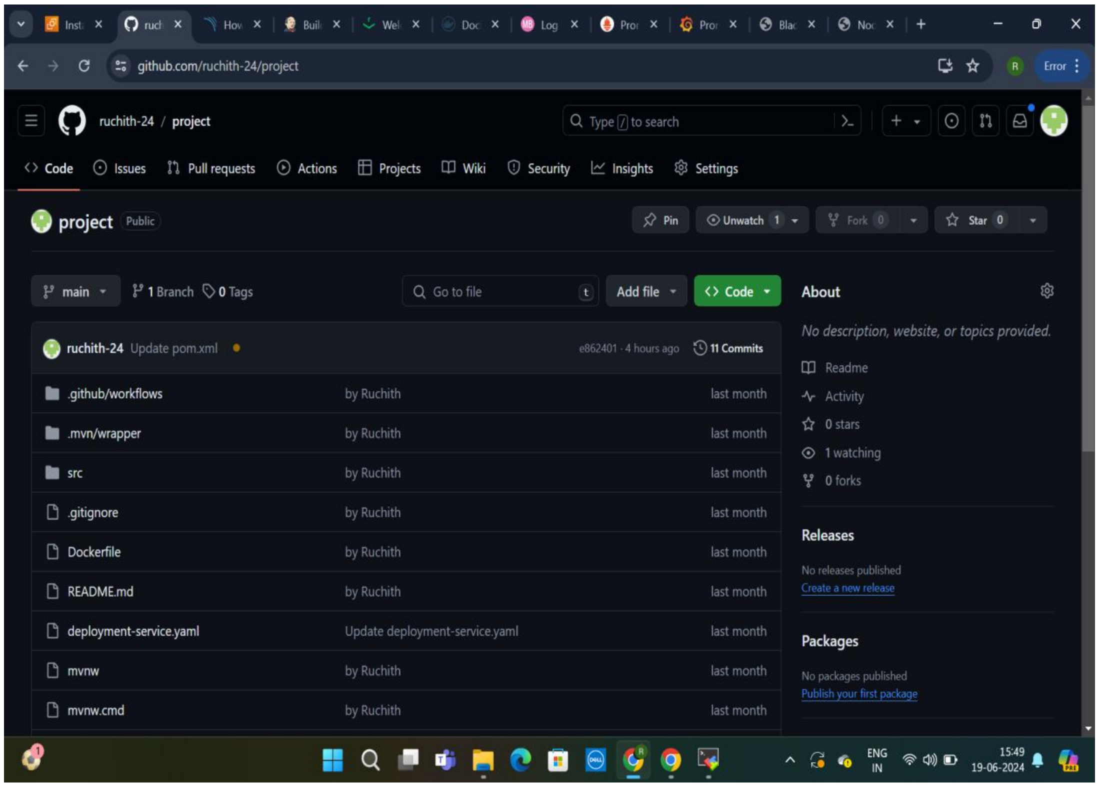
Performance monitoring benefits from Prometheus’s ability to track detailed performance metrics. By configuring Prometheus to collect metrics like page load times, server response times, and resource usage, teams can monitor application performance closely. Grafana dashboards can display these metrics in various formats, helping to identify performance bottlenecks and trends over time.

4. Alerting Module

The alerting module in a CI/CD pipeline is essential for quickly detecting and resolving issues during development, build, and deployment stages. It begins by defining alerting rules based on specific metrics, logs, or events that impact application performance and stability, such as build failures or deployment issues. This module integrates closely with monitoring tools like Prometheus and Grafana, which help set up and manage alerts. Notifications are then sent through channels like email to ensure that the right team members are informed. Effective alert management includes triaging alerts, managing incidents, and regularly reviewing and optimizing alerting rules to prevent fatigue and ensure continued relevance and helps to maintain operational awareness and ensures the stability and reliability of the application.

4.3.2 Database Design

The database that is used in the project is a private Git repository. The source code for the desired website is displayed in a git repository. We opt for a private Git repository because we want the network environment to be isolated and secure. The source code within the repository is pushed into the Git.



github.com/ruchith-24/project

ruchith-24 / project

project Public

main · 1 Branch · 0 Tags

ruchith-24 Update pom.xml · e862401 · 4 hours ago · 11 Commits

.github/workflows	by Ruchith	last month
.mvn/wrapper	by Ruchith	last month
src	by Ruchith	last month
.gitignore	by Ruchith	last month
Dockerfile	by Ruchith	last month
README.md	by Ruchith	last month
deployment-service.yaml	Update deployment-service.yaml	last month
mvnw	by Ruchith	last month
mvnw.cmd	by Ruchith	last month

About

No description, website, or topics provided.

Readme · Activity · 0 stars · 1 watching · 0 forks

Releases

No releases published
Create a new release

Packages

No packages published
Publish your first package

Fig 4.3.2.1 Database Design

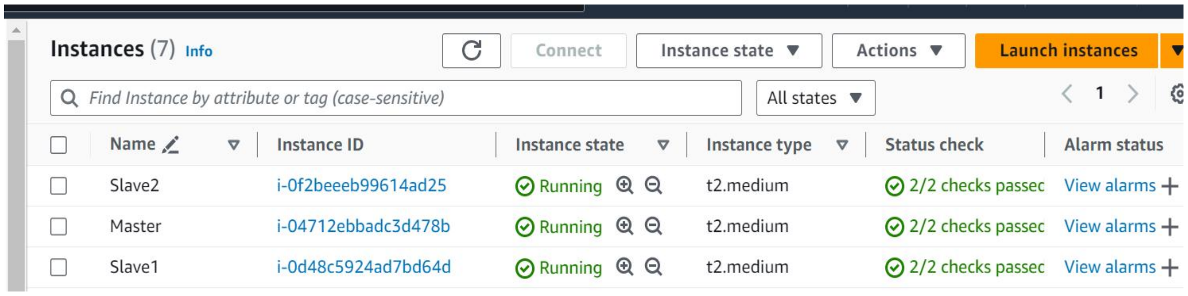
5. IMPLEMENTATION

5.1 IMPLEMENTATION

Phase 1

Network Environment

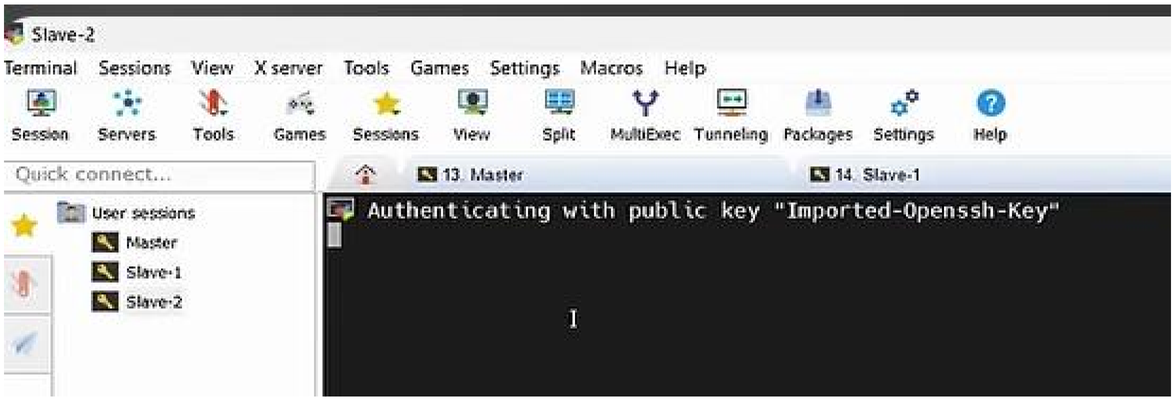
1. Have an AWS account.
2. Download MobaXterm to work with Virtual Machines.
3. Back to AWS - Search for Virtual Private Cloud (VPC)
4. Create a new VPC
5. Go to EC2 to Determine the access rights go to Security Groups. There should exist a default security group but create a new security group called 'primary'.
6. Click on the security group link and preview open ports such as custom TCP, SSH, HTTP,HTTPS.
7. Back to EC2 Dashboard , Go to instances . Launch three new instances of Ubuntu. Amazon Machine Image being Ubuntu Server 22.04 LTS(HVM) ,SSD Volume Type. Instance type being 4gb (t2.medium).
8. Now for the key pair , there should be a pre-existing else create new RSA private key , as soon as you create a key it gets downloaded in the system.
9. Scroll below to network settings and select the security group.
10. Enter 25 GB for Configure Storage.
11. Click on Launch Instance. Now name them.
12. First will be Master , second will be Slave1 and third will be Slave2. Now we configure them.



Name	Instance ID	Instance state	Instance type	Status check	Alarm status
Slave2	i-0f2beeeb99614ad25	Running	t2.medium	2/2 checks passec	View alarms
Master	i-04712ebbadc3d478b	Running	t2.medium	2/2 checks passec	View alarms
Slave1	i-0d48c5924ad7bd64d	Running	t2.medium	2/2 checks passec	View alarms

Fig 5.1.1 Instances

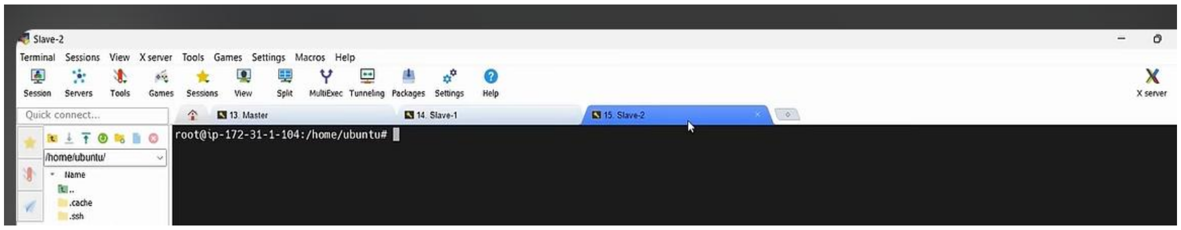
13. Click on Session-> SSH -> Paste public IP address for the master node from AWS instance and user default name as ubuntu -> create new sessions named ‘Master’, ‘Slave 1’ and ‘Slave 2’.


Slave-2
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
13. Master
14. Slave-1
User sessions
Master
Slave-1
Slave-2

```
Authenticating with public key "Imported-Openssh-Key"
```

Fig 5.1.2 Slave 1

14. Advance SSH setting -> use private key -> to browse -> click on the downloaded key.
15. Become root user on all the nodes by entering *sudo su*

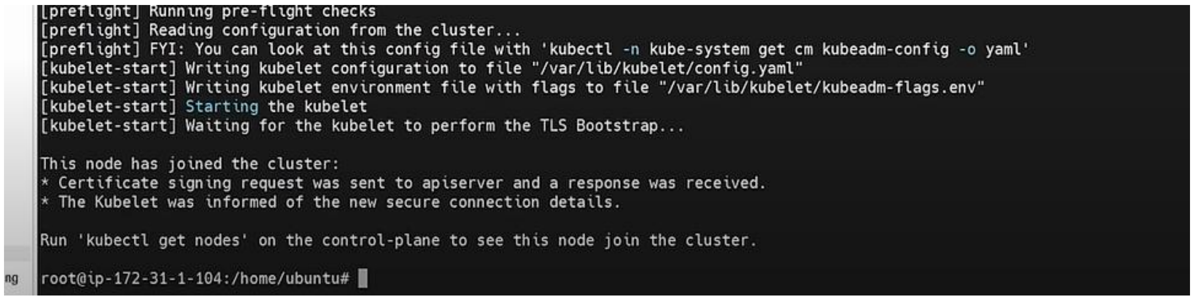

Slave-2
Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
X server
Quick connect...
13. Master
14. Slave-1
15. Slave-2
/home/ubuntu/
Name
..
.cache
.ssh

```
root@ip-172-31-1-104:/home/ubuntu#
```

Fig 5.1.3 Slave 2

Implementation of Kubernetes Clusters.

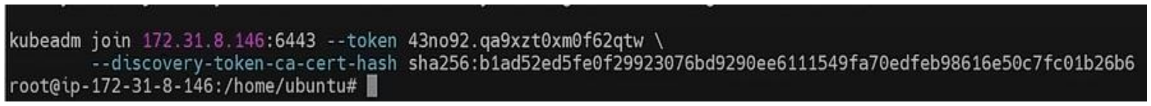
16. Few commands have to be executed in the individual node to enable kubernetes. The nodes join the cluster.



```
[preflight] Running pre-flight checks
[preflight] Reading configuration from the cluster...
[preflight] FYI: You can look at this config file with 'kubectl -n kube-system get cm kubeadm-config -o yaml'
[kubelet-start] Writing kubelet configuration to file "/var/lib/kubelet/config.yaml"
[kubelet-start] Writing kubelet environment file with flags to file "/var/lib/kubelet/kubeadm-flags.env"
[kubelet-start] Starting the kubelet
[kubelet-start] Waiting for the kubelet to perform the TLS Bootstrap...

This node has joined the cluster:
* Certificate signing request was sent to apiserver and a response was received.
* The Kubelet was informed of the new secure connection details.

Run 'kubectl get nodes' on the control-plane to see this node join the cluster.

root@ip-172-31-1-104:/home/ubuntu#
```

Fig 5.1.4 Implementation of kubernetes Clusters

17. We get a kubernetes ID that we can use to connect other virtual machines to the same cluster as the previously connected clusters. The template kubernetes Id is displayed as

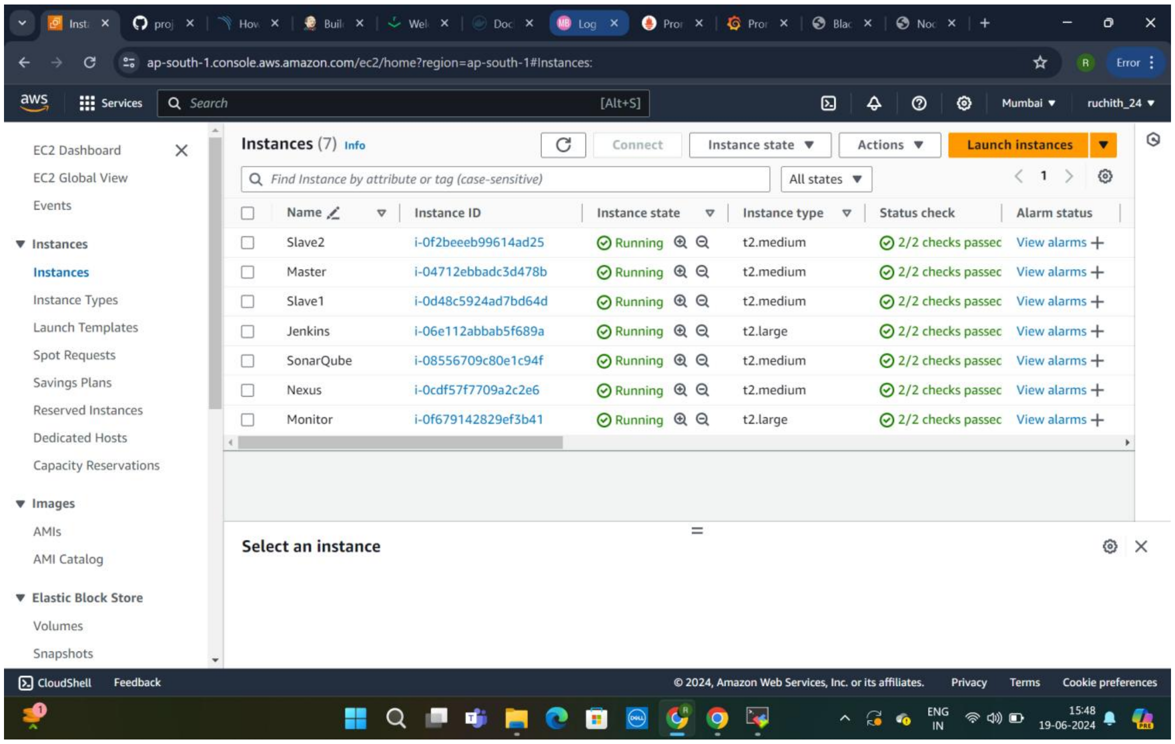


```
kubeadm join 172.31.8.146:6443 --token 43no92.qa9xzt0xm0f62qtw \
        --discovery-token-ca-cert-hash sha256:b1ad52ed5fe0f29923076bd9290ee6111549fa70edfeb98616e50c7fc01b26b6
root@ip-172-31-8-146:/home/ubuntu#
```

Fig 5.1.5 Implementation of kubernetes Clusters

18. Create a directory using *mkdir -p $HOME/.kube* to create a file that consists of the configuration of the kubernetes.
19. We scan the kubernetes cluster for any kind of issues within it for that there are multiple tools - here we use *kuberaudit.* Download kuberaudit in the master node of MobaXterm.

Create Virtual Machines to configure servers and tools



Name	Instance ID	Instance state	Instance type	Status check	Alarm status
Slave2	i-0f2beeeb99614ad25	Running	t2.medium	2/2 checks passec	View alarms
Master	i-04712ebbadc3d478b	Running	t2.medium	2/2 checks passec	View alarms
Slave1	i-0d48c5924ad7bd64d	Running	t2.medium	2/2 checks passec	View alarms
Jenkins	i-06e112abbab5f689a	Running	t2.large	2/2 checks passec	View alarms
SonarQube	i-08556709c80e1c94f	Running	t2.medium	2/2 checks passec	View alarms
Nexus	i-0cdf57f7709a2c2e6	Running	t2.medium	2/2 checks passec	View alarms
Monitor	i-0f679142829ef3b41	Running	t2.large	2/2 checks passec	View alarms

Fig 5.1.6 Creating virtual machines

20. Go back to AWS and launch 2 new instances that will be used for SonarQube. Select the existing security group. Storage should be around 20 that is more than enough for the application.
21. Back to the instances dashboard , select the pending instances and rename them as ‘SonarQube’ and ‘Nexus’.
22. Get the public IP addresses , go back to MobaXterm and now duplicate the Master/ Slave session and now rename it as SonarQube. Now we have a SonarQube VM. Paste the public IP address from the AWS. The key has to be the same.
23. Repeat the same with Nexus.
24. Now we create a VM for Jenkins. For Jenkins we need a bigger resource so the storage to be selected for this instance has to be t2.large and configure storage has to be 30gb. Back to the instances dashboard and now rename the pending instance as Jenkins. Go to MobaXterm and create a session similar to the rest and paste the public IP address at the SSH and click ok. Session for Jenkins has been generated.
25. On the SonarQube and Nexus session run *sudo apt update.* To use any resources in the virtual machine from the repository , we run this command.
26. Now install Docker in SonarQube and Nexus virtual machine using the *for pkg in docker.io docker-doc docker-compose docker-compose-v2 podman-docker containerd runc; do sudo apt-get remove $pkg; done*
27. Create a docker container for SonarQube and Nexus with the port number 9000 and 8081 respectively.
28. Sign in into the SonarQube and Nexus websites with userid and password. Now SonarQube and Nexus servers are up and running.
29. Install JDK 17 in Jenkins VM. Now install docker in Jenkins.
30. Go to Jenkins website and install all the necessary plugins.

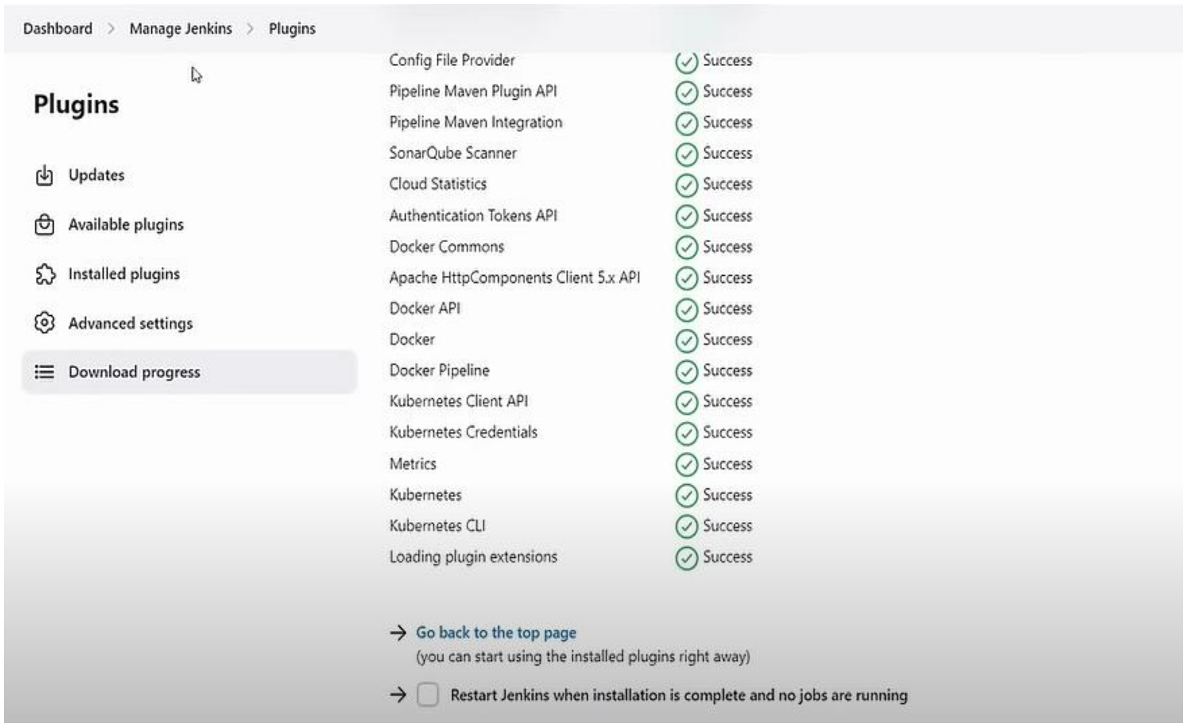
Phase 2

Git

1. Create a GitHub account and create a new repository. The repository named in the project is “project”. Make sure the repository is private.
2. Install GitBash based on the OS. Here we have used the Windows version.
3. Push the desired java web application into the repository using GitBash.
4. Create a new git personal access token (classic) for enabling configuration with jenkins and kubernetes. Now under select scopes we can add the application scopes for the web application by selecting each scope box.

Configuring Jenkins

5. Open Jenkins main website by using IP address followed by port 8080.
6. Go to Manage -> Plugins -> Available Plugins.
7. The following plugins have to installed :
   - Eclipse Terinum installer
   - Config File Provider
   - Pipeline Maven Integration
   - SonarQube Scanner
   - Docker
   - Docker pipeline
   - docker-build-step
   - Maven Integration
   - Kubernetes
   - Kubernetes CLI
   - Kubernetes Client API
   - Kubernetes Credentials


Dashboard > Manage Jenkins > Plugins

Config File Provider	✓ Success
Pipeline Maven Plugin API	✓ Success
Pipeline Maven Integration	✓ Success
SonarQube Scanner	✓ Success
Cloud Statistics	✓ Success
Authentication Tokens API	✓ Success
Docker Commons	✓ Success
Apache HttpComponents Client 5.x API	✓ Success
Docker API	✓ Success
Docker	✓ Success
Docker Pipeline	✓ Success
Kubernetes Client API	✓ Success
Kubernetes Credentials	✓ Success
Metrics	✓ Success
Kubernetes	✓ Success
Kubernetes CLI	✓ Success
Loading plugin extensions	✓ Success

→ Go back to the top page
(you can start using the installed plugins right away)

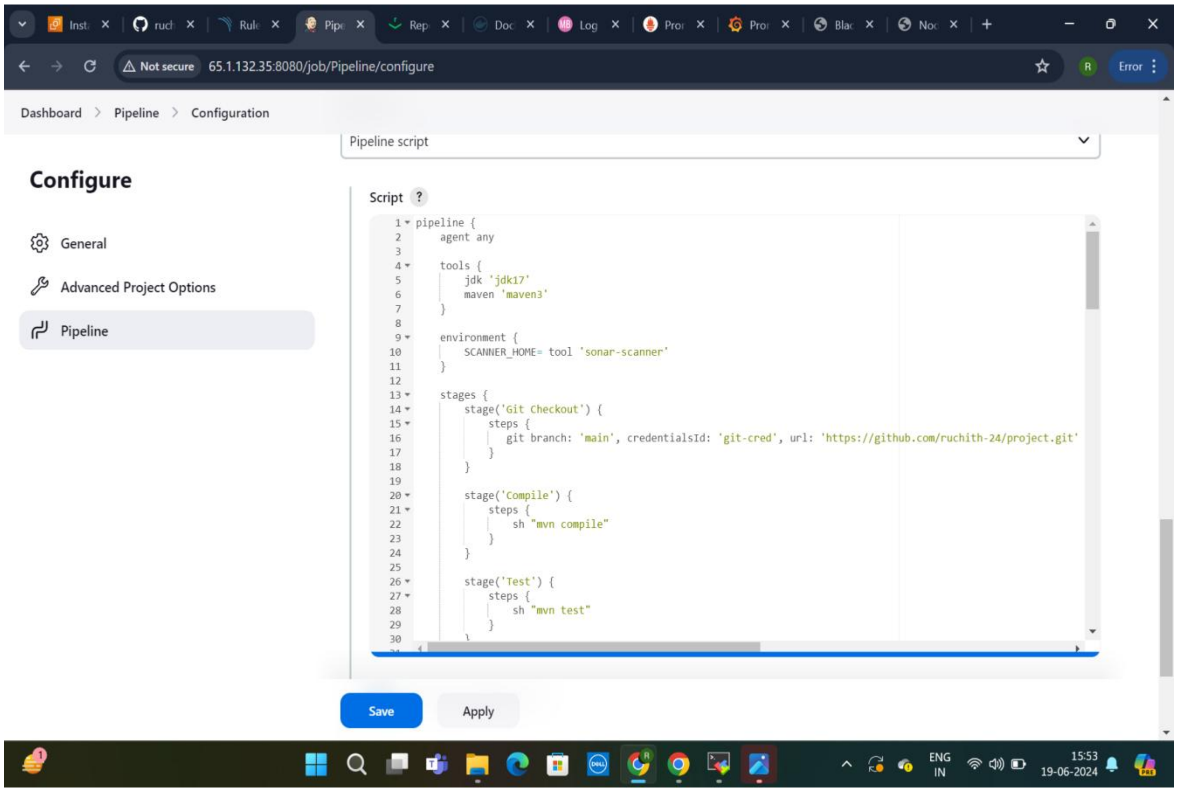
→ ☐ Restart Jenkins when installation is complete and no jobs are running

Fig 5.1.7 Configuration Jenkins

8. Go to Manage Jenkins -> Tools -> JDK Installations. In this give name as 'jdk17' and add installer as 'install from adoptium.net and select the required version.
9. Add SonarQube Scanner. Name it as 'sonar-scanner' with desired version to be selected.
10. Now move to Maven installations, name it as 'maven' any version above 3.6 to be selected.
11. Move to Docker installations naming it as 'docker'. The installation root should be 'download from docker.com'. Make sure the latest version is installed.
12. Now go to Manage Jenkins and add a pipeline naming it as 'BoardGame'.
13. Go to Configuration -> General -> Discard Old Builds. Here enter two as the maximum number of builds to keep.

Phase 3

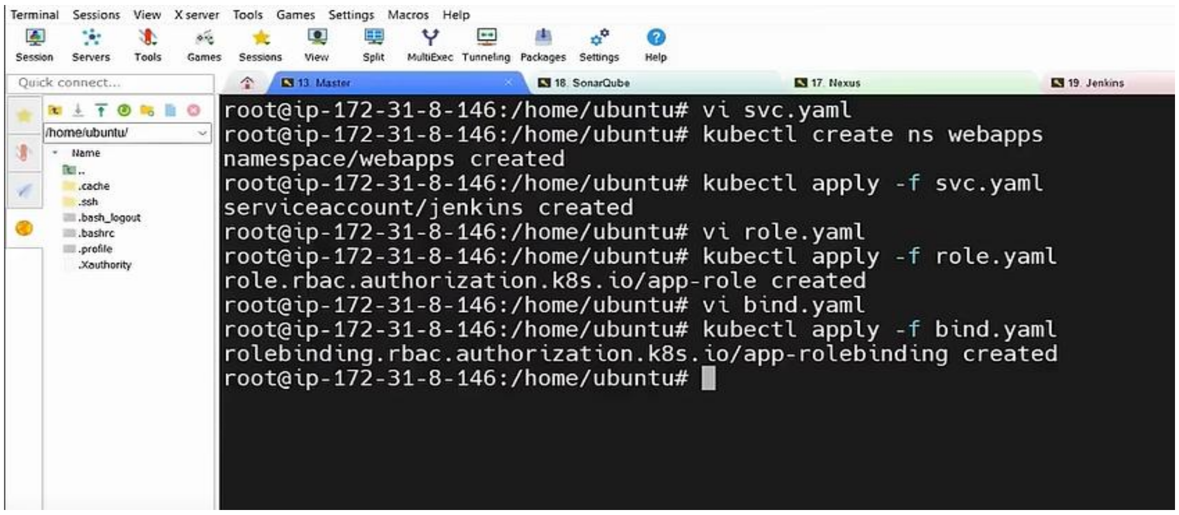
1. Now move on to Pipeline and write the pipeline script. As the primary codebase, this script serves as the backbone of our project, coordinating and integrating and executing all key components of the project.
2. There is the code for Pipeline script to be entered for every stage in the pipeline script.
3. We need to configure SonarQube server, save the script and go back to Dashboard -> Manage Jenkins -> System -> SonarQube Servers -> Add name as 'sonar' , url that is copied from token generation and the Server authentication token 'sonar-token' is to be added.
4. Dashboard -> Boardgame -> Configure -> General -> Pipeline -> Script -> Pipeline Syntax -> Steps -> withSonarQubeEnv -> Generate Pipeline Script -> paste that in the main pipeline script.
5. There are changes that need to be made as the syntax generated is not accurate.
6. Include a new stage in the pipeline script called 'Quality Gate'. For that we need to go to SonarQube Server -> Administration -> Configuration -> Webhooks -> Create Webhook
7. Name the webhook as 'jenkins' and paste its URL. Click on create.
8. Now we can build the application under the Build stage.
9. Make sure that no two stages have the same name , otherwise there will be an error.



```
pipeline {
    agent any

    tools {
        jdk 'jdk17'
        maven 'maven3'
    }

    environment {
        SCANNER_HOME= tool 'sonar-scanner'
    }

    stages {
        stage('Git Checkout') {
            steps {
               git branch: 'main', credentialsId: 'git-cred', url: 'https://github.com/ruchith-24/project.git'
            }
        }

        stage('Compile') {
            steps {
                sh "mvn compile"
            }
        }

        stage('Test') {
            steps {
                sh "mvn test"
            }
```

Fig 5.1.8 Pipeline Script

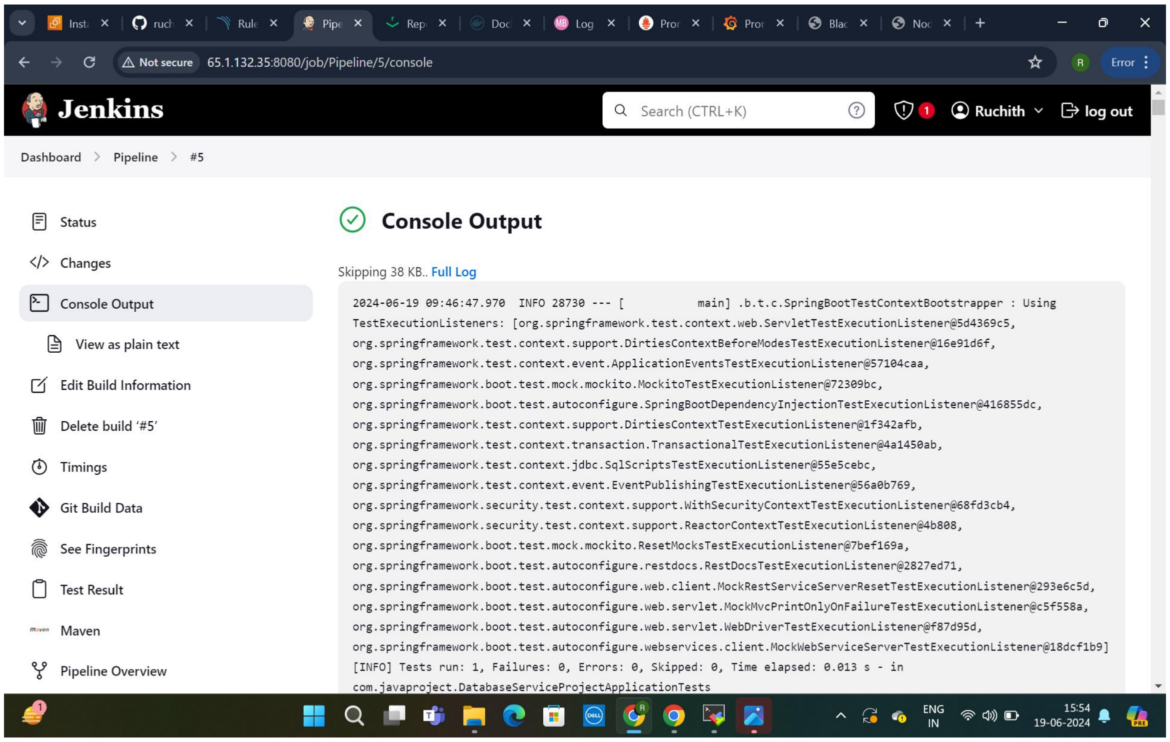
10. Next stage is 'Publish to Nexus'. For that we need to do multiple configurations. We need to add repository URLs to POM.xml file.
11. Go to Jenkins website -> Manage -> Managed File -> Config File -> Global Maven Setting -> Provide ID -> Next.
12. Under Content we provide credentials to access Nexus.
13. Under <server> the ID is maven-releases and enter the username and password for the repository. The same should be repeated to the maven-snapshots with the same credentials.
14. Back to Pipeline we continue with the script.
15. Add Docker credentials using pipeline syntax for easy processing. Attach the existing docker file in the Git repository to the pipeline script.
16. We create a service account in the MobaXterm Master session for efficient deployment.


Terminal Sessions View X server Tools Games Settings Macros Help
Session Servers Tools Games Sessions View Split MultiExec Tunneling Packages Settings Help
Quick connect...
13. Master
18. SonarQube
17. Nexus
19. Jenkins
/home/ubuntu/
Name
.cache
.ssh
.bash_logout
.bashrc
.profile
.Xauthority

```
root@ip-172-31-8-146:/home/ubuntu# vi svc.yaml
root@ip-172-31-8-146:/home/ubuntu# kubectl create ns webapps
namespace/webapps created
root@ip-172-31-8-146:/home/ubuntu# kubectl apply -f svc.yaml
serviceaccount/jenkins created
root@ip-172-31-8-146:/home/ubuntu# vi role.yaml
root@ip-172-31-8-146:/home/ubuntu# kubectl apply -f role.yaml
role.rbac.authorization.k8s.io/app-role created
root@ip-172-31-8-146:/home/ubuntu# vi bind.yaml
root@ip-172-31-8-146:/home/ubuntu# kubectl apply -f bind.yaml
rolebinding.rbac.authorization.k8s.io/app-rolebinding created
root@ip-172-31-8-146:/home/ubuntu# 
```

Fig 5.1.9 Creating of Mobaxterm Master Session

17. Enter *kubertl describe secret mysecretname -n webapps* to get a token. Copy that token and paste it as new credentials of Jenkins under secret text and name it as ‘k8-cred’ copy the same as description.
18. Configure Mail notification. Sign in to your google account and enter your password. App password will generate an application password that is different from account password. Provide the app name as ‘jenkins’. A password is generated , copy that , this password is used to send mail notifications.
19. Manage Jenkins -> System -> Extended Email Notification -> SMTP server for gmail is ‘smtp.gmail.com’ and SMTP port is 465. Advanced -> Use SSL -> Provide credentials and the recent generated password.
20. Email notification -> SMTP server is ‘smtp.gmail.com’ and Use SSL. Provide Email ID as username and generated application password as password then save it.
21. We can build the pipeline now. The console output shows the status of the pipeline.

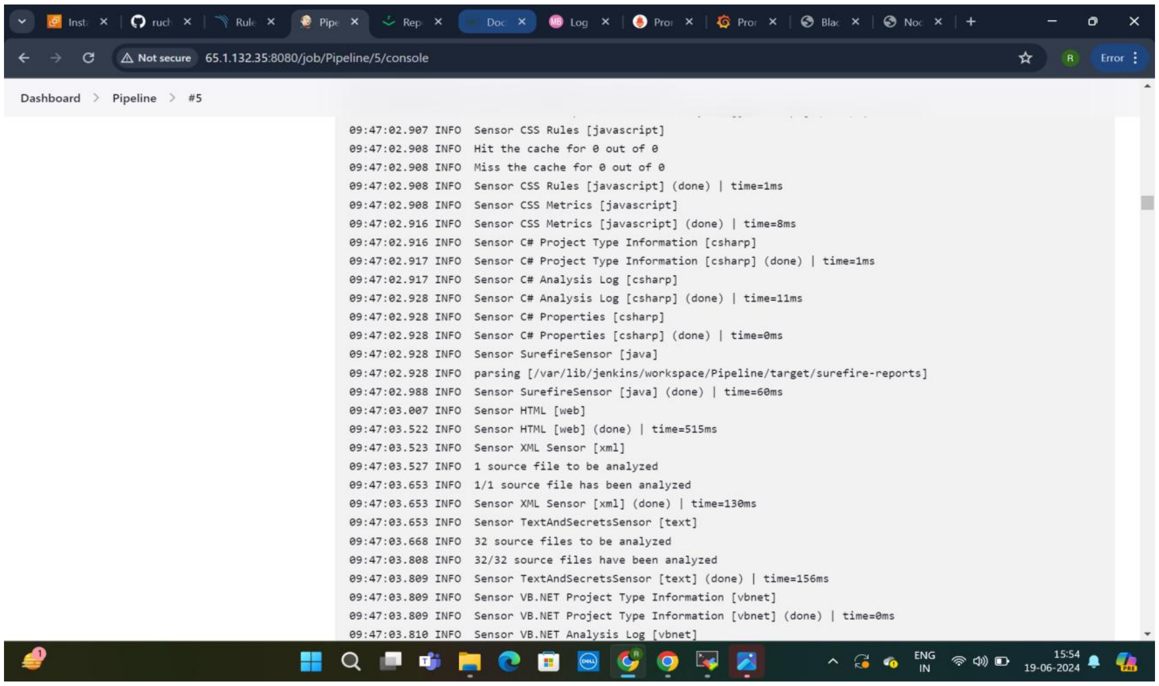


Console Output

Skipping 38 KB.. Full Log

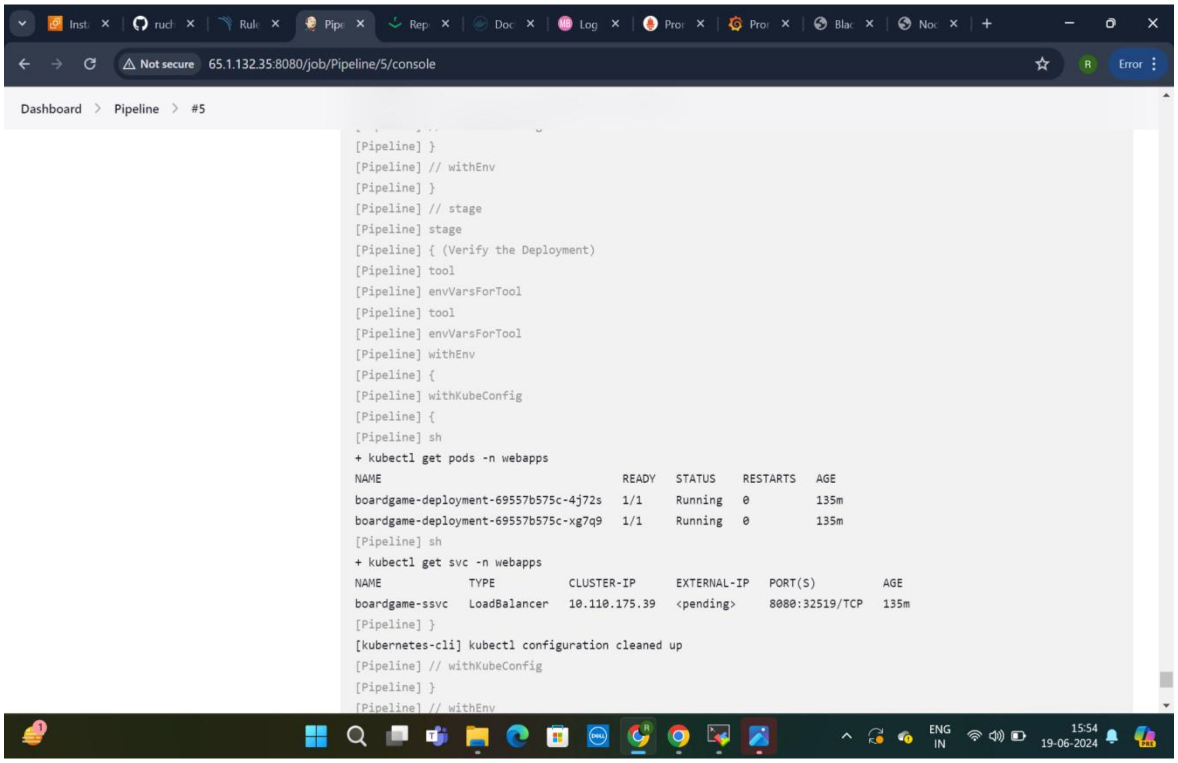
```
2024-06-19 09:46:47.970  INFO 28730 --- [           main] .b.t.c.SpringBootTestContextBootstrapper : Using
TestExecutionListeners: [org.springframework.test.context.web.ServletTestExecutionListener@5d4369c5,
org.springframework.test.context.support.DirtiesContextBeforeModesTestExecutionListener@16e91d6f,
org.springframework.test.context.event.ApplicationEventsTestExecutionListener@57104caa,
org.springframework.boot.test.mock.mockito.MockitoTestExecutionListener@72309bc,
org.springframework.boot.test.autoconfigure.SpringBootDependencyInjectionTestExecutionListener@416855dc,
org.springframework.test.context.support.DirtiesContextTestExecutionListener@1f342afb,
org.springframework.test.context.transaction.TransactionalTestExecutionListener@4a1450ab,
org.springframework.test.context.jdbc.SqlScriptsTestExecutionListener@55e5cebc,
org.springframework.test.context.event.EventPublishingTestExecutionListener@56a0b769,
org.springframework.security.test.context.support.WithSecurityContextTestExecutionListener@68fd3cb4,
org.springframework.security.test.context.support.ReactorContextTestExecutionListener@4b808,
org.springframework.boot.test.mock.mockito.ResetMocksTestExecutionListener@7bef169a,
org.springframework.boot.test.autoconfigure.restdocs.RestDocsTestExecutionListener@2827ed71,
org.springframework.boot.test.autoconfigure.web.client.MockRestServiceServerResetTestExecutionListener@293e6c5d,
org.springframework.boot.test.autoconfigure.web.servlet.MockMvcPrintOnlyOnFailureTestExecutionListener@c5f558a,
org.springframework.boot.test.autoconfigure.web.servlet.WebDriverTestExecutionListener@f87d95d,
org.springframework.boot.test.autoconfigure.webservices.client.MockWebServiceServerTestExecutionListener@18dcf1b9]
[INFO] Tests run: 1, Failures: 0, Errors: 0, Skipped: 0, Time elapsed: 0.013 s - in
com.javaproject.DatabaseServiceProjectApplicationTests
```

Fig 5.1.10 Console Output


65.1.132.35:8080/job/Pipeline/5/console
Dashboard > Pipeline > #5

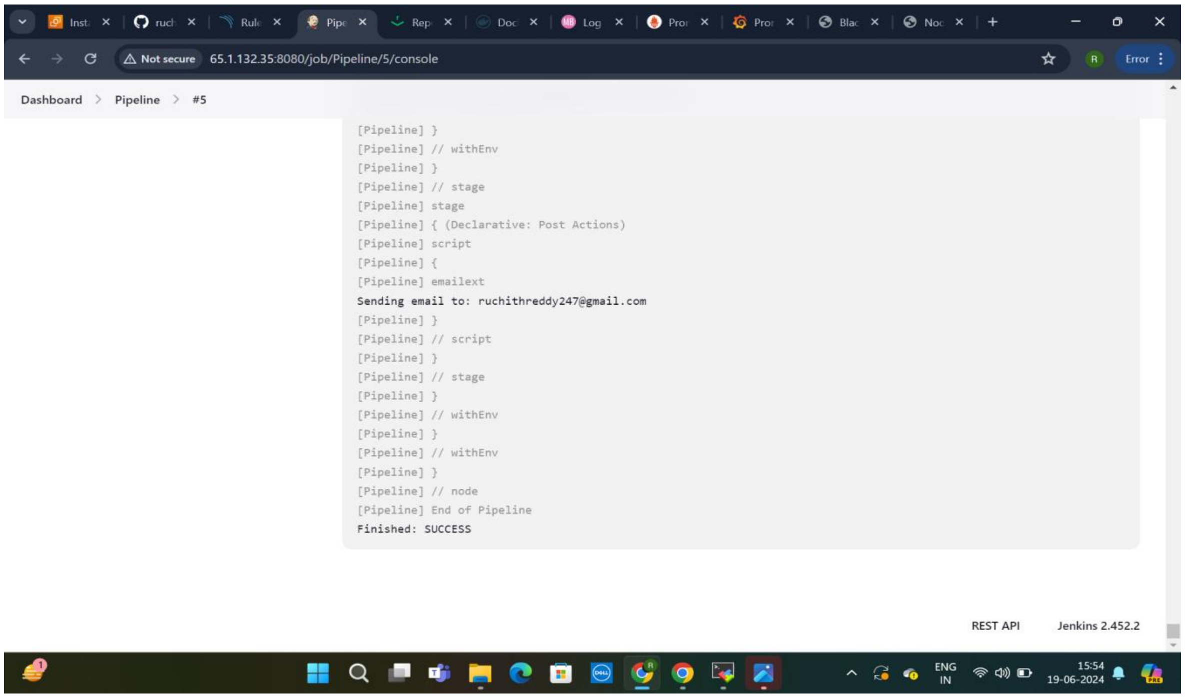
```
09:47:02.907 INFO  Sensor CSS Rules [javascript]
09:47:02.908 INFO  Hit the cache for 0 out of 0
09:47:02.908 INFO  Miss the cache for 0 out of 0
09:47:02.908 INFO  Sensor CSS Rules [javascript] (done) | time=1ms
09:47:02.908 INFO  Sensor CSS Metrics [javascript]
09:47:02.916 INFO  Sensor CSS Metrics [javascript] (done) | time=8ms
09:47:02.916 INFO  Sensor C# Project Type Information [csharp]
09:47:02.917 INFO  Sensor C# Project Type Information [csharp] (done) | time=1ms
09:47:02.917 INFO  Sensor C# Analysis Log [csharp]
09:47:02.928 INFO  Sensor C# Analysis Log [csharp] (done) | time=11ms
09:47:02.928 INFO  Sensor C# Properties [csharp]
09:47:02.928 INFO  Sensor C# Properties [csharp] (done) | time=0ms
09:47:02.928 INFO  Sensor SurefireSensor [java]
09:47:02.928 INFO  parsing [/var/lib/jenkins/workspace/Pipeline/target/surefire-reports]
09:47:02.988 INFO  Sensor SurefireSensor [java] (done) | time=60ms
09:47:03.007 INFO  Sensor HTML [web]
09:47:03.522 INFO  Sensor HTML [web] (done) | time=515ms
09:47:03.523 INFO  Sensor XML Sensor [xml]
09:47:03.527 INFO  1 source file to be analyzed
09:47:03.653 INFO  1/1 source file has been analyzed
09:47:03.653 INFO  Sensor XML Sensor [xml] (done) | time=130ms
09:47:03.653 INFO  Sensor TextAndSecretsSensor [text]
09:47:03.668 INFO  32 source files to be analyzed
09:47:03.808 INFO  32/32 source files have been analyzed
09:47:03.809 INFO  Sensor TextAndSecretsSensor [text] (done) | time=156ms
09:47:03.809 INFO  Sensor VB.NET Project Type Information [vbnet]
09:47:03.809 INFO  Sensor VB.NET Project Type Information [vbnet] (done) | time=0ms
09:47:03.810 INFO  Sensor VB.NET Analysis Log [vbnet]
```

Fig 5.1.11 Console Output



```
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Verify the Deployment)
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] tool
[Pipeline] envVarsForTool
[Pipeline] withEnv
[Pipeline] {
[Pipeline] withKubeConfig
[Pipeline] {
[Pipeline] sh
+ kubectl get pods -n webapps
NAME                                    READY   STATUS    RESTARTS   AGE
boardgame-deployment-69557b575c-4j72s   1/1     Running   0          135m
boardgame-deployment-69557b575c-xg7q9   1/1     Running   0          135m
[Pipeline] sh
+ kubectl get svc -n webapps
NAME             TYPE           CLUSTER-IP      EXTERNAL-IP   PORT(S)          AGE
boardgame-ssvc   LoadBalancer   10.110.175.39   <pending>     8080:32519/TCP   135m
[Pipeline] }
[kubernetes-cli] kubectl configuration cleaned up
[Pipeline] // withKubeConfig
[Pipeline] }
[Pipeline] // withEnv
```

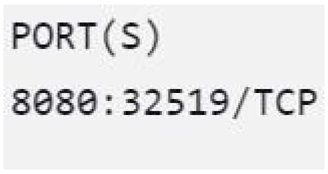
Fig 5.1.12 Console Output



```
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // stage
[Pipeline] stage
[Pipeline] { (Declarative: Post Actions)
[Pipeline] script
[Pipeline] {
[Pipeline] emailext
Sending email to: ruchithreddy247@gmail.com
[Pipeline] }
[Pipeline] // script
[Pipeline] }
[Pipeline] // stage
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // withEnv
[Pipeline] }
[Pipeline] // node
[Pipeline] End of Pipeline
Finished: SUCCESS
```

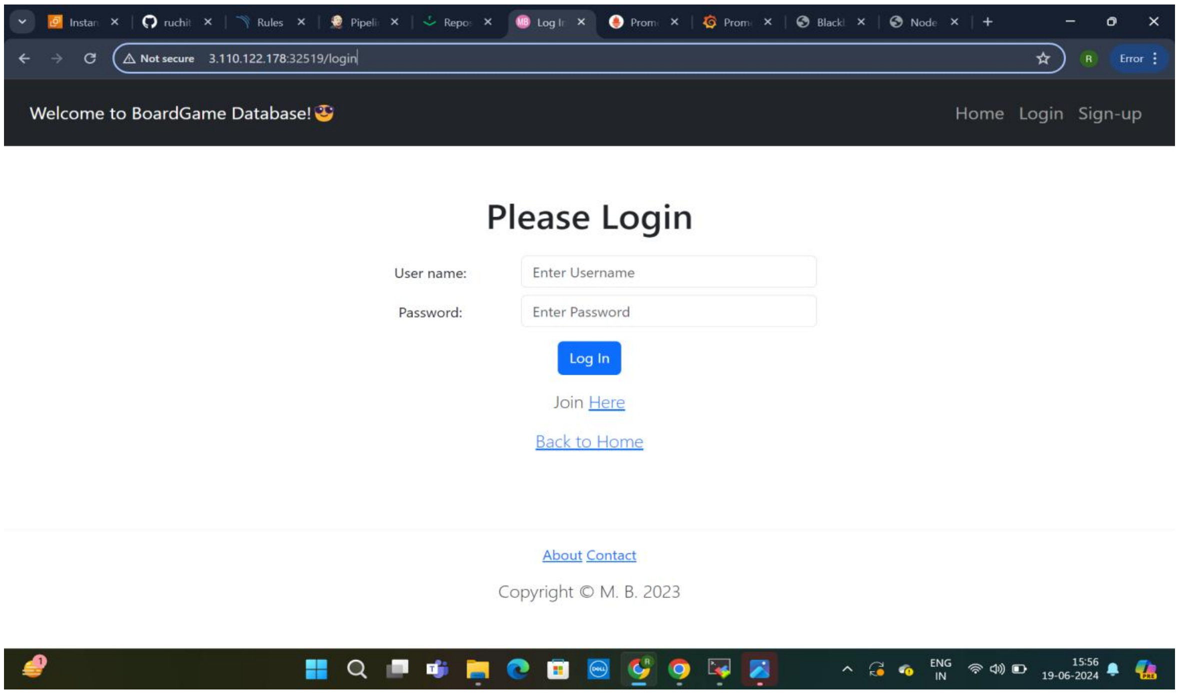
Fig 5.1.13 Console Output

22. The host port on which the application will be accessible is the following. We can access this by copying the port number and the IP address of Slave1 instance. Copy the address followed by the port number.



```
PORT(S)
8080:32519/TCP
```

Fig 5.1.14 Hosted Port Number



3.110.122.178:32519/login

Welcome to BoardGame Database!

Please Login

User name: Enter Username

Password: Enter Password

Log In

Join Here

Back to Home

About Contact

Copyright © M. B. 2023

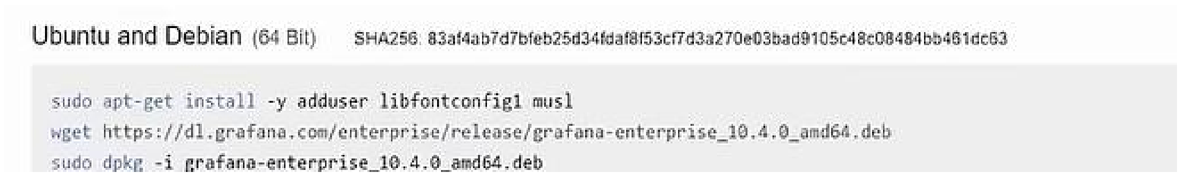
Fig 5.1.15 Login Page

Phase 4

Monitoring

1. We create a new virtual machine for monitoring the application and name it as ‘monitor’ with t2.large size and the same security.

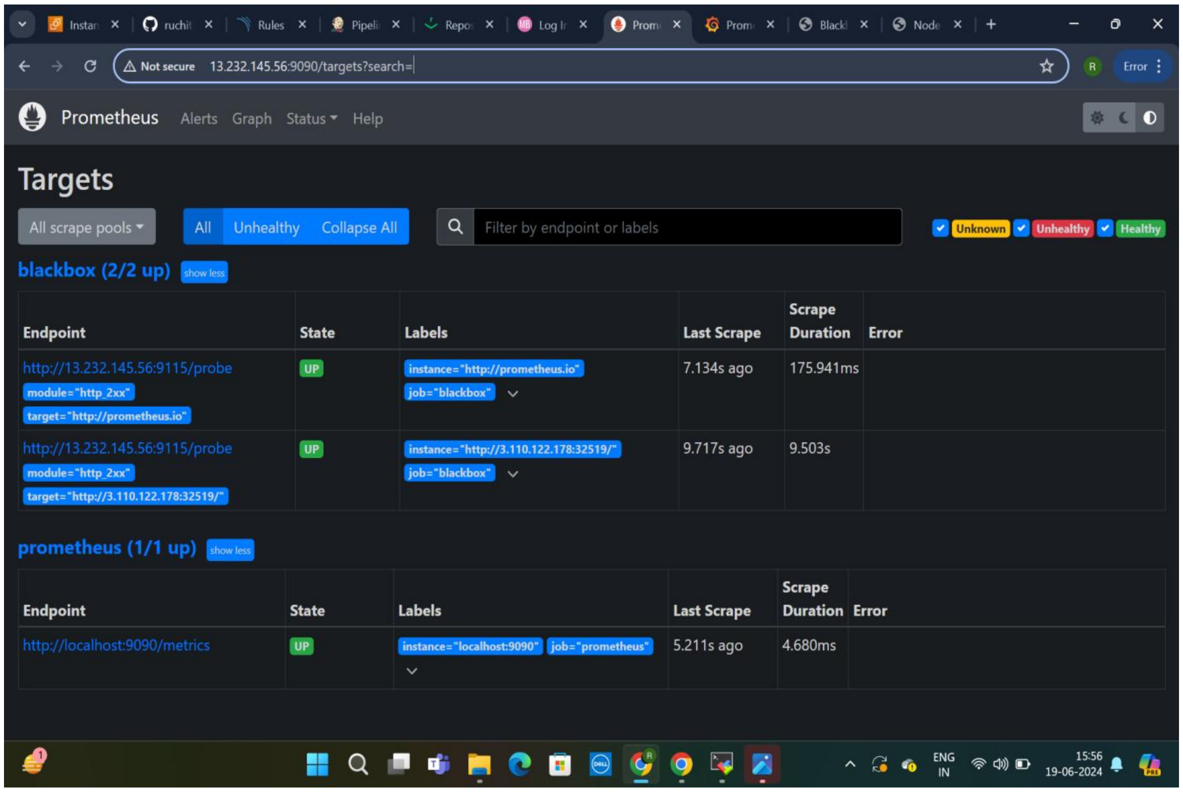
2. Copy the IP address and create a new MobaXterm session with the copied IP address. Run *sudo apt update*.
3. Start with the installation of Prometheus.Copy the download link for Linux and enter it followed by the *wget* command.
4. Install Grafana and for that go to Grafana website and run the following command on MobaXterm


Ubuntu and Debian (64 Bit) SHA256: 83af4ab7d7bfeb25d34fdaf8f53cf7d3a270e03bad9105c48c08484bb461dc63

```
sudo apt-get install -y adduser libfontconfig1 musl
wget https://dl.grafana.com/enterprise/release/grafana-enterprise_10.4.0_amd64.deb
sudo dpkg -i grafana-enterprise_10.4.0_amd64.deb
```

Fig 5.1.16 Grafana Installation

5. We can access the vms of Prometheus and Grafana using the IP address in AWS instances followed by port number 9090 and 3000 respectively.
6. Download BlackBox Exporter. Remove the Tar file of BlackBox that we do not require. We are left with an executable file of BlackBox Exporter. By default it will be running on port 9115.
7. We enable targets for the Prometheus and see that the endpoints status will be shown as 'up'. The target endpoint is our application URL.
8. Next step that we do is that we add Prometheus as a data source inside our Grafana. For that we need to go to Grafana website -> Connections -> Data sources -> Add data source -> Prometheus -> Connection -> Provide Prometheus URL -> Save and Test.
9. Search blackbox grafana dashboard -> Get this Dashboard -> Copy ID to clipboard -> Grafana -> Import dashboard -> Load the Copied ID -> Import data source as Prometheus.



Targets

blackbox (2/2 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://13.232.145.56:9115/probe module="http_2xx" target="http://prometheus.io"	UP	instance="http://prometheus.io" job="blackbox"	7.134s ago	175.941ms	
http://13.232.145.56:9115/probe module="http_2xx" target="http://3.110.122.178:32519/"	UP	instance="http://3.110.122.178:32519/" job="blackbox"	9.717s ago	9.503s	

prometheus (1/1 up)

Endpoint	State	Labels	Last Scrape	Scrape Duration	Error
http://localhost:9090/metrics	UP	instance="localhost:9090" job="prometheus"	5.211s ago	4.680ms	

Fig 5.1.17 Prometheus

5.2 SAMPLE CODE

The following is the pipeline script

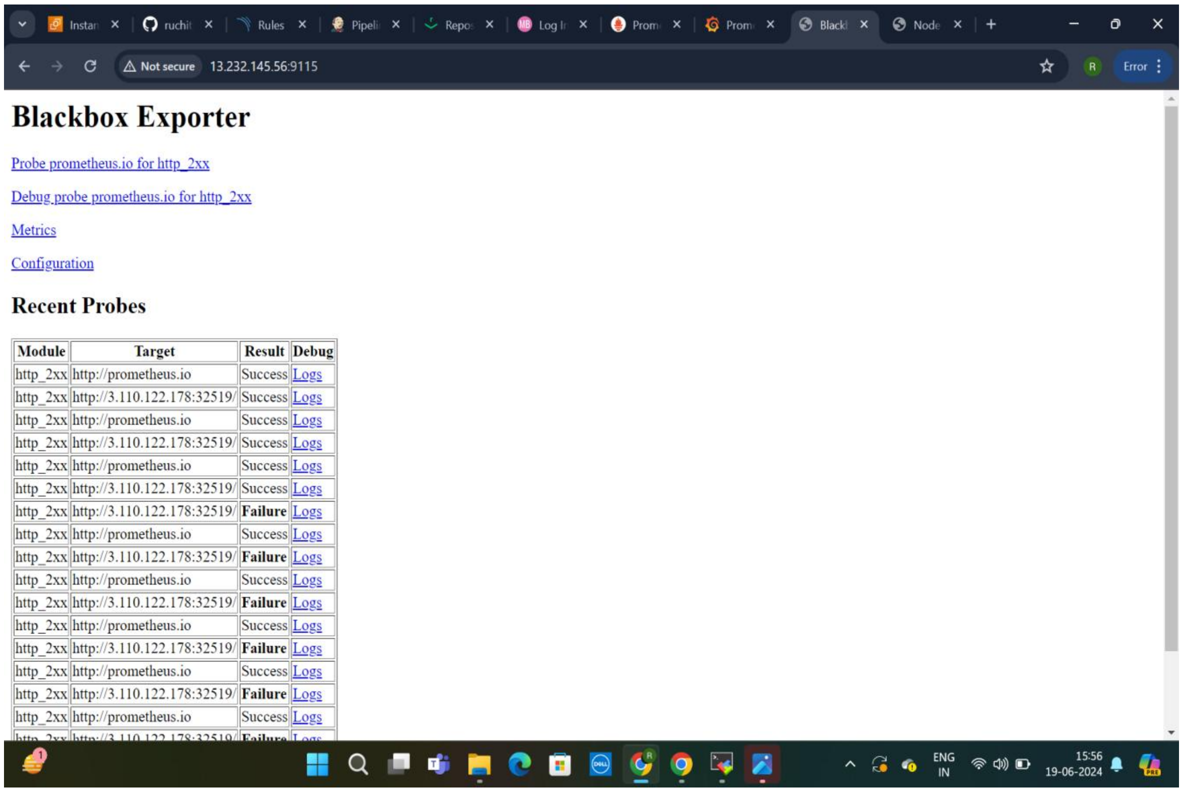
```
pipeline {
 agent any
     tools {
      jdk 'jdk17'
      maven 'maven3'
   }
```

```
    environment {
        SCANNER_HOME= tool 'sonar-scanner'
    }

    stages {
        stage('Git Checkout') {
            steps {
                git branch: 'main', credentialsId: 'git-cred', url: 'https://github.com/ruchith-24/project.git'
            }}
        stage('Compile') {
            steps {
                sh "mvn compile"}}
      stage('Test') {
            steps {
                sh "mvn test"}}
        stage('File System Scan') {
            steps {
                sh "trivy fs --format table -o trivy-fs-report.html ."}}
        stage('SonarQube Analsyis') {
            steps {
                withSonarQubeEnv('sonar') {
                    sh ''' $SCANNER_HOME/bin/sonar-scanner -Dsonar.projectName=BoardGame -Dsonar.projectKey=BoardGame \
  -Dsonar.java.binaries=. '''}}}
      stage('Quality Gate') {
         steps {
            script {
              waitForQualityGate abortPipeline: false, credentialsId: 'sonar-token' }}}
      stage('Build') {
         steps {
           sh "mvn package" }}
      stage('Publish To Nexus') {
         steps {
           withMaven(globalMavenSettingsConfig: 'global-settings', jdk: 'jdk17',
maven: 'maven3', mavenSettingsConfig: '', traceability: true) {
               sh "mvn deploy" }}}
      stage('Build & Tag Docker Image') {
         steps {
           script {
              withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {
                     sh "docker build -t ruchith24/boardshack:latest ." } } } }
      stage('Docker Image Scan') {
         steps {
            sh "trivy image --format table -o trivy-image-report.html
ruchith24/boardshack:latest" } }

      stage('Push Docker Image') {
         steps {
           script {
              withDockerRegistry(credentialsId: 'docker-cred', toolName: 'docker') {
                     sh "docker push ruchith24/boardshack:latest" } } } }
      stage('Deploy To Kubernetes') {
         steps {
           withKubeConfig(caCertificate: '', clusterName: 'kubernetes', contextName:
'', credentialsId: 'k8-cred', namespace: 'webapps', restrictKubeConfigAccess: false,
serverUrl: 'https://172.31.36.10:6443') {
                   sh "kubectl apply -f deployment-service.yaml" } } }
      stage('Verify the Deployment') {
         steps {
           withKubeConfig(caCertificate: '', clusterName: 'kubernetes', contextName:
'', credentialsId: 'k8-cred', namespace: 'webapps', restrictKubeConfigAccess: false,
serverUrl: 'https://172.31.36.10:6443') {
                   sh "kubectl get pods -n webapps"
                   sh "kubectl get svc -n webapps” } } } }
    post {
    always {
      script {
         def jobName = env.JOB_NAME
         def buildNumber = env.BUILD_NUMBER
         def pipelineStatus = currentBuild.result ?: 'UNKNOWN'
         def bannerColor = pipelineStatus.toUpperCase() == 'SUCCESS' ? 'green' :
'red'
```

```
        def body = """
          <html>
          <body>
          <div style="border: 4px solid ${bannerColor}; padding: 10px;">
          <h2>${jobName} - Build ${buildNumber}</h2>
          <div style="background-color: ${bannerColor}; padding: 10px;">
          <h3 style="color: white;">Pipeline Status:
${pipelineStatus.toUpperCase()}</h3>
          </div>
          <p>Check the <a href="${BUILD_URL}">console output</a>.</p>
          </div>
          </body>
          </html>
        """     emailext (
          subject: "${jobName} - Build ${buildNumber} -
${pipelineStatus.toUpperCase()}",
          body: body,
          to: 'ruchithreddy247@gmail.com',
          from: 'jenkins@example.com',
          replyTo: 'jenkins@example.com',
          mimeType: 'text/html',
          attachmentsPattern: 'trivy-image-report.html'
        )
    }}}}
```

6. TESTING

6.1 TESTING



Blackbox Exporter

Probe prometheus.io for http_2xx

Debug probe prometheus.io for http_2xx

Metrics

Configuration

Recent Probes

Module	Target	Result	Debug
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	Success	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	Success	Logs
http_2xx	http://3.110.122.178:32519/	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	**Failure**	Logs
http_2xx	http://prometheus.io	Success	Logs
http_2xx	http://3.110.122.178:32519/	**Failure**	Logs

Fig 6.1.1 Blackbox Exporter TestCases

Integrating the Blackbox Exporter into a CI/CD pipeline significantly enhances the monitoring capabilities during deployments, providing real-time insights into the health and performance of your web services and APIs. The Blackbox Exporter, which is part of the Prometheus ecosystem, allows you to perform various types of probes—such as HTTP, TCP, DNS, and ICMP (ping) checks—against your endpoints. This flexibility enables you to monitor different aspects of your services, such as response times, availability, and the correct functioning of web pages or APIs.

To start, you need to set up the Blackbox Exporter, which can be done by running it as a Docker container or installing it directly on your server. The exporter uses a configuration file, typically blackbox.yml, where you define your probes. For example, an HTTP probe might be set up to check that a specific endpoint returns a 2xx status code, indicating successful responses. This configuration allows you to tailor the monitoring to the specific needs of your services, ensuring that the most critical aspects are being watched closely.

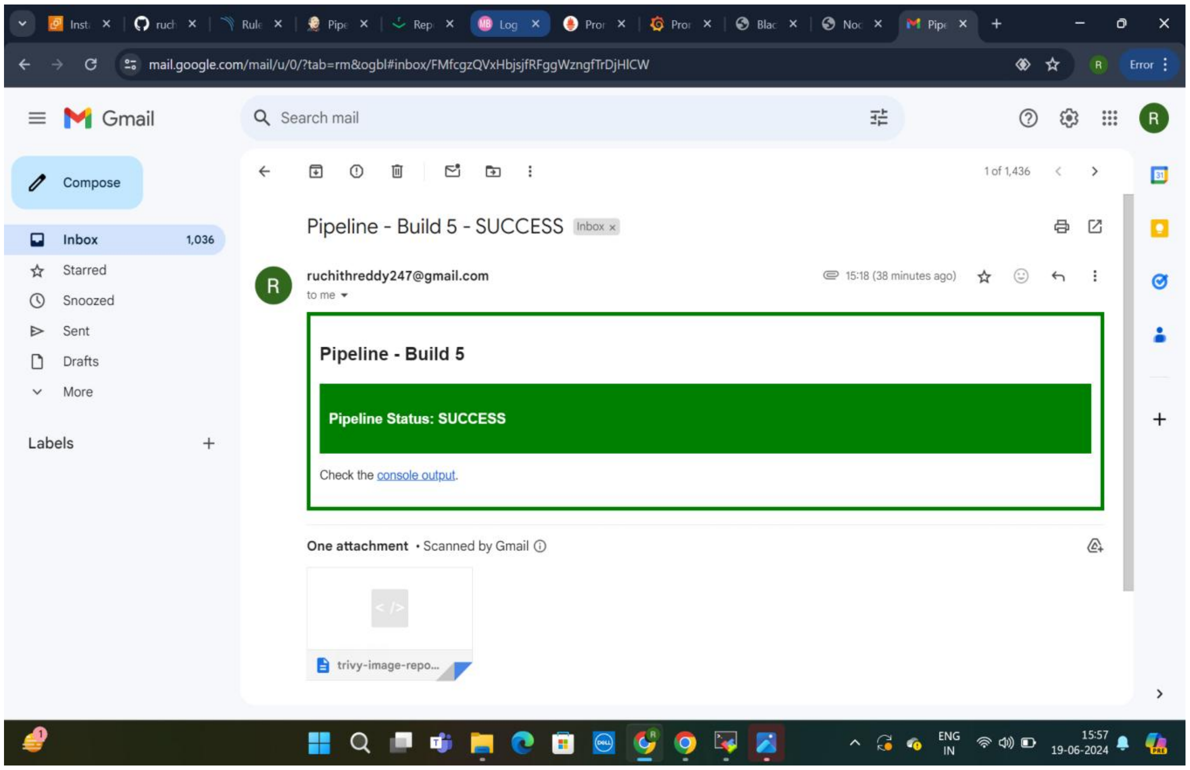
Once the Blackbox Exporter is set up, you integrate it with Prometheus by adding it as a target in the Prometheus configuration (prometheus.yml). This integration is key to making the monitoring data available for collection and visualization. Prometheus regularly scrapes the Blackbox Exporter for metrics, which can then be used to trigger alerts or feed into dashboards in Grafana. This setup ensures that any issues detected by the Blackbox Exporter, such as an endpoint going down or a significant delay in response times, are immediately visible to your operations team.

Incorporating the Blackbox Exporter into your CI/CD pipeline allows you to continuously monitor the health of your services before, during, and after deployments. This proactive approach means that you can catch issues early—potentially even before they impact users. If a deployment introduces a problem that affects service availability or performance, the monitoring system can alert you immediately, allowing for rapid response and rollback if necessary. This continuous monitoring ensures a higher level of reliability and stability in your deployment process, ultimately leading to better service quality and user satisfaction.

Success Status : The successful test cases involve the authorization of authentic users of the application , in that case the BlackBox Exporter displays results as ‘success’.

Failed Status: The unsuccessful test cases the authorization of unauthentic users of the application , in that case the BlackBox Exporter displays results as ‘failed’.

7. OUTPUT SCREEN



mail.google.com/mail/u/0/?tab=rm&ogbl#inbox/FMfcgzQVxHbjsjfRFggWzngfTrDjHlCW

Pipeline - Build 5 - SUCCESS

ruchithreddy247@gmail.com
to me

15:18 (38 minutes ago)

Pipeline - Build 5

Pipeline Status: SUCCESS

Check the console output.

One attachment • Scanned by Gmail

trivy-image-repo...

Fig 7.1 Mail Notification

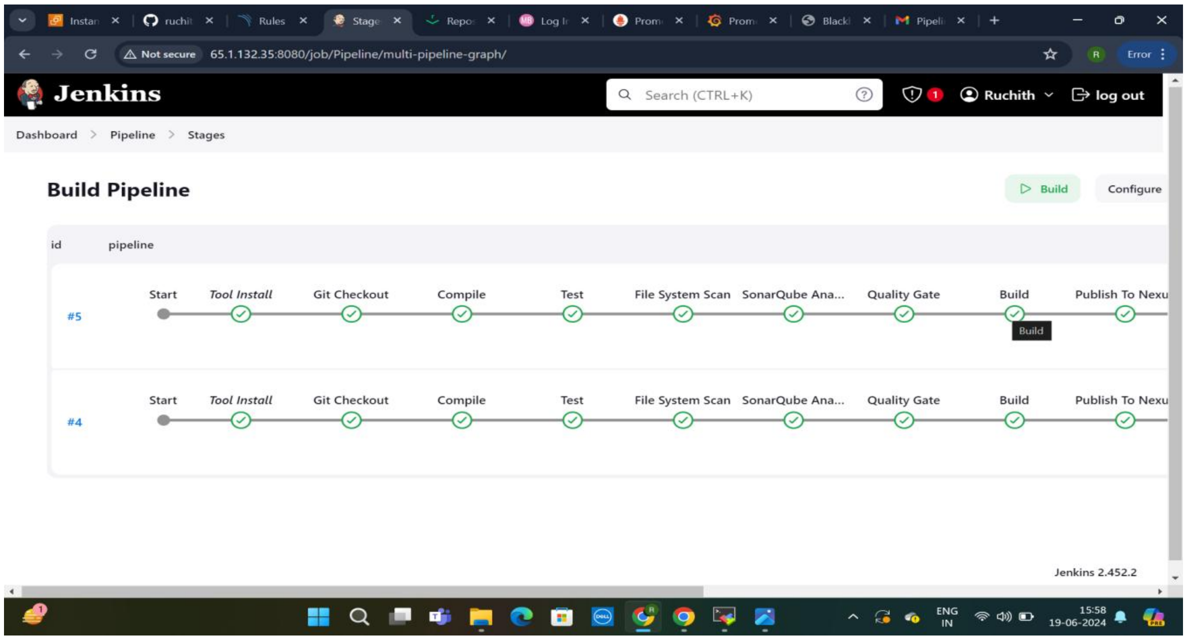


Fig 7.2 Pipeline Build

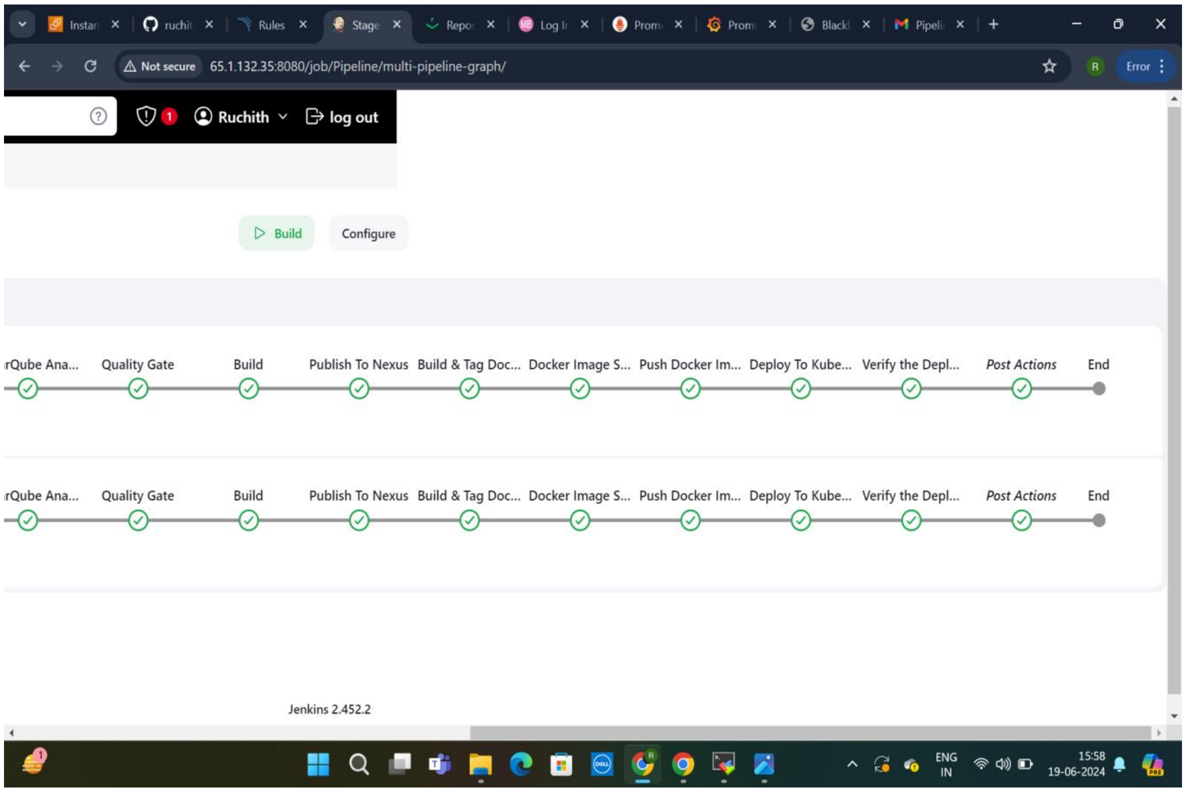
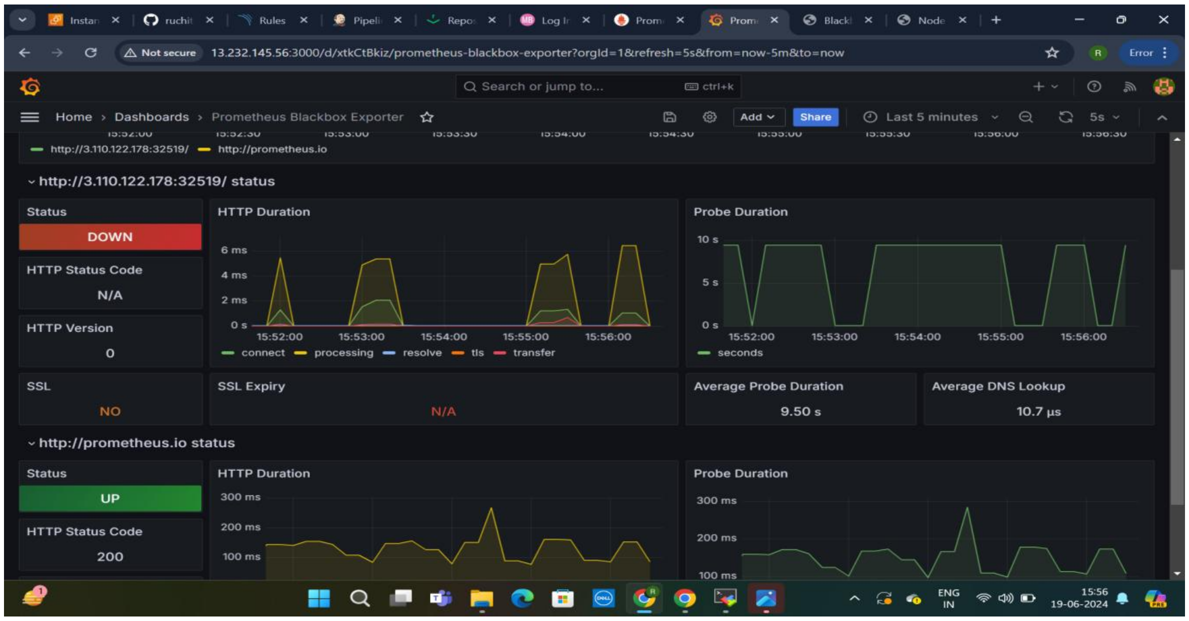


Fig 7.3 Pipeline Build



http://3.110.122.178:32519/ status
Status DOWN
HTTP Status Code N/A
HTTP Version 0
SSL NO
SSL Expiry N/A
Average Probe Duration 9.50 s
Average DNS Lookup 10.7 µs
http://prometheus.io status
Status UP
HTTP Status Code 200

Fig 7.4 Grafana Monitoring Dashboard

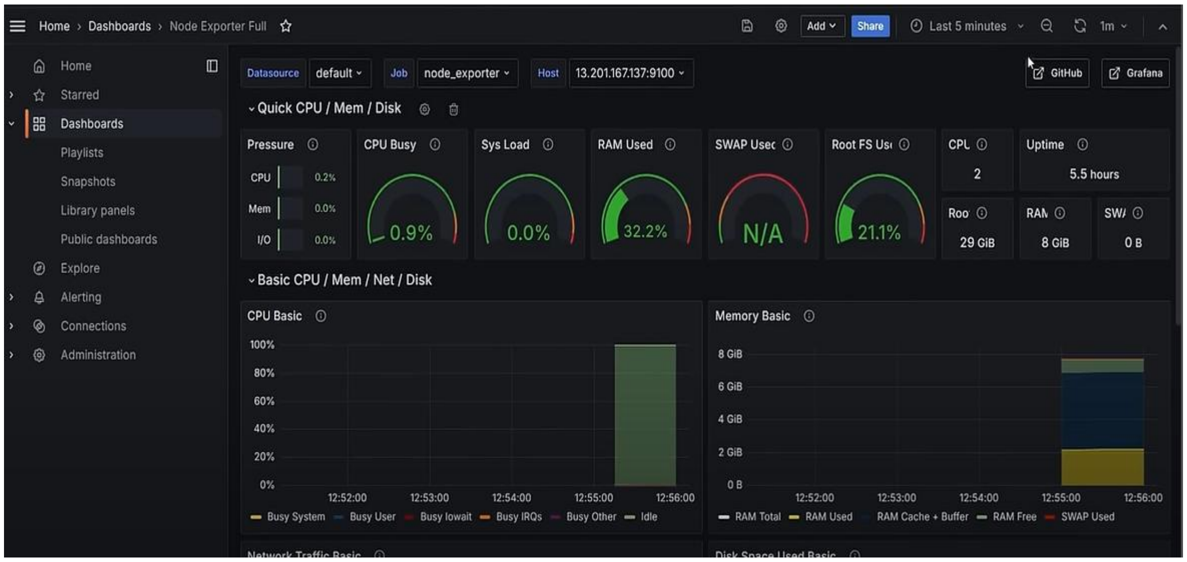


Fig 7.5 Grafana Monitoring Dashboard

8.CONCLUSION

8.1 CONCLUSION

Implementing CI/CD in Web App Development

Version Control and Repository Setup

Use a version control system (VCS) like Git and host your code on platforms such as GitHub to effectively manage code changes, releases, and feature branches, ensuring a structured and organized development process.Make sure that the repository is private and isolated to ensure maximum security and efficiency.

Automated Testing and Build Automation

Develop unit tests, integration tests, and end-to-end tests for the web app, and integrate them into your CI/CD pipeline to ensure tests are automatically executed upon each code commit. Automate the building and packaging process of the web app. The create Docker containers for your web app and its dependencies. Proceed to define Docker Compose files for local development and Kubernetes manifests for production deployments.

CI/CD Pipeline Configuration and Deployment Automation

Set up a CI/CD server, such as Jenkins or Travis CI, and configure it to monitor your version control system (VCS) for code changes. Define a CI pipeline that automatically builds and tests your web app with each code commit. Implement deployment automation scripts to provision and configure the necessary infrastructure. Use container orchestration tools like Kubernetes to manage deployments and scaling, ensuring efficient and consistent application delivery.

Monitoring and Feedback

The process involves integrating monitoring tools to track application performance and gather user feedback. Alerts and notifications are configured to ensure prompt reactions to issues in production, enhancing the reliability and responsiveness of the application.

Continuous Improvement

Continuously refine and optimize your CI/CD pipeline by leveraging metrics and feedback to identify areas for improvement. Encourage collaboration among development, testing, and operations teams to ensure a seamless feedback loop, enabling ongoing enhancements in efficiency and quality.

Continuous Integration and Deployment (CI/CD) is a transformative approach to web application development that empowers teams to build, test, and deploy code changes more efficiently and reliably. By automating key aspects of the development pipeline and implementing robust testing practices, you can deliver high-quality web apps to users faster and with fewer hiccups.

Investing in CI/CD not only improves the development process but also enhances collaboration among development, testing, and operations teams. As the landscape of web development continues to evolve, embracing CI/CD is a crucial step toward staying competitive and delivering web apps that meet the demands of today’s fast-paced digital world.

8.2 FURTHER ENCHANTMENTS

Enhancing a CI/CD pipeline involves integrating additional tools and practices to increase automation, improve quality, and ensure faster, more reliable deployments. Here are several ways to further enhance your CI/CD pipeline:

1. Automated Testing at Multiple Levels

Unit Testing: Integrate automated unit tests that run every time code is pushed to the repository. This ensures that individual components work as expected.

Integration Testing: Automate integration tests to check how different modules of your application work together. This is crucial for catching issues that only arise when components interact.

End-to-End Testing: Incorporate end-to-end testing to simulate user interactions with the application, ensuring that the application behaves as expected from the user's perspective.

Performance Testing: Include automated performance testing in your pipeline to measure how new changes impact the application's speed and scalability. Tools like JMeter or Gatling can be integrated for this purpose.

2. Infrastructure as Code (IaC)

Version-Controlled Infrastructure: Use tools like Terraform, Ansible, or AWS CloudFormation to manage your infrastructure as code. This allows you to version-control your infrastructure configurations, making deployments and rollbacks more consistent and reliable.

Automated Infrastructure Provisioning: Integrate IaC tools into your CI/CD pipeline to automatically provision and configure infrastructure as part of the deployment process. This is particularly useful in cloud environments where resources can be spun up or down on demand.

3. Continuous Security

Static Application Security Testing (SAST): Implement SAST tools that analyze your codebase for security vulnerabilities during the development process. Tools like SonarQube or Snyk can be integrated to automatically scan code as part of the CI pipeline.

Dynamic Application Security Testing (DAST): Add DAST tools to test your running application for security issues such as SQL injection, XSS, and other common vulnerabilities. These tests can be automatically triggered as part of the CD pipeline.

Container Security: If you're using Docker, integrate container security tools like Clair or Trivy to scan your container images for vulnerabilities before they are deployed.

4. Advanced Monitoring and Alerting

Service-Level Objective (SLO) Monitoring: Define and monitor SLOs for your services, such as availability, latency, and error rates. This helps ensure that your application meets the required performance standards.

9. BIBLIOGRAPHY

9.1 BOOKS REFERENCES

The following are the books referred for the understanding of concept

"The DevOps Handbook: How to Create World-Class Agility, Reliability, & Security in Technology Organizations" by Gene Kim, Patrick Debois, John Willis, and Jez Humble"

Humple, J., & Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley.

Hightower, K., Burns, B., & Beda, J. (2017). *Kubernetes Up & Running: Dive into the Future of Infrastructure*. O'Reilly Media.

9.2 WEBSITE REFERENCES

1.https://www.redhat.com/en/topics/devops/what-cicd-pipeline
2.https://about.gitlab.com/topics/ci-cd/
3.https://www.comparitech.com/
4.https://www.browserstack.com/guide/building-ci-cd-pipeline

10.APPENDICES

10.1 Software

- Jenkins
- Amazon Web Services
- GitHub
- GitBash
- Docker
- Kubernetes

10.2 Methodologies

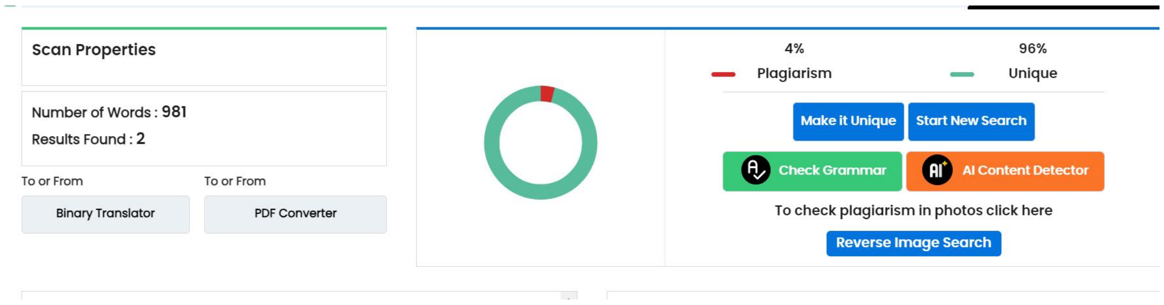
The continuous integration and continuous delivery (CI/CD) methodology is a software development practice that uses automation to streamline the process of making code changes, testing, and deploying updates. The CI/CD pipeline is a key component of this methodology and is made up of four stages

- **Source:** A change is made to the code repository
- **Build:** The code and its dependencies are merged together
- **Test:** Automated testing identifies errors and issues with the code changes
- **Deploy:** The code is moved into a new environment

10.3 Testing Methods

- Unit Tests
- Integration Tests

11 . Plagiarism Report



Scan Properties

Number of Words : 981

Results Found : 2